Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2011,26
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Formal Verification of Object-Oriented Software

Papers presented at the 2nd International Conference,
October 5-7, 2011, Turin, Italy

Bernhard Beckert
Ferruccio Damiani
Dilian Gurov (Eds.)

2011

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

Bernhard Beckert
Ferruccio Damiani
Dilian Gurov (Eds.)

Formal Verification of Object-Oriented Software

Papers presented at the 2nd International Conference,
October 5-7, 2011, Turin, Italy

Published as:
Technical Report,
Department of Informatics,
Karlsruhe Institute of Technology,
2011-26

Editors

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

Ferruccio Damiani
Dipartimento di Informatica
Università di Torino
Corso Svizzera 185, I-10149 Torino, Italy
Email: damiani@di.unito.it

Dilian Gurov
Department of Theoretical Computer Science
School of Computer Science and Communications
KTH Royal Institute of Technology
SE-100 44 Stockholm, Sweden
Email: dilian@csc.kth.se

Preface

This volume contains the invited papers, research papers, case studies, and position papers presented at the *International Conference on Formal Verification of Object-Oriented Software* (FoVeOOS 2011), that was held October 5-7, 2011 in Torino, Italy. Post-conference proceedings with revised versions of selected papers will be published within Springer's *Lecture Notes in Computer Science* series after the conference.

Formal software verification has outgrown the area of academic case studies, and industry is showing serious interest. The logical next goal is the verification of industrial software products. Most programming languages used in industrial practice are object-oriented, e.g. Java, C++, or C♯. FoVeOOS 2011 aimed to foster collaboration and interactions among researchers in this area.

FoVeOOS was organised by COST Action IC0701 (`www.cost-ic0701.org`), but it went beyond the framework of this action. The conference was open to the whole scientific community. All submitted papers were peer-reviewed, and of the 28 submissions, the Programme Committee selected 19 for presentation at the conference.

We wish to sincerely thank all the authors who submitted their work for consideration. We would also like to 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.

In addition to the contributed papers, the programme of FoVeOOS 2011 included four excellent keynote talks. We are grateful to Alan Mycroft (Cambridge University), James J. Hunt (aicas incorporated), Anindya Banerjee (IMDEA Software) and Peter Wong (Fredhopper) for accepting the invitation to address the conference.

It was a team effort that made the conference so successful. We particularly thank Sarah Grebing and Vladimir Klebanov for their hard work and help in making the conference a success. In addition, we gratefully acknowledge the generous support of COST Action IC0701, the Karlsruhe Institute of Technology, the Museo Regionale di Scienze Naturali (MRSN) of Torino, and the University of Torino.

October 2011

Bernhard Beckert
Ferruccio Damiani
Dilian Gurov

Program Committee

Bernhard Beckert	Karlsruhe Institute of Technology, Germany
Frank S. de Boer	CWI, The Netherlands
Marcello Bonsangue	Universiteit Leiden (LIACS), The Netherlands
Einar Broch Johnsen	University of Oslo, Norway
Gabriel Ciobanu	ICS, Romanian Academy, Iaşi, Romania
Mads Dam	KTH Stockholm, Sweden
Ferruccio Damiani	University of Torino, Italy
Sophia Drossopoulou	Imperial College, UK
Paola Giannini	University Piemonte Orientale, Italy
Dilian Gurov	KTH Stockholm, Sweden
Reiner Hähnle	Chalmers University of Technology, Gothenburg, Sweden
Marieke Huisman	University of Twente, The Netherlands
Bart Jacobs	Katholieke Universiteit Leuven, Belgium
Thomas Jensen	INRIA Rennes, France
Ioannis Kassios	ETH Zürich, Switzerland
Vladimir Klebanov	Karlsruhe Institute of Technology, Germany
Joe Kiniry	ITU Copenhagen, Denmark
Dorel Lucanu	University Alexandru Ioan Cuza, Romania
María del Mar Gallardo	University of Málaga, Spain
Claude Marché	INRIA Saclay-Île-de-France, France
Julio Mariño	Universidad Politécnica de Madrid, Spain
Marius Minea	Politehnica University of Timişoara, Romania
Anders Møller	University Aarhus, Denmark
Rosemary Monahan	NUI Maynooth, Ireland
Wojciech Mostowski	Radbound University Nijmegen, The Netherlands
James Noble	Victoria University of Wellington, New Zealand
Bjarte M. Østvold	Norwegian Computing Center, Norway
Olaf Owe	University of Oslo, Norway
Matthew Parkinson	Cambridge University, UK
David Pichardie	IRISA, France
Frank Piessens	Katholieke Universiteit Leuven, Belgium
Ernesto Pimentel	University of Málaga, Spain
Arnd Poetzsch-Heffter	University of Kaiserslautern, Germany
Erik Poll	University of Nijmegen, The Netherlands
António Ravara	New University of Lisbon, Portugal
Wolfgang Reif	University of Augsburg, Germany
René Rydhof Hansen	University of Aalborg, Denmark
Ina Schaefer	Technical University of Braunschweig, Germany
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
Amiram Yehudai	University of Tel Aviv, Israel
Elena Zucca	University of Genova, Italy

Program Co-Chairs

Ferruccio Damiani	University of Torino, Italy
Dilian Gurov	KTH Stockholm, Sweden

Organising Committee

Bernhard Beckert	Karlsruhe Institute of Technology, Germany
Sara Capecchi	University of Torino, Italy
Ferruccio Damiani	University of Torino, Italy
Dilian Gurov	KTH Stockholm, Sweden
Vladimir Klebanov	Karlsruhe Institute of Technology, Germany
Luca Padovani	University of Torino, Italy

Sponsoring Institutions

COST Action IC0701 “Formal Verification of Object-Oriented Software”
Karlsruhe Institute of Technology
Museo Regionale di Scienze Naturali (MRSN), Torino
University of Torino

Additional Referees

Richard Bubel
Delphine Demange
Gidon Ernst
Axel Habermaier
Michiel Helvensteijn
Ilham W. Kurnia
Jurriaan Rot

Table of Contents

Abstracts of Invited Talks

Contributed Papers

Abstracts of Invited Talks

Local Reasoning for Verification of Object-Based Programs using First-Order Assertions

Anindya Banerjee

IMDEA Software
anindya.banerjee@imdea.org

Shared mutable objects pose challenges in reasoning, especially for data abstraction and modularity. We present a logic for error-avoiding partial correctness of programs featuring shared mutable objects. Using a first order assertion language, the logic provides heap-local reasoning about mutation and separation, via ghost fields and variables of type "region" (finite sets of object references). We show the logic in use in proving the correctness of the composite design pattern. We also show how the logic can be used to reason about the hiding of internal invariants in a procedure specification and how to perform client reasoning in a manner that does not invalidate the internal invariants.

Joint work with David A. Naumann and Stan Rosenberg (Stevens Institute of Technology).

Using Kilim's Isolation Types for Multicore Efficiency

Alan Mycroft

Cambridge University
am@cl.cam.ac.uk

We identify a 'memory isolation' property which enables multi-core programs to avoid slowdown due to cache contention. We give a tutorial on existing work on Kilim and its isolation-type system building bridges with both substructural types and memory isolation.

The Practical Application of Formal Methods: Where is the Benefit for Industry?

James J. Hunt

aicas GmbH
Haid-und-Neu-Str. 18, D-76131 Karlsruhe, Germany
jjh@aicas.com
http://www.aicas.com/JamesJeffersHunt.html/

Abstract. Though the use of formal methods for software verification has progress tremendously in the last ten year, its take up in industry has been meager, but with the right emphasis this could change dramatically. Software certification standards have started to take formal methods seriously as an alternative to testing. By focusing on practical issues such as common specification languages, adaption to industrial processes for safety certification, scalability, training, and synergies between tools, common reservations about using formal methods could be lain to rest. This could help formal methods become the center of software engineering in the coming decade.

1 Introduction

Though the use of formal methods for software verification has progress tremendously in the last ten year, its take up in industry has been meager. There appear to be three main reasons for this: the perceived difficult of using formal methods, the focus of the formal methods community on theoretically interesting problems rather than the engineering work needed for applying formal methods to the practical concerns of applying those methods, and the lack of cooperation between different groups within formal methods community. The question is, how can these issues be resolved?

2 Application Domains

Before answering this question, one must first understand which applications areas are most likely to yield positive results. Software engineers are notorious for not wanting to change the way they do their development. Anyone who has ever tried to change the development process will understand this. Unless the team already has a problem that is causing pain, most team members will find any excuse not to change. There are areas tough, where one can expect that there will be pain.

2.1 Security

An obvious area is security. This has always been the area where formal methods has been most appreciated. The US Department of Defense has been a major source of funding for investigating the application of formal methods to security. Banks have also shown interested for certain types of applications, as illustrated by the Mondex case study[12]. The main goal is to ensure that a given system guarantees some aspect of ensuring that information is only visible to the agents with the proper privileges and the actions on that data are correctly executed. This is mainly a question of information access and transmission, identity management, and program functional correctness. In some sense, security is a special case of safety, i.e., safety with some special system requirements. The highest Evaluation Assurance Level (EAL) of the Common Criteria for Information Technology Security Evaluation, an international standard for certifying security relevant systems (ISO/IEC 15408), requires the use of formal methods at the highest level.

2.2 Safety

System safety is not just interesting as a basis for security, but also for the preservation of human life and livelihood. Ensuring the safe operation of systems, particularly embedded systems, is becoming increasingly necessary as ever more systems are automated. Likewise, safety-standards are increasingly recognizing the need, or at least the value of formal methods for ensuring safety. Avionics software certification standards are a good example of this.

The current software certification standards for avionics (DO-178B/ED-12B and DO-278/ED-109) are based on the implicit use of procedural programming languages and specification-base verification through testing. The recently approved replacements (DO-178C/ED-12C and DO-278A/ED-109A) have technology specific supplements on object-oriented technology (ED-217), formal methods (ED-218), and model-based development (ED-216), as well as a companion document on tool qualification (ED-215)[1]. What is interesting for the use of formal methods for object-oriented system design and development is that there is now a clear role for the use of formal methods in verifying such systems and a means of qualifying the formal-methods-based tools for this use.

The Object-Oriented Technology and Related Techniques Supplement[7] address those aspects of certification that are specific to object-oriented technology as well as some related techniques that are commonly associated with but not limited to object-oriented languages. One central issue, is the use of dynamic dispatch with inheritance. Here the supplement clearly identifies Liskov Substitution Principle (LSP) as the primary means of ensuring proper subclassing and substitution to minimizing the testing burden. Formal methods is clearly identified as a sufficient means for verifying LSP without the need for additional testing.

[1] The DO numbers for the US have not yet been assigned.

The Formal Methods Supplement[8] proscribes how formal methods can be used for verification. In the past, formal methods might only be used to reduce the need for code inspection, but we not permitted to be used to reduce the need for testing. The new supplement provides clear guidelines for replacing testing with formal analysis. Issues of soundness and completeness of the tools used are addressed. Some testing will always be required, because that is the only verification environment that takes the complete system, processor, memory, I/O, etc., and the external environment into account. This is particularly true of integration testing. Still much of unit testing could be eliminated, which should reduce overall cost. The supplement also makes clear that certain properties of a system are best addressed with the help of formal methods.

Finally, the Software Tool Qualification Considerations[6] standard describes how tools used in the software development process should be qualified. Formal analysis tools are not part of an aviation application, but they do either contribute to the code or make important determination about the quality of code. Tool qualification is about the processes necessary to demonstrate that tools functions correctly and the degree to which a tools needs to fulfill the same certification requirements as actual flight software. This also depends on whether or not the output of the tool can be independently verified.

Table 1. Tool Qualification Categories

DO-178B / ED-12B Tool Category & Definition	**DO-178C / ED-12C Tool Qualification Criteria & Definition**
Development tools: tools whose output is part of airborne software and thus can introduce errors.	**Criteria 1:** tool whose output is part of the resulting software and could insert errors.
Verification tools: Tools that cannot introduce errors, but may fail to detect them.	**Criteria 2:** A tool that automates the verification process and thus could fail to detect an error, and whose output is used to justify the elimination or reduction of – verification process not automated by tool or – development process which could impact the resulting software. **Criteria 3:** A tool that, within the scope of its intended use, could fail to detect an error.

The Tool Qualification supplement breaks tools down according to how their output is used in the software certification process. In DO-178B, only two categories of tools where considered: tools that contributed to the executable code in an avionics system and tools that are use to verify this code. In DO-178C, there is a new category of tools: those that automate parts of the verification process. Table 1 describes these three tool criteria.

Table 2. Tool Qualification Levels

Software	Criteria		
Level	1	2	3
A	TQL-1	TQL-4	TQL-5
B	TQL-2	TQL-4	TQL-5
C	TQL-3	TQL-5	TQL-5
D	TQL-4	TQL-5	TQL-5

These criteria are used to determine the level of work necessary to qualify a given tool for use in creating and certifying aviation software. Table 2 depicts the relationship between the software criticallity level of the software being developed, the criteria level of the tool and the tool qualification level (TQL) governing the qualification of the given tool. There are five tool qualification levels 1 through 5, which are equivalent to software criticallity levels A though E respectively. For example, a criteria one tool used for producing level A software must be developed with the same rigor as any level A software, whereas a criteria 3 tool must to be developed with the rigor of level C software.

Table 3. Tool Qualification Equivalences

DO-178B/ED-12B Qualification Type	DO-178B/ED-12B Software Level	DO-178C/ED-12C Qualification Level
Development	A	1
Development	B	2
Development	C	3
Development	D	4
Verification	All	4 or 5

For criteria one tools, this is no different than the rules for DO-178B; but for the other criteria, it represents a rigor that more explicitly aligns to that of the software being developed with the tool (Figure 3).

More importantly, the tools qualification supplement also provides guidance for tool reuse. Figure 1 depicts the process of deciding what needs to be done at the planning level when a tool is reused in a new project. When no change has been made, the tool can be reused easily; when the operation environment changes, only revalidation is needed; otherwise, any changes will require changes to requirements to justify code changes, as well as revalidation.

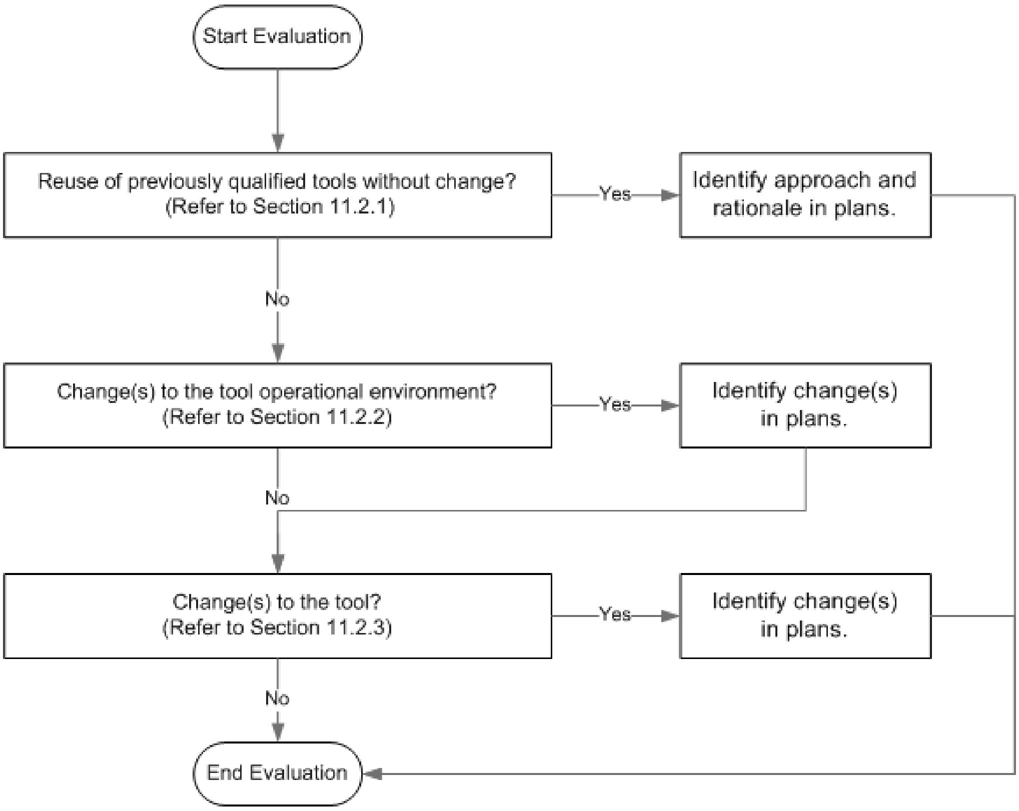

Start Evaluation
Reuse of previously qualified tools without change? (Refer to Section 11.2.1)
Yes
Identify approach and rationale in plans.
No
Change(s) to the tool operational environment? (Refer to Section 11.2.2)
Yes
Identify change(s) in plans.
No
Change(s) to the tool? (Refer to Section 11.2.3)
Yes
Identify change(s) in plans.
No
End Evaluation

Fig. 1. Tool Reuse

2.3 Multicore

Aside from safety and security, the advent of multicore systems will give software developers a reason to reevaluate the way software is written. As the number of cores in a processor increase, so does the scope for parallel execution. This means the developers will be increasingly faced with errors due to missing or improper synchronization yielding deadlocks, livelocks, and data corruption. These kinds of errors typically require a global view of program execution, which is both hard to see with inspection and hard to test. Formal methods can provide both better paradigms for design and coding parallelism and better tools for analyzing these global concerns.

3 Impedances

There are several commercial companies that sell so called static analysis tools to the software development community. So called not because they are static analysis tools, since most formal verification tools are static as well, but because the term is misused to refer to just tools that do not used any formal approach.

An informal survey by the author seems to suggest that most tools available are informal, not because the developers do not understand issues of soundness and completeness, but rather that uses are not keen on formal tools.

At first glance, this may appear strange. One would expect that a software engineer would like to produce the best possible product and that formal methods could help them do this is a short period of time. Unfortunately, human nature play a strong role. It is actually quite understandable that developers would resist using formal methods. Firstly, most engineers do not really understand the background, so there is a large learning curve. This is hard, not just because no one likes to feel like they do not understand their own business, but also because they may not believe they have the time to devote to understanding formal methods. Secondly, it would probably mean rethinking how a system is structured in the first place, which becomes more costly as development progresses.

A good example of this is to discuss Liskov Substitution Principle (LSP) with a software developer. It should be clear that the testing required to show that a system is correct is reduced by LSP's application, but most software developers do not understand this. This is in part because of the seductive effect of using subclassing for code reuse. More importantly, the realization that type checking only a partial check is lacking. After all, the type system of a language is just a proxy for the categories of objects behind the type system. In an object-oriented language, it is up to the user to specify and maintain the consistency of the categories with respect to their types as used in the type system. Most problems with fulfilling LSP can be resolved with architectural changes.

Pattern-based checkers fit the current engineering mind set better. They run quickly and can point out errors that are simple to fix. They also do not require much in the way of additional understanding. This makes it easy for the engineer to say that they used the tools and could prevent these errors from being propagated to test. They do not require restructuring of the application to fix what are often considered to be theoretical problems. One does not want to be confronted with a deep architectural problem late in the design phase.

This means that part of the solution is education. It is not enough for people to understand what formal methods are and for that they are good, but also what design principle should one consider when building a system. For example, if one thinks about it, it may be obvious that conforming to LSP has an impact on the architecture of a system; but how many people have though about this enough that considering LSP during system modeling comes naturally? There are certainly other principles as well. In fact, relying on design principles established from experience with formal methods may be the only way of building complex parallel systems that are reliable.

4 Usability

Once one is convinced that formal methods are necessary for building robust and reliable system, then comes the next hurdle. There are many tools available, but then come the questions.

- What is the best tool or tools to use?
- How do they fit into my process?
- Are they integrated into my IDE?
- Are they integrated into my build service?
- Can I understand their results?
- When must a user intervene?
- Can a developer understand what to do when intervention is necessary?
- How do I know that I can rely on their results?
- What happens to existing work when the tool is updated?

There are surely others as well. It all boils down to can the theory be used to improve software robustness and reliability in practice?

A tool or a set of tools is not enough. One needs to understand how the tools work in the software development process and where they fit into their process. What new steps must be included? Are the cost of these new steps worth the results? Where in the process should they be applied? What analysis needs to be done when a requirement, a model, or some part of the code are changed?

Formal methods must become part of the software engineering process and formal-methods-based tools have to be developed with a software engineering process. There is a certain bootstrapping problem here. It would be advantageous to use a tool in its own software development process and validate itself, but bringing the further development of a tool into such a process later is also good. The closer this process is to a process acceptable for safety-critical software community the better.

This move would provide a wealth of experience with using each tool in a software development process. It should also demonstrate that the given tool is scalable. This may well require the specification of a large set of library code, but this will be needed for other systems as well.

The reason for targeting a safety-critical software development process is that using formal methods should make the process less costly so that other domains might be tempted to use formal methods as well. After all, using formal methods should result in a reduced need for testing as well as be able to cover attributes of design that are not easy to find via testing. This should also encourage the combination of formal-methods-based tools in the process.

When a group uses its own tools the initiated can easily use the tools and cross fertilization would also be enhanced, but what about other practitioners? It is already difficult enough to take a C programmer and have him start developing in Java, but what about making him learn to use Coq or Isabelle? Languages and tools need to be designed for the intended users. That means bringing formal methods to the users domain not visa verse. It does not help the user much if the program is first translated into an unfamiliar language before it can be proven to be correct.

5 Languages

Despite what the UML and Model Driven Engineering community believes, language is still important, not only for software development, but also for specification. The choice of languages effects both the difficulty of generating code and verifying code correctness. There is no sense waisting effort on poorly designed languages.

5.1 Implementation

Some argue that the most popular programming languages should be supported by the formal methods community, but this is not necessarily a wise choice. Certainly, a focus on languages that are actually in use helps acceptance, but some languages are so poorly design that they are more apt end up supporting the criticism that formal methods cannot be used on "real" programming languages. Even a successful attempt will require extensive effort to circumvent poor syntax and semantics.

C and C++ are good examples, as they are often used for embedded system. However, for safety-critical systems, one is expected not to allocate memory dynamically and only use a subset of the language such as defined by MISRA[5]. With these restrictions, one may well be able to model the program as a state machine or a pushdown automaton, where a domain specific language may be a better choice.

In any case, both language have poorly designed semantics with no well defined memory model. On top of that, they use a preprocessor with a syntax that is not compatible with the base language, which adds additional complexity to the analysis problem. Perhaps one should consider defining a variant of C for low-level programming that resolves these issues and just say no to C++.

More complex programs are better done with a language that has some runtime support. This should include

- a memory model,
- a well defined syntax and semantics,
- static typing, and
- a realtime garbage collector.

It should not include

- pointer manipulation, and
- a preprocessor.

These attributes should make both generating code from models and analysis code correctness simpler.

5.2 Specification

More important than the implementation language is specification languages. The sooner one uses a formal notation for specification, the easier it is to use formal methods to ensure program correctness, for both functional and nonfunctional aspects of a system. Once a formal specification exists, it must be refined during the development process. Currently, this refinement process is poorly supported.

There are different languages available for each of the development process stages. One might use a language such as Z Notation[11] or Alloy[3] for the initial or high-level specification. Then one might use Object Constraint Language[2] (OCL) at the modeling level, where the specification can be related to the system architecture. Finally, one mightuse Java Modeling Language[4] (JML) during code development and maybe even Bytecode Modeling Language (BML) for the bytecode. But how can one refine one to the next?

The easiest refinement is from JML to BML, though this is hardly a refinement at all, but a representation change. Still recoding JML as Java annotations and using the annotation internal format to represent BML would be a good step in the right direction. The annotation mechanism would itself need extension, but this is just a standardization problem not a technical one. One could even imaging extending BML to machine code by providing an encoding for ELF files. Other languages such as Scala that compile to Java Bytecode could use a similar approach.

By having the specification at the source code, bytecode, and machine code level, different tools can share specification information. For example, a proof checker at the machine code level could not only ensure that the code one has actually fulfills the proof provided with it, one could also rule out the possibility that the compiler inserted some error. One could also minimized the tool qualification effort by qualifying the proof checker instead of the verification tool itself. As another example, a resource analysis tool could use general program attributes such as loop and recursion invariants encoded in a specification language for bounding execution contexts, while a program verification tool could prove the correctness of these attributes.

More difficult is the refinement from initial specification to model specification and code specification. To start with, neither OCL nor JML have a clear separation of specification and proof support information such as loop invariants. Still, there should be a subset of JML and OCL that are equivalent for specifying the behavior of a method and invariants on a class, but this subset along with the translation from one language to the other needs a formal specification. The refinement from an initial specification to a model specification is still more difficult.

6 Synergies

No one technique, formal or not, can cover all verification issues. There is a tendency to try to use ones own tool for as many problems as possible. Though interesting for comparison purposes, it is not half as useful as showing which tool is best for solving with verification or transformation problem and how various tools can be combined to provide a complete design process.

As can be seen from the Formal Methods Supplement to DO-178C/ED-12C and DO-278A/ED-109B, testing will always be part of the process, but formal methods can also be used to assist testing directly. For instance, the KeY tool[1], a deductive program verification tool originally from the University of Karlsruhe, has been adapted for generating specification based tests and test vectors. By using the KeY verification engine to identify all paths through the code for generating test vectors and JML postconditions to generate code to evaluated the results of a test, a full set of unit tests for a given method can be generated. By using this tool, a significant amount of manual test writing could be avoided. The side benefit is to get developers to consider writing specification in a formal language, thus making the next step easier.

The HATS project is a good starting point for putting together a full set of tools. Still, much more needs to be done to see what tools work well together and how they can be integrated into the software development process. A good exercise is to ask what role can a given tool play in the avionics software certification process, what effort is needed to qualify the tool, what other tools could be used for support tool qualification and avionics software certification, and what must be done to make the tool usable in the process?

7 Conclusion

It is the authors opinion that computer science is not really a science, but an engineering discipline. There is the same relationship between physics and other engineering disciplines as there is with mathematics and computer science. This is just as true for formal methods as any other computer science discipline. Theory is not enough. Without solid engineering and experimentation, formal methods will not take its proper place in the software engineering toolkit. The theory is now well enough advanced that the focus should be one engineering and demonstrating tangible results. To make formal methods more useful in practice, the main focus should be on

- using common specification languages,
- implementation languages that are not just well know but well suited to formal analysis,
- industrial processes for safety certification,
- scalability (which may include having a body of prespecified libraries),
- finding synergies between tools, and
- eating your own dog food, i.e., using your tool on your tool.

Conferences such as this one should support this change by asking for more practical results.

References

1. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag, 2007.
2. T. Clark and J. Warmer, editors. *Object Modeling with the OCL. The Rationale behind the Object Constraint Language*, volume 2263 of *LNCS*. Springer, 2002.
3. D. Jackson. Alloy: A logical modelling language. In *ZB*, page 1, 2003.
4. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR #98-06t, Department of Computer Science, Iowa State University, 2002.
5. MISRA Consortium. *MISRA-C: 2004 — Guidelines for the use of the C language in critical systems*, 2004.
6. S. of the SC-205/WG-71 Plenary. ED-215: Software Tool Qualification Considerations, 2010.
7. S. of the SC-205/WG-71 Plenary. ED-217: Object-Oriented Technology Supplement to ED-12C and ED-109A, 2010.
8. S. of the SC-205/WG-71 Plenary. ED-218: Formal Methods Supplement to ED-12C and ED-109A, 2010.
9. S. Plenary. DO-178C/ED-12C: Software considerations in airborne systems and equipment certification, 2010.
10. S. Plenary. DO-278A/ED-109A: Software standard for non-airborne systems, 2010.
11. J. M. Spivey. *The Z notation: a reference manual.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
12. S. Stepney, D. Cooper, and J. Woodcock. *An electronic purse: Specification, Refinement and Proof.* Technical Monograph PRG–126. Oxford University Computing Laboratory, Programming Research Group, 2000.

Modelling Adaptable Distributed Object Oriented Systems using the HATS Approach – A Fredhopper Case Study ⋆

Peter Y. H. Wong[1], Nikolay Diakov[1], and Ina Schaefer[2]

[1] Fredhopper B.V., Amsterdam, The Netherland
{peter.wong,nikolay.diakov}@fredhopper.com
[2] Technische Universität Braunschweig, Germany
i.schaefer@tu-bs.de

Abstract. The HATS project aims at developing a model-centric engineering methodology for the design, implementation and verification of distributed, concurrent and highly configurable systems. Such systems also have high demands on their dependability and trustworthiness. The HATS approach is centered around the Abstract Behavioural Specification modelling language (ABS) and its accompanying tools suite. The HATS approach allows precisely specifying and analyzing the abstract behaviour of distributed software systems and their variability. The HATS project measures its success by applying its framework not only to toy examples, but to real industrial scenarios. In this paper, we evaluate the HATS approach for modelling an industrial scale case study provided by the eCommerce company Fredhopper. In this case study we consider Fredhopper Access Server (FAS). We model the commonality and variability of FAS's replication system using the ABS language and provide an evaluation based on our experience.

Keywords: Variability modelling; Software product lines; Industrial case study; Formal modelling and specification; Evaluation

1 Introduction

Software systems evolve to meet changing requirements over time. Evolving software systems may require substantial changes to the software and often result in quality regressions. After a change in a software system, typically some work is needed in order to regain the trust of its users. The "Highly Adaptable and Trustworthy Software using Formal Models" (HATS) project aims at developing tools, techniques and a formal software product line (SPL) development methodology [10, 34] for rigorously engineering distributed software systems are subject to changes.

⋆ This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu).

The HATS approach is centered around the Abstract Behavioural Specification (ABS) modelling language [23, 13], an accompanying ABS tool suite [11, 36] and a formal engineering methodology for developing SPL [10]. ABS facilitates to model precisely SPLs of distributed concurrent systems, focusing on their functionality, while providing the abstraction to express concerns, such as available resources, deployment scenarios and scheduling policies. In particular, the language of ABS provides modelling concepts for specifying SPL's variability from the level of feature models down to object behaviour. This permits large scale reuse within SPLs and rapid product construction during the application engineering phase of the SPL engineering methodolody [10].

In this paper, we evaluate the application of the HATS approach to an industrial SPL case study of the Fredhopper Access Server (FAS) product line. FAS, developed by Fredhopper B.V. (`www.fredhopper.com`), is a distributed service-oriented software system for Internet search and merchandising. In particular we consider FAS's *replication system*; the replication system ensures data consistency across the FAS deployment. We use this case study to evaluate the HATS approach with respect to the following criteria, derived during the requirement elicitation activity conducted at the beginning of the HATS project [17]:

Expressiveness We evaluate the ABS language with respect to its *practical* language expressiveness. We investigate from the user's perspective how readily and concisely ABS allows users to express program structures and behavior, and its capability to capture variability in SPLs.

Scalability We evaluate the ABS language with respect to the size and the complexity of the modelled system. It is important to provide mechanisms at the language level that permit separations of concerns, reuse and compositional development of SPLs.

Usability We evaluate the HATS approach with respect to its overall usability, focussing on the ease of adoption and learnability. We take into account the tool support, as well as the language's syntax and semantics.

The structure of this paper is as follows: Section 2 briefly presents the HATS approach; Section 3 describes the functionality of the Fredhopper Access Server (FAS) product line and its replication system; Section 4 considers how to model the replication system's commonality using ABS; Section 5 considers how to model the replication system's variability using ABS. We present an evaluation based on our experience using ABS in Section 6. We provide an overview of the existing approaches to model and analyse concurrent distributed systems with variabilities in Section 7 and a summary of this paper in Section 8.

2 HATS approach

The HATS approach is designed as a formal methodology for developing SPL [10]. The HATS methodology is a combination of the ABS language, a set of well-defined techniques and tool suite for ABS, and a formal methodology to bind them to specific steps in a SPL development process. ABS comprises a core language with specialised language extensions, each focusing on a particular aspect of modelling SPLs, while respecting the separation of concerns principle and encouraging reuse.

The *Core ABS* is a strongly typed, concurrent, object-based modelling language with a formal executable semantics and a type system [23]. The Core ABS consists of a functional and a concurrent object levels: The functional level provides a flexible way to model internal data in concurrent objects, while separating the concerns of internal computation from the model; this is an important language feature for scalability. The functional level supports user-defined parametric data types and functions with pattern matching. The concurrent object level is used to capture concurrent control flow and communication in ABS models; the concurrency model of the Core ABS is based on the concept of Concurrent Object Groups. A typical ABS model consists of multiple object groups at runtime. These groups can be regarded as autonomous, runtime components that are executed concurrently, share no state and communicate asynchronously. The core ABS's object-based model structure provides a good fit with UML modelling approaches, while its type system guarantee type safety at runtime for well typed core ABS models. The core ABS's executable semantics supports early verification and validation.

The ABS then extends the Core ABS with the following specialised extensions [13].

- The *Micro Textual Variability Language* (μTVL), based on Classen et al.'s TVL [14], expresses the variability of SPL at the level of feature models during the family engineering phase of the SPL engineering process.
- The *Delta Modeling Language* (DML), based on delta modelling [33], models variability of SPL at the level of object behavior during the family engineering phase of the SPL engineering process. The variability at the behavioral level is represented by a set of delta modules that contain modifications of the ABS model of software artifacts in SPL, such as additions, modifications and removals of model entities. Delta modules provides the capability to define generic software artifacts in SPL.
- The *Product Line Configuration Language* (CL) connects the variability of SPL from feature models down to object behavior by specifying the relationship between features and delta modules. Each delta module is associated with one or more feature in the feature model, thereby allowing reuse delta modules across features in the SPL.
- The *Product Selection Language* (PSL) specifies individual products in the SPL by providing a particular feature selection along with its initialization code.

The ABS tools suite [36] includes an ABS compiler front end, which takes a complete ABS model of the SPL as input, checks the model for syntax and semantic errors and translates it into an internal representation. The front end supports automatic product generation, variability of the SPL can be resolved by applying the corresponding sequence of delta modules to its core ABS model at compile time; variability resolution is one of the core activities during the

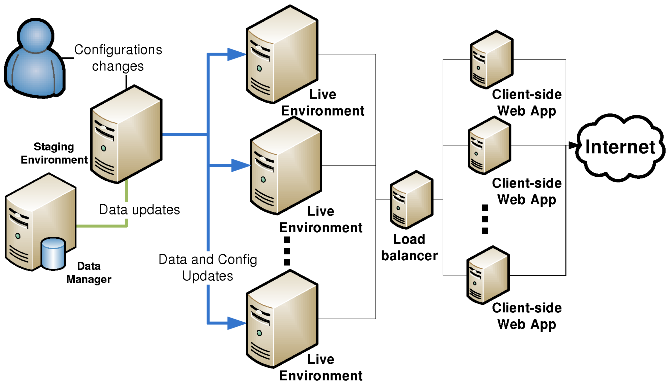


Fig. 1. An example of a FAS deployment

application engineering phase of the SPL. Different back ends translate the internal representation into Maude or Java, allowing ABS models to be executed and analyzed. The tools suite also includes a plug-in for the Eclipse IDE (`www.eclipse.org`). The plugin provides an Eclipse perspective for navigating, editing, visualizing, and type checking ABS models, and an integration with the back ends, so that ABS models can be executed or simulated directly from the IDE.

3 Fredhopper Access Server

The Fredhopper Access Server (FAS) is a component-based and service-oriented distributed software system. It provides search and merchandising services to e-Commerce companies such as large catalogue traders, travel agencies, etc. Each FAS installation is deployed to a customer according to the FAS deployment architecture. Figure 1 shows an example setup. A detailed presentation of FAS's individual components and its deployment model can be found in the HATS project report [18].

A FAS deployment consists of a set of live and staging environments. A live environment processes queries from client web applications via web services. FAS aims at providing a constant query capacity to client-side web applications. A staging environment is responsible for receiving data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments according to the replication protocol.

Implementations of the replication protocol are provided by the *replication system*. A replication system consists of a set of computation nodes; one of which is the synchronization server residing in a staging environment, while all other nodes are synchronization clients residing in the live environments. The synchronization server takes care of determining the schedule of replication, as well as the content of each replication item. The synchronization client is responsible for receiving data and configuration updates. A replication item is a set of files and represents a single unit of replicable data.

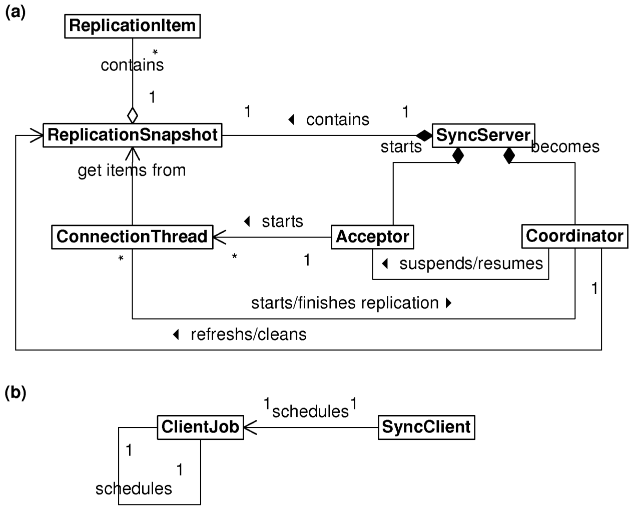


Fig. 2. Class diagram of (a) synchronisation server and (b) synchronisation client

The synchronization server communicate to clients via connection threads that serve as the interface to the server-side of the replication protocol. On the other hand, synchronization clients schedule client jobs to handle communications to the client-side of the replication protocol. In our ABS model, both connection threads and client jobs belong to separate concurrent object groups [23] (a mechanism to structure the object heap into separate units) and communicate via asynchronous method invocations. Cooperative multitasking and strict data encapsulation between the concurrent object groups prevent deadlocks and race conditions.

As part of the FAS product line, the replication system defines variability on the types of replication items, the coordination policy of replication and the resource consumption during replication. This allows members of the product line to be tailored for FAS deployments with specific data requirement and platform resource constraints.

4 Modelling Commonality

In this section, we present how to model the replication system's commonality using the Core ABS. Figures 2(a) and (b) show the UML class diagram of the synchronization server and client respectively. The synchronization server consists of an acceptor, several connection threads, a coordinator, a SyncServer and a replication snapshot. The synchronization client consists of a SyncClient and one or more client jobs. Listing 1.1 shows the ABS interfaces for the core components of the synchronization server and clients. For brevity, we have omitted ABS class definitions.

The Acceptor component is responsible for accepting connections from the synchronization clients and is specified by the interface Acceptor. The interface provides a method for a client job to obtain a reference to a connection thread, as well as methods to enable and disable the synchronization server to accept a new client job connection.

```
interface ConThread { Unit command(Command c); }
interface Acceptor {
  ConThread getConnection(Job job);
  Bool isAcceptingConnection();
  Unit suspendConnection();
  Unit resumingConnection(); }

interface Coordinator {
  Unit process();
  Unit startUpdate(ConThread worker);
  Unit finishUpdate(ConThread worker);}

interface SyncServer {
  Acceptor getAcceptor();
  Coordinator getCoordinator();
  Snapshot getSnapshot();
  DB getDataBase(); }

interface Snapshot {
  Unit refreshSnapshot(Bool r);
  Unit clearSnapshot();
  Set<Item> getItems();}

interface Item {
  FileEntry getContents();
  ItemType getType();
  Id getAbsoluteDir();
  Unit refresh();
  Unit cleanup();}

interface SyncClient {
  Acceptor getAcceptor();
  DB getDataBase();
  Unit becomesState(State state);
  Unit setAcceptor(Acceptor acceptor);}

interface Job {
  Bool registerItems(CheckPoint checkpoint);
  Maybe<FileSize> processFile(Id id);
  Unit processContent(File file);
  Unit receiveSchedule();}
```

Listing 1.1. ABS interfaces of the replication system

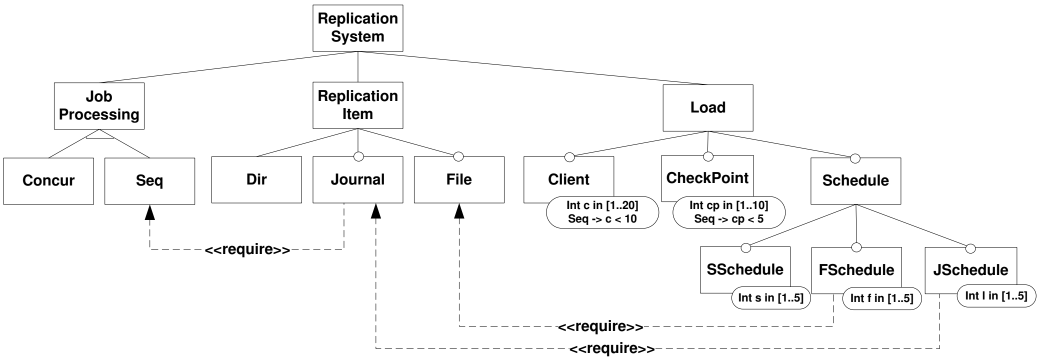


Fig. 3. Feature model of the Replication System

The connection thread and the client job are specified by interfaces ConThread and Job, respectively. Each connection thread is instantiated by the Acceptor component. After the Acceptor receives a connection from a client job, it instantiates a ConThread to carry out the replication protocol. A connection thread is specified by the interface ConThread. The ConThread component has a single method command(), which is asynchronously invoked by Job objects to determine the current state of a replication. The Coordinator component is responsible for coordinating the connections that the Acceptor accepts from synchronization clients. It also provides methods for preparing and clearing replication items before and after replication sessions. The SyncServer component starts the Acceptor and the Coordinator components. It also keeps a reference to the relevant replication snapshot, i.e., the data that is currently being replicated.

Listing 1.1 also shows the ABS interfaces of the components that are part of the synchronization clients. A SyncClient communicates with the SyncServer via job scheduling. At initialization time, the SyncClient schedules a client job to acquire a replication schedule from the server. Using this schedule, the client job creates a new client job for performing the actual replication. Each client job, thereafter, is responsible to request replication schedules and to set up the subsequent jobs for further replication. Each client job receives replication items from a connection thread and updates the synchronization client's files (configuration and data). The client job is specified by interface Job.

5 Modelling Variability

As part of FAS product line, the replication system defines variability on the types of replication item and replication strategy. We capture this variability using the ABS language extensions described in Section 2.

Figure 3 shows the feature diagram of the replication system and Listing 1.2 shows the corresponding μTVL model. Specifically, the replication system has three main features: JobProcessing, RepItem and Load.

The feature JobProcessing requires an alternative choice between the two sub features Seq and Concur, capturing the choice between sequential and concurrent client job processing, respectively.

The feature RepItem allows choosing between three replication item types represented by the features Dir, File and Journal. The Dir feature is mandatory, that is, all versions of the replication system support replicating complete file directories. The File feature is optional and is selected to support replicating a file set, whose files' name matches a particular pattern. the Journal feature is optional and is selected to support replicating database journal. In particular, the Journal feature requires the feature Seq which means that variants of the replication system that support database journal replication may only schedule client jobs sequentially.

```
root ReplicationSystem {
 group allof {
  JobProcessing { group oneof { opt Seq, opt Concur }},
  RepItem { group [1..*] { Dir, opt File, opt Journal { require: Seq; }}},
  opt Load {
   group [1..3] {
    Client { Int c in [1 .. 20]; Seq -> c < 10; },
    CheckPoint { Int cp in [1 .. 10]; Seq -> cp < 10; },
    Schedule {
     group [0..3] {
      SSchedule { Int s in [1 .. 5]; },
      FSchedule { Int f in [1 .. 5]; requires: File; },
      JSchedule { Int l in [1 .. 5]; requires: Journal; }
     }}}}}}
```

Listing 1.2. Feature model of the replication system in μTVL

The feature Load is an optional feature that configures the load of the replication system. It offers sub features Client, CheckPoint and Schedule. The feature Client configures the number of synchronisation clients, and defines the constraint such that if client job processing is sequential, the number of clients must be less than ten. The feature CheckPoint configures the number of updates allowed per execution, defines the constraint such that if the client job processing is sequential, the number of updates must be less than five. The feature Schedule configures the number of locations in the file system at which changes to different replication item types are monitored. It is an optional feature that offers sub-features SSchedule, FSchedule and JSchedule to record the number of locations for directory, file set, and journal replication respectively. Note that FSchedule and JSchedule cannot be selected unless features File and Journal are selected respectively.

The basis replication system supports sequential client job processing. This functionality is implemented by the active class JobImpl. A partial ABS class definition of JobImpl is shown in Listing 1.3. Each instance of the JobImpl class initialises the Boolean field newJob to False and invokes its run method. This method in turn invokes scheduleNewJob() asynchronously. The method scheduleNewJob() waits for field newJob to become True before creating a new instance of Job. Setting newJob to True at the end of the run method ensures that each client job is scheduled sequentially.

The lower half of Listing 1.3 defines the delta module Concurrent. A delta module specifies changes to a basis ABS model, such as the addition, modification or removal of classes, in order to define the shape of the model in another system variant. The delta module Concurrent specifies a class modifier for the class JobImpl that contains a method modifier. The method modifier removes the await statement from the method scheduleNewJob() such that a new instance of the class Job is created as soon as the current Job instance releases the lock of this object group. This allows scheduling client jobs concurrently.

```
class JobImpl(SyncClient c, JobType job) implements Job {
  Bool nj = False;
  Unit newJob() { await nj; new JobImpl(this.c,Replication); }
  Unit run() { .. this!newJob(); .. nj = True; .. }
  Unit state(State state) { .. } ..
}

delta Concurrent {
  modifies class JobImpl {
    modifies Unit newJob() { new JobImpl(this.c,Replication); }}}
```

Listing 1.3. Modeling job processing

```
class Dirs(Id q, DB db) implements Item { .. }

class SnapshotImpl(DB db, Schedules ss)
implements Snapshot {
  Set<Item> items = EmptySet;
  Unit item(Schedule s) {
    if (isSearchItem(s)) {
     Item item = new Dirs(left(item(s)),this.db);
     this.items = Insert(item,this.items);}..}}
```

Listing 1.4. Partial implementation of replication item

Listing 1.4 shows a partial definition of the classes DirectoryItem and SnapshotImpl. The class DirectoryItem defines a replication item for a complete file directory. The class SnapshotImpl represents a replication snapshot. The method item defined in the class SnapshotImpl takes a replication schedule, creates a corresponding Item object and adds it to the set of replication items. By default, this method only handles replication schedules for complete file directories.

In Listing 1.5, two delta modules are shown that contain the necessary functionality and modifications to handle other types of replication items. The delta module FileDelta is applied for file set replication and has three class modifiers. The first modifier adds the class FItem, implementing the interface Item, for handling replication file sets that match a regular expression. The second class modifier changes the class SnapshotImpl by updating the method item to handle replication schedules with file sets; here the statement **original**(s) calls the previous version of the method SnapshotImpl.item(s). The third class modifier changes the initial Main by introducing a new instance field files that records a list of schedules for file set replications. Similarly, the delta module JournalDelta contains the necessary modifications for handling database journal replication. It has three class modifiers to add a new implementation of interface Item, to update the method item to handle replication schedules with data base journals and to add new instance field logs that records a list of schedules for database journal replications.

```
delta FileDelta {
  adds class FItem(Id q, String p, DB db)
  implements Item { .. }

  modifies class SnapshotImpl {
    modifies Unit item(Schedule s) {
      original(s);
      if (isFileItem(s)) {
        Pair<Id,String> it = right(item(s));
        Item item = new FItem(fst(it),snd(it),this.db);
        items = Insert(item,items);
      }}}

  modifies class Main { adds List<Schedule> files = ... }
}

delta JournalDelta {
  adds class Journals(..) implements Item { .. }
  modifies class SnapshotImpl { .. modifies Unit item(..) ..}}
  modifies class Main { adds List<Schedule> logs = ... }
```

Listing 1.5. Deltas for replication items

Listing 1.6 shows the configuration of the replication system product line using the product line configuration language CL. The product line configuration links the modifications contained in the listed delta modules to product features and determines for which feature configurations the modifications have to be applied. In the considered example, the features Dir and Seq are the features provided by the core system. The application condition for delta module FileDelta states that this delta module is applied if feature File is selected, while the application condition for the delta module FSched states that the module is applied if feature FSchedule is selected and that it must be applied after delta module SSched should its corresponding feature be selected.

Listing 1.6 also shows two example product selections for the replication system product line specified in the product selection language PSL. Product P1 defines the basis variant of the replication system that supports the basic set of features, and product P2 that supports both directory and file set replication, concurrent client job scheduling and deploys three SyncClients for receiving replications.

```
productline ReplicationSystem {
  features Dir, File, Journal, Seq, Concur, Client, Schedule, CheckPoint,
           SSchedule, FSchedule, JSchedule;

  delta FileDelta when File;
  delta JournalDelta when Journal;
  delta Concurrent when Concur;
  delta CD(Client.c) when Client;
  delta CP(CheckPoint.cp) when CheckPoint;
  delta SSched(SSchedule.s) when SSchedule;
  delta FSched(FSchedule.f) after SSched when FSchedule;
  delta JSched(JSchedule.l) after FSched when JSchedule;
}

product P1 (Dir, Seq);
product P2 (Dir, File, Concur, Client({c=3}));
```

Listing 1.6. Product configuration and selection

6 Evaluation

This section presents an evaluation of the HATS approach with respect to the ABS language's expressiveness, scalability and usability. The specific criteria have been derived from the HATS project's requirement elicitation activity [17].

6.1 Expressiveness

We evaluate the practical expressiveness and the modeling capabilities of the ABS language. Specifically we investigate 1) how readily and concisely the ABS language expresses various kinds of program structures and behaviours, and 2) its capabilities to express variabilities behaviourally.

Data types Using ABS's algebraic data type, we were able to provide a high-level model of the replication system that abstracts from the underlying physical environment such as operating system, file storage and data base.

Functions Using the combination of ABS's algebraic data types and functions, we were able to use abstract data types such as lists, sets and maps, and subsequently define data types to abstract from the underlying environment such as file storage. We have also found functions to be useful as they guarantee to be free of side-effects and are more amenable to formal reasonsing. However, the ABS language does not support higher order functions. This means we cannot abstract certain behaviour, limiting reusability of function definition. This also implies that we cannot pass functions as parameters to methods.

Polymorphism ABS's algebraic data types and functions support parametric polymporhism, allowing data types and functions to be data-independently defined. However, this ABS classes and methods are not parametrically polymorphic, hence one has to specialise the types of method parameters, reducing the reusability of method implementations.

Syntactic Sugaring To model communications between active objects in ABS we often define the sequence of statements **Fut**<A> f = o!m(); **await** f?; A v = f.**get**; in an active object to model invoking method m() of object o asynchronously, yielding the thread control of its object group and blocking its own execution until the method call returns. We believe the usability of the language could be improved by providing syntactic sugaring to this kind of patterns of behaviours, and at the time of writing we know this types of sugaring are being added to the ABS language.

Concurrency We were able to model the replication system's concurrent behaviour in terms of asynchronous method invocation, this ABS model provides a high level view of the communication between synchronisation server and clients, thereby separating the concerns of the physical communication layers between them and hence reducing the complexity of the model considerably. Another advantage of the cooperative scheduling offered by ABS's concurrency model is that we can safely define a method that modifies a state of an object without the need to explicitly enforce mutual exclusion on that state. Nevertheless, due to inherent nondeterministic scheduling between COGs as well as active objects within a COG [23], it is not possible to enforce fairness over competing active objects when simulating an ABS model. At the time of writing an implementation of a real-time extension of the ABS language is being developed [24]. This would provide the mechanism to specify schedules on the asynchronous method invocations, and allow us to enforce orders of execution to avoid starvation.

Variability Delta Modelling Language offers the expressivity to specify variability at the level of object behaviour. Together with Product Line Configuration, Product Selection and μTVL Language, the ABS language offers a holistic approach to expressing variabilities as features and relating them to object behaviour. We were able to use ABS to incrementally and compositionally develop the replication system product line that yields members that are well-typed and valid with respect to the product line's variability. Nevertheless, the current implementation of DML does not support modification of functions and data types, and this means we cannot capture their variabilities in the same way as classes and interfaces. At the time of writing, we know the implementation of DML is being improved to support functions and data types.

6.2 Scalability

We evaluate the ABS language with respect to scalability and reusability.

Data types Using ABS's algebraic data types, we separate the concern of the replication system's physical environments such as operation system, file storage and data bases from its ABS model. This allows us to scale the replication system product line model, such as increasing the number of SynClients, without being constrained by the physical environment.

Modularity The module system allows us to model both the commonality and the variability of the replication system separately and incrementally. Specifically, we started modeling the commonality of the product line independent from the product line's variability and individual components of the replication system commonality are modeled in separate modules (and files). Moreover, we modeled the product line's variability in terms of delta modules, this allows variation to be modeled incrementally while dividing delta modules in terms of the components which variations are to be resolved.

Code reusability DML provides the mechanism to express variability at the level of behaviour. This together with functional and object composition, the ABS language provides a wide range of mechanism for code reuse. In particular the combination of object composition and delta modelling allows us to achieve code reusability similar to that of class inheritance. In addition, the ABS module system also allows more generic definitions such as data types and functions to be reused across the ABS model of the product line. Nevertheless, while the current implementation of DML supports **original**() in method modifiers, which invokes that method's previous implementation, it does not support **original**() of a specific implementation, this has reduced code reusability when resolving conflict [12]. At the time of writing, DML is being improved to support delta-specific **original**().

Timing and resource information The current ABS semantics does not take time and the environment's resources into consideration. This separation of concerns allows one to focus on functional and partial correctness. Nevertheless, at the time of writing, an implementation of a real-time extension of the ABS language is being developed [24]. This would allow ABS models to express behavioural constraints due to timing information as well as the resources of the environment. This would enable one to analyse an ABS model with specific environment constraints such as process speed, memory etc.

6.3 Usability

In this section we evaluate ABS with respect to its overall usability, focusing on the ease of adoption and learnability, and taking into account the ABS tool suite as well as the language's syntax and semantics.

Note that the case study has been conducted in tandem with the development of the ABS language and tool suite, the case study, which has been conducted over the span of 14 months, has consequently led to many enhancement and fixes. Specifically, the HATS project employs an open source ticket tracking system (`http://trac.edgewall.org/`) to track bugs and feature requests, and the case study has brought about ten enhancements and over sixty fixes. As the ABS tool suite has been at development stage during the case study, our evaluation of usability would take this into account.

Syntax and semantics ABS has been designed to be as easy to learn as possible by building on language constructs well known from mainstream programming languages. Both functional and sequential imperative fragments of ABS can be easily acquainted by users with a working knowledge of any functional and object-oriented languages. However, it seems not as easy at first to learn the concurrent fragment of ABS, especially for those who are used to the multithreaded concurrency model. We believe this issue is remedied in twofold: 1) the availability of literature such as the technical papers [23, 13], the tutorial chapter [11] and case studies [18], and 2) the support of the ABS tool suite [36].

Compiler front end The ABS tool suite comes with a front end that takes an ABS model, performs parsing and type checking, and outputs the model's Abstract syntax tree (AST). The design of the ABS language and the availability of the front end guarantees that the ABS model constructed in the case study is well-typed. Moreover, the front end allows product derivation based on the the ABS model's product line configuration and product selection.

Maude/Java back ends The current version of the ABS tool suite comes with a back end for Maude and Java: The Maude back end takes the type checked AST of an ABS model and outputs the corresponding Maude model that can be then simulated using the Maude engine (`maude.cs.uiuc.edu`). Using both the front end and the Maude back end, we were able to quickly simulate multiple versions of the replication system. The ABS tool suite also provides a Java back end that takes the type checked AST of an ABS model and outputs the corresponding Java source codes that can be compiled and executed independently. We have found the Java back end to be particularly useful when used in conjunction with the ABS debugger.

Eclipse plugin – The ABS Eclipse plugin provides syntax highlighting, content completion and code navigation similar to those provided by the Eclipse JDT (`www.eclipse.org/jdt`) for Java. The plugin also integrates the front end and back ends as a singe source technology such that construction, compilation and simulation of ABS models can be carried out directly via the Eclipse IDE. We have found the availability of the IDE greatly increases the scalability of the HATS approach during modeling. The ABS Eclipse plugin can be installed as a bundle via its Eclipse update site `tools.hats-project.eu/update-site`. A recent version of the ABS Eclipse plugin provides the capabilities to import and navigate ABS packages; ABS packages are JAR files containing ABS source codes. We believe this feature increases ABS's applicability in the industry where collaborative software development is prevalent and third party libraries are heavily used.

Debugger The ABS Eclipse plugin offers a debugging perspective for debugging ABS models. The novelty of this perspective is that users can define explicitly the order in which asynchronous method invocations are executed within a concurrent object group. The debugger also offers the option to save and replay histories of asynchronous method invocations. These facilities greatly ease our task of debugging and reproducing bugs during the case study.

Visualization Through the ABS Eclipse plugin's debugging perspective, the ABS tool suite offers a visualization tool that generates UML sequence diagrams of asynchronous communications between concurrent object groups during the debugging session. Sequence diagrams provide high-level views of the communications between components in the replication system, and this increases our understanding of the system's concurrent behaviour.

7 Related work

In this section we consider related work in the context of abstract behavioural, variability modelling, and evaluating SPL engineering methodologies.

The ABS language is a modelling language that aims to close the gap between design-level notations and implementation languages. The concurrent object model of ABS based on asynchronous communication and a separation of concern between communication and synchronization is part of a trend in programming languages today, due to the increasing focus on distributed systems. For example, the recent programming language Go (`http://golang.org`, promoted by Google) shares in its design some similarities with ABS: a nominal type system, interfaces (but no inheritance), concurrency with message passing and non-blocking receive. The internal concurrency model of concurrent objects in ABS stems from the intra-object cooperative scheduling introduced in Creol [16] This model allows active and reactive behavior to be combined within objects as well as compositional verification of partial correctness properties [1].

Existing approaches to express variability in modelling and implementation languages can be classified into two main categories [35, 25]: annotative and compositional. As a third approach, model transformations are applied for representing variability mainly in modelling languages.

Annotative approaches consider one model representing all products of the product line. Variant annotations, e.g., using UML stereotypes in UML models [19] or presence conditions [15], define which parts of the model have to be removed to derive a concrete product model. The orthogonal variability model (OVM) proposed in Pohl et al. [31] models the variability of product line artifacts in a separate model where links to the artifact model take the place of annotations. Similarly, decision maps in KobrA [7] define which parts of the product artifacts have to be modified for certain products.

Compositional approaches, such as delta modelling [33], associate model fragments with product features that are composed for a particular feature configuration. A prominent example of this approach is AHEAD [9], which can be applied on the design as well as on the implementation level. In AHEAD, a product is built by stepwise refinement of a base module with a sequence of feature modules. Design-level models can also be constructed using aspect-oriented composition techniques [21, 35, 30]. Apel et al. [2] apply model superposition to compose model fragments.

In feature-oriented software development (FOSD) [9], features are considered on the linguistic level by feature modules. Apart from Jak [9], there are various other languages using the feature-oriented paradigm, such as FeatureC++ [3], FeatureFST [4], or Prehofer's feature-oriented Java extension [32]. In [29, 4], combinations of feature modules and aspects are considered. In [5], an algebraic representation of FOSD is presented. Feature Alloy [6] instantiates feature-oriented concepts for the formal specification language Alloy.

Model transformations are used to represent product variability mainly on the artifact modelling level. The common variability language (CVF) [20] represents the variability of a base model by rules describing how modelling elements of the base model have to be substituted in order to obtain a particular product model. In [22], graph transformation rules capture artifact variability of a single kernel model comprising the commonalities of all systems.

There have been interests to evaluate SPL methodologies using case studies [26, 27]. In Lopez-Herrejon et al. work [26], for example, they propose the Graph Product Line (GPL) as a standard problem to implement for evaluating SPL methodologies. In this work they compare qualities such as performance and lines of code between implementations of GPL using the GenVoc SPL methodology [8]. There are also evaluation strategies that focus on other concerns such as tool support. For example, Matinlassi [28] compares several SPL methodologies with respect to qualities such as tool support, guidance and application domains.

8 Summary

In this paper, we presented an evaluation on the HATS approach by conducting a case study on an industrial-scale software product line of a distributed and highly configurable software system using the ABS language and its accompanying ABS tools suite. We modelled the replication system's commonality using Core ABS and the replication system's variability using the Full ABS. At the time of writing the replication system product line ABS model consists of 5000 lines of code, and defines 40 classes, 43 interfaces, 15 features, 8 deltas and 288 products. Based on the case study we provided an evaluation of the ABS language with respect to practical expressiveness and modeling capabilities, scalability and usability.

By performing the case study in tandem the development of the HATS approach, we were able to provid timely feedback on language expressiveness and the applicability of the modeling tools. The work on the case study has influenced decisions in the design of the ABS language, and in both the enhancements and fixes to the tools suite. As the HATS approach continues to mature, we aim to extend the replication system case study to capture further variabilities and evolution scenarios, and to conduct formal validation and verification of the replication system using the formal analysis tools developed for ABS models.

References

1. W. Ahrendt and M. Dylla. A system for compositional verification of asynchronous objects. *Science of Computer Programming*, 2011. In Press. To appear.

2. S. Apel, F. Janda, S. Trujillo, and C. Kästner. Model Superimposition in Software Product Lines. In *International Conference on Model Transformation (ICMT)*, 2009.
3. S. Apel, T. Leich, M. Rosenmüller, and G. Saake. Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In *GPCE*, volume 3676 of *Lecture Notes in Computer Science*, pages 125–140. Springer-Verlag, 2005.
4. S. Apel and C. Lengauer. Superimposition: A language-independent approach to software composition. In *Software Composition*, volume 4954 of *Lecture Notes in Computer Science*, pages 20–35. Springer-Verlag, 2008.
5. S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebraic foundation for automatic feature-based program synthesis. *Science of Computer Programming*, 75(11):1022 – 1047, 2010.
6. S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Detecting Dependences and Interactions in Feature-Oriented Design. In *IEEE International Symposium on Software Reliability Engineering*, 2010.
7. C. Atkinson, J. Bayer, , and D. Muthig. Component-Based Product Line Development: The KobrA Approach. In *SPLC*, 2000.
8. D. Batory and B. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering*, Feb. 1997.
9. D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.*, 30(6), 2004.
10. D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, G. Puebla, B. Weitzel, and P. Y. H. Wong. HATS: A Formal Software Product Line Engineering Methodology. In *Proceedings of International Workshop on Formal Methods in Software Product Line Engineering*, Sept. 2010.
11. D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, and P. Y. H. Wong. Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *LNCS*, pages 417–457, June 2011.
12. D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract Delta Modeling. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 13–22, New York, NY, USA, Oct. 2010. ACM.
13. D. Clarke, R. Muschevici, J. Proença, I. Schaefer, and R. Schlatte. Variability modelling in the ABS language. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, Lecture Notes in Computer Science. Springer-Verlag, 2011. To appear.
14. A. Classen, Q. Boucher, and P. Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming*, Nov. 2010.
15. K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.
16. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, 2007.
17. Requirement Elicitation, Aug. 2009. Deliverable 5.1 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.
18. Evaluation of Core Framework, Aug. 2010. Deliverable 5.2 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

19. H. Gomaa. *Designing Software Product Lines with UML*. Addison Wesley, 2004.
20. Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. Olsen, and A. Svendsen. Adding Standardized Variability to Domain Specific Languages. In *SPLC*, 2008.
21. F. Heidenreich and C. Wende. Bridging the Gap Between Features and Models. In *Aspect-Oriented Product Line Engineering (AOPLE'07)*, 2007.
22. P. K. Jayaraman, J. Whittle, A. M. Elkhodary, and H. Gomaa. Model composition in product lines and feature interaction detection using critical pair analysis. In *MoDELS*, pages 151–165, 2007.
23. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, Lecture Notes in Computer Science. Springer-Verlag, 2011. To appear.
24. E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Validating timed models of deployment components with parametric concurrency. In B. Beckert and C. Marché, editors, *Proc. International Conference on Formal Verification of Object-Oriented Software (FoVeOOS'10)*, volume 6528 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2011.
25. C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE*, pages 311–320, 2008.
26. R. E. Lopez-Herrejon and D. S. Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *International Conference on Generative and Component-Based Software Engineering (GCSE'01)*, volume 2186 of *LNCS*, pages 10–24, 2001.
27. R. E. Lopez-Herrejon, D. S. Batory, and W. R. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *European Conference on Object-Oriented Programming (ECOOP'05)*, volume 3586 of *Lecture Notes in Computer Science*, pages 169–194. Springer-Verlag, 2005.
28. M. Matinlassi. Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA. In *International Conference on Software Engineering (ICSE'04)*, pages 127–136. IEEE Computer Society, 2004.
29. M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *SIGSOFT FSE*, pages 127–136. ACM, 2004.
30. N. Noda and T. Kishi. Aspect-Oriented Modeling for Variability Management. In *SPLC*, 2008.
31. K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, Heidelberg, 2005.
32. C. Prehofer. Feature-oriented programming: A fresh look at objects. In *European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer-Verlag, 1997.
33. I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Proc. of 15th Software Product Line Conference (SPLC 2010)*, Sept. 2010.
34. I. Schaefer and R. Hähnle. Formal methods in software product line engineering. *IEEE Computer*, 44(2):82–85, Feb. 2011.
35. M. Völter and I. Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *SPLC*, pages 233–242, 2007.
36. P. Y. H. Wong, E. Albert, R. Muschevici, J. Proença, J. Schäfer, and R. Schlatte. The ABS Tool Suite: Modelling, Executing and Analysing Distributed Adaptable Object-Oriented Systems, Sept. 2011. Submitted for publication.

Provably Correct Control-Flow Graphs from Java Programs with Exceptions

Afshin Amighi[1], Pedro de Carvalho Gomes[2], and Marieke Huisman[1]

[1] University of Twente, Enschede, The Netherlands
{a.amighi,m.huisman}@utwente.nl
[2] KTH, Royal Institute of Technology, Stockholm, Sweden
pedrodcg@csc.kth.se

Abstract. We present an algorithm to extract flow graphs from Java bytecode, focusing on exceptional control flows. We prove its correctness, meaning that the behaviour of the extracted control-flow graph is an over-approximation of the behaviour of the original program. Thus any safety property that holds for the extracted control-flow graph also holds for the original program. This makes control-flow graphs suitable for performing different static analyses.

For precision and efficiency, the extraction is performed in two phases. In the first phase the program is transformed into a BIR program, where BIR is a stack-less intermediate representation of Java bytecode; in the second phase the control-flow graph is extracted from the BIR representation. To prove the correctness of the two-phase extraction, we also define a direct extraction algorithm, whose correctness can be proven immediately. Then we show that the behaviour of the control-flow graph extracted via the intermediate representation is an over-approximation of the behaviour of the directly extracted graphs, and thus of the original program.

Keywords: Software Verification, Static Analysis, Program Models

1 Introduction

Over the last decade software has become omnipresent, and in parallel, the demand for software quality and reliability has been steadily increasing. Different formal techniques are used to reach this goal, e.g., static analysis, model checking and (automated) theorem proving. A major remaining problem is that the state space of software is enormous (and often even infinite). Thus, appropriate abstractions make the formal analysis tractable. It is important for such abstractions that they are sound w.r.t. the original program: if a property holds over the abstract model, it should also be a property of the original program.

A common abstraction is to extract a program model from code, only preserving information that is relevant for the property at hand. In particular, control-flow graphs (CFGs) [4] are a widely used abstraction, where only the flow information is kept, and all program data is abstracted away. Concretely, in a control-flow graph, the nodes represent the control points of a method, and the edges represent the instructions that make the transitions between control points. Usually, CFG is not a suitable abstraction for verifications that need data.

The literature contains several approaches to extract control-flow graphs automatically from program code. However, typically no formal argument is given why the extraction is property-preserving. This paper fills this gap: it defines a flow graph extraction algorithm for Java bytecode (JBC) *and* it proves that the extraction algorithm is sound w.r.t program behaviour. The extraction algorithm considers all the typical intricacies of Java, e.g., virtual method call resolution, the differences between dynamic and static object types, and exception handling.

The analysis of exceptional flows is a major complication to extract control-flow graphs of Java bytecode for two distinct reasons. First, the stack-based nature of the Java Virtual Machine (JVM) makes it hard to determine the type of explicitly thrown exceptions, thus making it difficult to determine to which (exceptional) control point the program will flow. Second, the JVM can raise (implicit) run-time exceptions, such as *NullPointerException* and *IndexOutOfBoundsException*; to keep track of where such exceptions can be raised requires much information. This paper covers the explicitly thrown instructions, and a significative subset of run-time exceptions.

Two different extraction algorithms are presented. The first extraction algorithm (in Section 3) creates flow graphs directly from Java bytecode. Its correctness proof is quite direct, but the resulting control-flow graph is large: in bytecode, all operands are on the stack, thus many instructions for stack manipulation are necessary, which all give rise to an *internal transfer* edge in the control-flow graph. Moreover, because the operands of a `throw` instruction are also on the stack, the exceptional control-flow is significantly over-approximated. This algorithm produces a complete map from the JBC instructions to the control-flow of the program, however, it is not so efficient for control-flow safety verifiers (e.g. to verify whether a sequence of method calls is correct).

As an alternative, we also present a two-phase extraction algorithm using the bytecode Intermediate Representation (BIR) language [5]. BIR is a stackless representation of JBC. Thus all instructions (including the explicit `throw`) are directly connected with their operands and this simplifies the analysis of explicitly thrown exceptions. Moreover the representation of a program in BIR is smaller, because operations are not stack-based, but represented as expression trees. As a result, the CFGs are efficient. BIR has been developed by Demange *et al.* as a module of SAWJA [8], a library for static analysis of Java bytecode. Demange *et al.* have proven that their translation from bytecode to BIR is semantics-preserving with respect to observable events, such as throwing exceptions and sequences of method invocations. Advantages of using the transformation into BIR are that (1) it is proven correct, and (2) it generates special assertions that indicate whether the next instruction could potentially throw a run-time exception. Our indirect extraction algorithm first generates BIR from

JBC (using the transformation of Demange *et al.*), and then our own algorithm to extract control-flow graphs from BIR.

There is no behavior definition for BIR. Therefore, to prove the correctness of the indirect extraction, we connect the BIR CFGs to the CFGs produced by the direct algorithm. We show that every BIR CFG structurally simulates the JBC CFG. This allows us to exploit an existing result that structural simulation induces behavioural simulation. Further, we prove that the CFG produced by the direct algorithm behaviourally simulates the original Java bytecode program, and from this we can conclude that the behaviours of the CFG generated by the indirect algorithm (BIR) also are a sound over-approximation of the original program behaviour. Thus, the control-flow graph extraction algorithm is sound.

Organization The remainder of this paper is organized as follows. First, Section 2 provides the necessary background definitions for the algorithm and its correctness proof. Then, Section 3 discusses the direct extraction rules for control flow graphs from Java bytecode, while Section 4 discusses the indirect extraction rules via BIR and proves its correctness. Finally Sections 5 and 6 present related work and conclude.

2 Preliminaries

2.1 Java bytecode and the Java Virtual Machine

The Java compiler translates a Java source code program into a sequence of bytecode instructions. Each instruction consists of an operation code, possibly using operands on the stack. The Java Virtual Machine (JVM) is a stack-based interpreter that executes such a Java bytecode program.

Any execution error of a Java program is reported by the JVM as an exception. Exceptions also can be thrown explicitly by instruction `athrow`. Each method can define exception handlers. If no appropriate handler can be found in the currently executing method, its execution is completed abruptly and the JVM continues looking for an appropriate handler in the caller context. This process continues until a correct handler is found or no calling context is available anymore. In the latter case, the execution terminates exceptionally.

Freund and Mitchell propose a formal framework for Java bytecode [6]. A JBC program is modeled as an environment $\mathit{\Gamma}$, which is a partial map from class names, interface names and method signatures to their respective definitions. Subtyping in an environment is indicated by $\mathit{\Gamma} \vdash \tau_1 <: \tau_2$, meaning τ_1 is a subtype of τ_2 in environment $\mathit{\Gamma}$. Let $\textsc{Meth}$ be a set of method signatures. A method $m \in \textsc{Meth}$ in an environment $\mathit{\Gamma}$ is represented as $\mathit{\Gamma}[m] = \langle P, H \rangle$, where P denotes the body and H the exception handler table of method m. Let $\textsc{Addr}$ be the set of all valid instruction addresses in $\mathit{\Gamma}$. Then $\mathit{Dom}(P) \subset \textsc{Addr}$ is the set of valid program addresses for method m and $P[k]$ denotes the instruction at position $k \in \mathit{Dom}(P)$ in the method's body. For convenience, $m[k] = i$ denotes instruction $i \in \mathit{Dom}(P)$ at location k of method m.

In this formal model, a JVM execution state is a configuration $C = A;h$, where A denotes the sequence of activation records and h is the heap. Each activation record is created by a method invocation. Formally the sequence is defined as follows:

$$A \ ::= \ A' \mid \langle x \rangle_{exc}.A' \quad ; \quad A' \ ::= \ \langle m, pc, f, s, z \rangle.A' \mid \epsilon$$

Here, m is the method signature of the active method, pc is the program counter, f is a map from local variables to values, s is the operand stack, and z is initialization information for the object being initialized in a constructor. Finally, $\langle x \rangle_{exc}$ is an exception handling record, where $x \in \textsc{Excp}$ denotes the exception: in case of an exception, the JVM pushes such a record on the stack.

To handle exceptions, the JVM searches the exception table declared in the current method to find a corresponding set of instructions. The method's exception table H is a partial map which has the form $\langle b, e, t, \varrho \rangle$, where $b, e, t \in \textsc{Addr}$ and $\varrho \in \textsc{Excp}$. If an exception of subtype ϱ in environment Γ is thrown by an instruction with index $i \in [b, e)$ then $m[t]$ will be the first instruction of the corresponding handler. Thus, the instructions between b and e model the `try` block, while the instructions starting at t model the `catch` block that handles the exception. In order to manage `finally` blocks, a special type of exception called *Any* is defined. The instructions in a finally block always have to be executed by the JVM, therefore all exceptions are defined as a subtype of *Any*.

2.2 Program Model

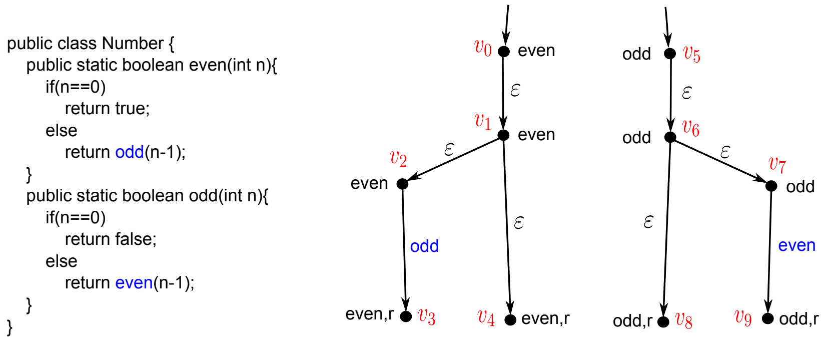
Control-flow graphs present an abstract model of a program. To define the structure and behavior of a control-flow graph we follow Gurov et al. and use the general notion of *model* [7, 9].

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

Method specifications are the basic building blocks of flow graphs. To model sequential programs with procedures and exceptions, method specifications are defined as an instantiation of initialized models as follows.

Definition 2 (Method Specification). *A* flow graph with exceptions *for $m \in$ $\textsc{Meth}$ over sets $M \subseteq \textsc{Meth}$ and $E \subseteq \textsc{Excp}$ is a finite model $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$ with V_m the set of control nodes of m, L_m the set of the labels which can be any instantiation to indicate the labels, $A_m = \{m, r\} \cup E$, $m \in \lambda_m(v)$ for all $v \in V_m$, and for all $x, x' \in E$, if $\{x, x'\} \subseteq \lambda_m(v)$ then $x = x'$, i.e., each control node is tagged with the method signature it belongs to and at most one exception. $\mathbb{E}_m \subseteq V_M$ is a non-empty set of entry control point(s) of m.*

A node $v \in V_m$ is marked with atomic proposition r to indicate that it is a return node of the method. The labeling set L_m is not specified intentionally to accept any instantiation. For example, figure 1 shows a sample program with corresponding CFG in which on the contrary to internal transitions, method calls are important. So L_m is instantiated as $L_m = M \cup \{\varepsilon\}$.



```
public 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. Method specifications of methods `even` and `odd`

Every flow graph comes with an interface that defines which methods are provided to and required from the environment.

Definition 3 (Flow Graph Interface). *A* flow graph interface *is a triple $I = (I^+, I^-, E, M_e)$, where $I^+, I^- \subseteq$ METH are finite sets of provided and required method signatures, and $E \subseteq$ EXCP is a finite set of exceptions and $M_e \subseteq$ METH is the set of entry methods (starting points of the program), respectively. If $I^- \subseteq I^+$ then I is* closed.

A flow graph of a program is the disjoint union of the flow graphs of all the methods defined in the program.

Definition 4 (Flow Graph Structure). *Flow graph $\mathcal{G}$ with interface I, written $\mathcal{G} : I$ is inductively defined by:*

- *$(\mathcal{M}_m, \mathbb{E}_m) : (\{m\}, \mathcal{M}, E)$ if $(\mathcal{M}_m, \mathbb{E}_m)$ is a method specification for m over M and E,*
- *$\mathcal{G}_1 \uplus \mathcal{G}_2 : I_1 \cup I_2$ if $\mathcal{G}_1 : I_1$ and $\mathcal{G}_2 : I_2$.*

Definition 5 (Weak Simulation over Models). *Let $a \Longrightarrow b$ be a sequence of silent transitions (possible zero) from a to b, and $a \overset{\beta}{\Longrightarrow} b$ a sequence containing a single visible transition β, and zero to many silent transitions. A weak simulation over a model $(S, L, \rightarrow, A, \lambda)$ is a binary relation R_w over S ($R_w \subseteq S \times S$) such that $\forall p, q \in S$, if $(p, q) \in R_w$ then $\lambda(p) = \lambda(q)$ and $\forall \beta \in L, \forall p' \in S$, if $p \overset{\beta}{\Longrightarrow} p'$ implies that there is a $q' \in S$ such that $q \overset{\beta}{\Longrightarrow} q'$ and $(p', q') \in R_w$.*

We use the following proposition to prove the weak simulation relation. The proof is trivial and we omit it.

Proposition 1. *R_w is a weak simulation if and only if $\forall(p,q) \in R_w$*
if $p \to p'$ then exists $q' \in S$ such that $q \implies q' \wedge (p',q') \in R_w$.
if $p \overset{\beta}{\to} p'$ then exists $q' \in S$ such that $q \overset{\beta}{\implies} q' \wedge (p',q') \in R_w$.

3 Extracting Control-Flow graphs from bytecode

CFG construction rules use bytecode instructions to build the graph. Depending on the instruction at a given address, edges between the current control node and the possible next control nodes are constructed.

For convenience, we group the different JBC instructions into disjoint sets: $\textsc{RetInst}$ is the set of normal return instructions (e.g. `return`); $\textsc{CmpInst}$ is the set of simple computational instructions (e.g. `push v, pop`); $\textsc{CndInst}$ is the set of conditional instructions (e.g. `ifeq q`); $\textsc{JmpInst}$ is the set of jump instructions (e.g. `goto q`); $\textsc{XmpInst}$ is the set of instructions that potentially can raise an exception (e.g. `div, getfield f`); $\textsc{InvInst}$ is the set of method invocation instructions (e.g. `invokevirtual (o,m)`); and $\textsc{ThrInst} = \{\texttt{throw}\ X\}$, where instruction `throw X` is the result of a stack analysis of the JBC. In JBC, `athrow` does not accept any argument and the type of the exception is determined at run-time (as the top of the stack). Stack analysis of the JBC can generate an exception variable to be thrown at run-time. We over-approximate the type of the exception using a set X that contains the static type of the variable, which is the result of the stack analysis and all its subtypes.

We define a JBC method body as a sequence of address and instruction pairs:

$$S ::= \quad \ell : inst\ ;\ S \mid \epsilon \qquad \ell \in \textsc{Addr},\ inst \in \textsc{Inst}$$

The nodes in a method CFG, define a map of the method's execution state, covering all possible JVM configurations. All nodes are tagged with pairs of an address and a method signature. The set of the addresses is extended by adding symbol $\flat$ to denote the *abort* state[3] of a program. Based on Definition 2, to construct the nodes we have to specify V_m, A_m and λ_m. For a node $v \in V_m$ indicating control point $\ell \in \textsc{Addr}_\flat$ of method m, we define $v = (m, \ell)$. The labeling function λ_m specifies A_m for a given $v \in V_m$. If $m[\ell] \in \textsc{RetInst}$ then the node is tagged with r. If the node is an exceptional node (an exception is raised) then it is marked with the exception type $x \in E$. The corresponding method signature is the default tag for all the method's control nodes. If $\ell = 0$ then the node will be a member of $\mathbb{E}_m$.

Two nodes are equal if they specify the same control address of the same method with equal atomic proposition sets. We use the following notation: $v \vDash x$

[3] The JVM's attempt to find a proper handler for an exception is unsuccessful and the program terminates abnormally.

$$
\begin{array}{l}
m\mathcal{G}(S_1;S_2,H) = m\mathcal{G}(S_1,H)\ \cup\ m\mathcal{G}(S_2,H)\\
m\mathcal{G}((p,i),H) = \{(\circ_m^p, i_g, \circ_m^{succ\ p})\} \qquad \texttt{if}\ \ i \in \textsc{CmpInst}\\
m\mathcal{G}((p,i),H) = \{(\circ_m^p, i_g, \circ_m^q)\} \qquad \texttt{if}\ \ i \in \textsc{JmpInst}\\
m\mathcal{G}((p,i),H) = \{(\circ_m^p, i_g, \circ_m^{succ\ p}), (\circ_m^p, i_g, \circ_m^q)\} \qquad \texttt{if}\ \ i \in \textsc{CndInst}\\
m\mathcal{G}((p,i),H) = \big\{\{(\circ_m^p, i_g, \bullet_m^{p,x})\}\ \cup\ \mathcal{H}_p^x \mid x \in X\big\} \qquad \texttt{if}\ \ i = \texttt{throw}\ X\\
m\mathcal{G}((p,i),H) = \{(\circ_m^p, i_g, \circ_m^{succ\ p})\}\ \cup\ \mathcal{E}_p^i \qquad \texttt{if}\ \ i \in \textsc{XmpInst}\\
\mathcal{E}_p^i = \big\{\{(\circ_m^p, i_g, \bullet_m^{p,x})\}\ \cup\ \mathcal{H}_p^x \mid x \in \mathcal{X}(i)\big\}\\
m\mathcal{G}((p,i),H) = \{(\circ_m^p, i_g, \bullet_m^{p,\varrho_N})\}\ \cup\ \mathcal{R}_p^i \cup\ \mathcal{H}_p^{\varrho_N} \cup\ \mathcal{N}_p^x \qquad \texttt{if}\ \ i \in \textsc{InvInst}\\
\mathcal{R}_p^i = \{(\circ_m^p, call\ (\tau,n), \circ_m^{succ\ p}) \mid\ \tau \in Rec_\Gamma(i)\}\\
\mathcal{N}_p^x = \big\{\{(\circ_m^p, handle_n, \bullet_m^{p,x})\}\ \cup\ \mathcal{H}_p^x \mid\ \bullet_n^{q,x,r} \in\ m\mathcal{G}(n),\ n \in\ res_\Gamma^\alpha(o,n)\ \big\}\\
(\ \Gamma \vdash x <: y\ ) \quad \Longrightarrow\ \mathcal{H}_p^x = \begin{cases} \{(\bullet_m^{p,x}, handle, \circ_m^t)\} & \hbar_{\Gamma[m]}(p,y) = t \neq 0\\ \{(\bullet_m^{p,x}, handle, \bullet_m^{p,x,r})\} & \hbar_{\Gamma[m]}(p,y) = 0\ \wedge\ m \notin M_e\\ \{(\bullet_m^{p,x}, handle, \bullet_m^{\flat,x,r})\} & \hbar_{\Gamma[m]}(p,y) = 0\ \wedge\ m \in M_e \end{cases}
\end{array}
$$

Fig. 2. CFG Construction Rules

means that node v is tagged with exception x; $\bullet_m^{\ell,x}$ indicates an exceptional control node and $\circ_m^\ell$ denotes a normal control node.

The CFG extraction rules for method m in environment Γ use the implementation of the method, $\Gamma[m] = \langle P, H\rangle$. For each instruction in $\Gamma[m]$, the rules build a set of labeled edges connecting control nodes.

Definition 6 (Method Control-Flow Graph Extraction). *Let V be the set of nodes and $I_g = \textsc{Inst}_g \cup \{handle\}$, where $\textsc{Inst}_g$ is any mapping from $\textsc{Inst}$ to the corresponding instruction. Let Π be a set of environments. Then the control-flow graph extraction of method m is $m\mathcal{G} : \Pi \times \textsc{Meth} \to \mathcal{P}(V \times I_g \times V)$, defined in Figure 2 (where succ denotes the next instruction address function).*

The construction rules are defined purely syntactically, based on the method's instructions. However, intuitively they justify the instruction's operational semantics. The first rule decomposes a sequence of instructions into individual instructions. For each individual instruction, a set of edges is computed.

For simple computational instructions, a direct edge to the next control address is produced. For jump instructions, an edge to the jump address (q, specified in the instruction) is generated. For conditional instructions two edges are generated: to the next control address and to the address specified for the jump (q). For instructions in $\textsc{XmpInst}$ edges for all possible flows are added: successful execution, and exceptional execution, with edges for successful and failed exception handling, as defined by function $\mathcal{H}_p^x$. This function constructs the outgoing edges of the exceptional nodes by searching the exception table for a suitable handler of exception type x at position p. If there is such a handler, it returns a transition edge from an exceptional node to a normal node. Otherwise it produces a transition to an exceptional return node. Function $\hbar$ seeks the proper handler in the exception handling table; it returns 0 if there is no entry for the exception at the specified control point. The function $\mathcal{X} : \textsc{XmpInst} \to \mathcal{P}(\textsc{Excp})$

is to determine possible exceptions of a given instruction. The `throw` instruction is handled similarly, where X is the set of possible exceptions, identified by the transformation algorithm.

To extract edges for method invocations, function $Rec_{\Gamma}(i)$ determines the set of possible receivers of a method call in environment Γ. For `invokevirtual`, the receiver is determined by late binding, and the virtual method call (VMC) resolution function res^{α}_{Γ} will be used, where α is a standard static analysis technique to resolve VMC.

$$Rec_{\Gamma}(i) = \begin{cases} \{staticT(o)\} & \text{if } i \in \{\texttt{invokespecial (o,n)}, \texttt{invokestatic (o,n)}\} \\ res^{\alpha}_{\Gamma}(o,n) & \text{if } i = \texttt{invokevirtual (o,n)} \end{cases}$$

Suppose that VMC resolution uses the *RTA* algorithm, i.e., $\alpha = RTA$, then the result of the resolution for object o and method n in environment Γ will be:

$$res^{\alpha}_{\Gamma}(o,n) = \{\tau \mid \tau \in IC_{\Gamma} \ \wedge \ \Gamma \vdash \tau <: staticT(o) \ \wedge \ n = lookup(n,\tau)\}$$

where IC_{Γ} is the set of instantiated classes in environment Γ, $staticT(o)$ gives the static type of object o and $lookup(n,\tau)$ corresponds to the signature of n, i.e., τ is a subtype of o's static type and method n is defined in class τ.

Given the set of possible receivers, calls are generated for each possible receiver. For each call, if the method's execution terminates normally, control will be given back to the next instruction of the caller. If the method terminates with an uncaught exception, the caller has to handle this propagated exception. If the current method is an entry method, m_e, then the program will terminate abnormally. The CFG extraction rules for method invocations produce edges for both ϱ_N=*NullPointerException* and for all propagated exceptions.

$\mathcal{R}^{i}_{p}$ is the set of the edges for normal terminating calls, $\mathcal{H}^{\varrho_N}_{p}$ is the set of edges to handle ϱ_N, and $\mathcal{N}^{x}_{p}$ defines the set of edges to handle all uncaught exceptions from all possible callees. We put the callees signature as an index of the `handle` label to differentiate between propagated exceptions from method calls and exceptions raised in the current method. Similar to generating outgoing edges for exceptional control points, $\mathcal{H}^{x}_{p}$ generates edges for successful/failed handlers for all exceptional nodes in CFG_n which is the CFG of method $n \in res^{\alpha}_{\Gamma}(o,n)$.

The CFG of a Java class C, denoted $c\mathcal{G}(C) : \textsc{Class} \rightarrow \mathcal{P}(V \times \textsc{Inst}_g \times V)$, is defined as the disjoint union of the CFGs of the methods in C. The CFG of a program P, denoted $\mathcal{G}(P) : \Pi \rightarrow \mathcal{P}(V \times \textsc{Inst}_g \times V)$, is the disjoint union of all CFGs of the classes in P.

3.1 CFG Correctness

In order to prove the soundness of the extracted flow graph we need to define the behavior of the flow graph. The following extends the behavior definition of flow graphs from [9], based on our extraction rules.

Definition 7 (CFG Behavior). *Let $\mathcal{G} = (\mathcal{M}, \mathbb{E}) : I$ be a closed flow graph with exceptions such that $\mathcal{M} = (V, L, \rightarrow, A, \lambda)$. The* behavior *of $\mathcal{G}$ is described by the specification $b(\mathcal{G})$, where $\mathcal{M}_g = (S_g, L_g, \rightarrow_g, A_g, \lambda_g)$ such that:*

- $S_g \in V \times (V)^*$, *i.e., states are pairs of control nodes and stacks of control nodes,*
- $L_g = \{\tau\} \cup L_g^C \cup L_g^X$ *where* $L_g^C = \{m_1\ l\ m_2 \mid l \in \{call, ret, xret\}, m_1, m_2 \in I^+\}$ *(the set of call and return labels) and* $L_g^X = \{l\ x \mid l \in \{throw,\ catch\}, x \in \textsc{Excp}\}$ *(the set of exceptional transition labels).*
- $A_g = A$ *and* $\lambda_g((v, \sigma)) = \lambda(v)$
- $\rightarrow_g \subset S_g \times S_g$ *is the set of transitions in* CFG_m *with the following rules:*

$$\begin{array}{lll}
\textit{[call]} & (v_1, \sigma) \xrightarrow{m_1\ call\ m_2}_g (v_2, v_1.\sigma) & \textit{if}\ m_1, m_2 \in I^+, v_1 \xrightarrow{call\ m_2}_{m_1} v_1', \\
 & & v_1' \in next(v_1),\ v_1 \nvDash\ \textsc{Excp} \\
 & & v_2 \vDash m_2,\ v_2 \in \mathbb{E},\ v_1 \vDash \neg r \\
\textit{[return]} & (v_2, v_1.\sigma) \xrightarrow{m_2\ ret\ m_1}_g (v_1', \sigma) & \textit{if}\ m_1, m_2 \in I^+, v_2 \vDash m_2 \wedge r \\
 & & v_1 \vDash m_1,\ v_1' \nvDash \textsc{Excp},\ v_1' \in next(v_1) \\
\textit{[xreturn]} & (v_2, v_1.\sigma) \xrightarrow{m_2\ xret\ m_1}_g (v_1', \sigma) & \textit{if}\ m_1, m_2 \in I^+,\ v_2 \vDash m_2,\ v_1 \vDash m_1 \\
 & & v_2 \xrightarrow{handle}_{m_2} v_2',\ v_1 \xrightarrow{handle}_{m_1} v_1' \\
 & & v_2 \vDash x,\ v_2' \vDash x \wedge r,\ v_1 \nvDash x,\ v_1' \vDash x, x \in \textsc{Excp}
\end{array}$$

$$\textit{[transfer]}\ (v, \sigma) \xrightarrow{\tau}_g (v', \sigma)\ \textit{if}\ m \in I^+,\ v \xrightarrow{i_g}_m v', v \vDash \neg r,\ v \nvDash \textsc{Excp},\ v' \nvDash\ \textsc{Excp}$$

$$\textit{[throw]}\ (v, \sigma) \xrightarrow{throw\ x}_g (v', \sigma)\ \textit{if}\ m \in I^+, v \xrightarrow{i_g}_m v', v \vDash \neg r, v' \vDash \textsc{Excp}$$

$$\textit{[catch]}\ (v, \sigma) \xrightarrow{catch\ x}_g (v', \sigma)\ \textit{if}\ m \in I^+, v \xrightarrow{handle}_m v', v \vDash \neg r \wedge \textsc{Excp}, v' \nvDash r, v' \nvDash \textsc{Excp}$$

Consider again the flow graph in Figure 1. One example run through its (branching, infinite-state) behavior, from an initial to a final configuration, is:

$$(v_0, \varepsilon) \xrightarrow{\tau} (v_1, \varepsilon) \xrightarrow{\tau} (v_2, \varepsilon) \xrightarrow{even\ call\ odd} (v_5, v_3) \xrightarrow{\tau} (v_6, v_3) \xrightarrow{\tau} (v_8, v_3) \xrightarrow{odd\ ret\ even} (v_3, \varepsilon)$$

To show the correctness of the extraction algorithm, we show that the extracted CFG of method m can match all possible moves during execution of m. In order to do this, we first define a mapping θ that abstracts JVM configurations to CFG behavioural configurations. Using θ, we can then prove that the behaviour of a CFG simulates the behaviour of the corresponding method in JBC.

Definition 8 (Abstraction Function for VM States). *Let* Vmc *be the set of JVM execution configurations and* S_g *the set of states in* $m\mathcal{G}$. *Then* $\theta : \textsc{Vmc} \rightarrow S_g$ *is defined inductively as follows:*

$$\begin{array}{rr}
\theta(\langle m, p, f, s, z\rangle.A; h) = \langle \circ_m^p, \theta(A; h)\rangle & \theta(\langle x\rangle_{exc}.\epsilon; h) = \langle \bullet_m^{\flat, x, r}, \epsilon\rangle \\
\theta(\langle m, p, f, s, z\rangle.\epsilon; h) = \langle \circ_m^p, \epsilon\rangle & \theta(\langle x\rangle_{exc}.\langle m, p, f, s, z\rangle.A; h) = \langle \bullet_m^{p, x}, \theta(A; h)\rangle
\end{array}$$

Now we can prove correctness of the CFG construction. Function θ specifies the corresponding JVM state in the extracted CFG. In order to match relating transitions we use simulation modulo relabeling: we map JVM transition labels $\textsc{Inst} \cup \{\epsilon\}$ to the CFG transition labels in CFG $\textsc{Inst}_g \cup \{handle\}$. Transition ϵ in the JVM labeling set denotes silent transitions: transitions of the JVM to handle raised exceptions.

$$\begin{array}{rcll}
\mathit{expr} & ::= & c \mid \texttt{null} & \text{(constants)}\\
 & \mid & \mathit{expr} \oplus \mathit{expr} & \text{(arithmetic)}\\
 & \mid & \mathit{tvar} \mid \mathit{lvar} & \text{(variables)}\\
 & \mid & \mathit{expr}.f & \text{(field access)}\\
\mathit{lvar} & ::= & \texttt{l} \mid \texttt{l}_1 \mid \texttt{l}_2 \mid \ldots & \text{(local var.)}\\
 & & \texttt{this} & \\
\mathit{tvar} & ::= & \texttt{t} \mid \texttt{t}_1 \mid \texttt{t}_2 \mid \ldots & \text{(temp. var.)}\\
\mathit{target} & ::= & \mathit{lvar} & \\
 & \mid & \mathit{tvar} & \\
 & \mid & \mathit{expr}.f &
\end{array}$$

$$\begin{array}{rcl}
\mathit{Assignment} & ::= & \mathit{lvar}\ \texttt{:=}\ \mathit{expr} \mid \mathit{expr}.f\,\texttt{:=}\mathit{expr}\\
\mathit{TempAssign} & ::= & \mathit{tvar}\ \texttt{:=}\ \mathit{expr}\\
\mathit{Return} & ::= & \texttt{vreturn}\ \mathit{expr} \mid \texttt{return}\\
\mathit{MethodCall} & ::= & \mathit{expr}.\texttt{m(}\mathit{expr}\texttt{,}\ldots\texttt{,}\ \mathit{expr}\texttt{)}\\
 & \mid & \mathit{target} := \mathit{expr}.\texttt{m(}\mathit{expr}\texttt{,}\ldots\texttt{,}\mathit{expr}\texttt{)}\\
\mathit{NewObject} & ::= & \mathit{target} := \texttt{new C(}\mathit{expr}\texttt{,}\ldots\texttt{,}\mathit{expr}\texttt{)}\\
\mathit{Assertion} & ::= & \texttt{notnull}\ \mathit{expr} \mid \texttt{notzero}\ \mathit{expr}\\
\mathit{instr} & ::= & \texttt{nop} \mid \texttt{if}\ \mathit{expr}\ \texttt{pc} \mid \texttt{goto pc}\\
 & \mid & \texttt{throw}\ \mathit{expr} \mid \texttt{mayinit C}\\
 & \mid & \mathit{Assignment} \mid \mathit{TempAssign}\\
 & \mid & \mathit{Return} \mid \mathit{MethodCall}\\
 & \mid & \mathit{NewObject} \mid \mathit{Assertion}
\end{array}$$

Fig. 3. Expressions and Instructions of BIR

Theorem 1 (CFG Simulation). *For a closed program P and corresponding flow graph $\mathcal{G}$, the behavior of $\mathcal{G}$ simulates the execution of P.*

Proof. For every possible JVM configuration c and instruction i, we establish the possible transitions to a set of configurations C based on the operational semantics. We apply θ to all elements in C, denoted $\Theta(C)$, to determine the abstract CFG configurations. Then we use the CFG construction algorithm to determine which edges are established for instruction i. These edges determine the possible transitions paths from $\theta(c)$ to the next CFG states S, and we show that the set S corresponds to the configurations $\Theta(C)$. To show that this indeed holds, we use a case analysis on Vmc. For more details we refer to Amighi's Master thesis [2]. □

4 Extracting Control-Flow Graphs from BIR

This section presents the two-phase transformation from Java bytecode into control-flow graphs using BIR as intermediate representation. First we briefly present BIR and the transformation from JBC into BIR. Then, we present the transformation from BIR into control-flow graphs and prove its correctness.

4.1 The BIR language

The BIR language is an intermediate representation of Java bytecode. The main difference with standard JBC is that BIR instructions are stack-less, i.e., they have explicit operators and do not operate over values stored in the operand stack. This subsection gives a brief overview of BIR, for a full account we refer to [5].

Syntax and Expression trees Figure 3 summarizes the BIR syntax. Its instructions operate over expression trees, i.e., arithmetic expressions composed of constants, operations, variables, and fields of other expressions ($\mathit{expr}.\texttt{f}$). BIR does not have operations over strings and booleans,these are transformed into methods calls by the BC2BIR transformation. The transformation algorithm discussed below reconstructs expression trees, i.e., it collapses one-to-many stack-based operations into a single expression. As a result, a program represented in BIR typically has fewer instructions than the original JBC program.

There are two kinds of variables in BIR: *var* and *tvar*. The first are identifiers that are also present in the original bytecode; the latter are new variables introduced by the transformation. Both variables and object fields can be target of an assignment.

Many of the BIR instructions have an equivalent JBC counterpart, e.g., `nop`, `goto` and `if`. A `vreturn` *expr* ends the execution of a method with return value *expr*, while `return` ends a *void* method. The `throw` instruction explicitly transfers control flow to the exception handling mechanism. Method call instructions are represented by the method signature. For non-*void* methods, the instruction assigns the result value to a variable.

In contrast to JBC, object allocation and initialization happen in a single step, during the execution of the `new` instruction. However, Java also has class initialization, i.e., the one-time initialization of a class's static fields. To preserve this class initialization order, BIR contains a special `mayinit` instruction. This behaves exactly as a `nop`, but indicates that at that point a class may be initialized for the first time.

Assertions The support for run-time exceptions in BIR is implemented in the form of special instructions called assertions. These instructions are inserted during the transformation of bytecode instructions that can potentially raise exceptions, as defined in the Java Virtual Machine specification.

We define $\mathcal{RE}$ as the set of supported run-time exceptions in BIR (following [3]). Figure 4 shows this set, and the function $\bar{\chi} : Assertion \to \mathcal{RE}$ that maps the assertion to the run-time exception it guards. Along the text we exemplify the use of assertions using `[notnull]` and `[notzero]` only, and its corresponding exceptions.

Assertion	$\mathcal{RE}$	*Assertion*	$\mathcal{RE}$
`[notnull]`	*NullPointerException*	`[notzero]`	*ArithmeticException*
`[checkbound]`	*IndexOutOfBoundsException*	`[checkcast]`	*ClassCastException*
`[notneg]`	*NegativeArraySizeException*	`[checkstore]`	*ArrayStoreException*

Fig. 4. $\bar{\chi}$: Mapping of Assertions and Runtime Exceptions

The `[notzero]` *expr* assertion is placed before all instructions containing an expression with division operation. It checks whether the divisor *expr* evaluates to zero, thus potentially raising an *ArithmeticException.* The `[notnull]` *expr* assertion is placed before any access to a reference and checks whether `expr` evaluates to a dereferenced object, thus raising a *NullPointerException.* In cases the assertion is successful, it behaves as a `[nop]`, and control-flow passes to the next instruction. In case of a failure, control is transferred to the exception handling mechanism, just like for a `[throw]` instruction. If a suitable exception handler is found, control is moved to the first instruction of this handler.

BIR Programs A BIR program is organized exactly the same way as a Java bytecode program. A program is a set of classes, ordered by an inheritance hierarchy. Every class consists of a name, methods and fields. A method's code is stored in an instruction array. However, in contrast to JBC, in BIR the indexes in the instruction array are sequential, starting with `0` for the entry control point.

4.2 Transformation from Java bytecode into BIR

Next we give a short overview of the `BC2BIR` transformation. In some points, the algorithm is quite complex, because it has to maintain consistency between object references and BIR variables. However, since the flow-graph extraction abstracts away from all data, these complex points are not relevant and we do not discuss them here. Instead we focus on the transformation of instructions, i.e., the $\texttt{BC2BIR}_{instr}$ function. For the complete algorithm, we refer to [5].

The transformation `BC2BIR` transforms a complete JBC program into BIR by symbolically executing the bytecode using an abstract stack. This stack is used to reconstruct expression trees. Moreover, it also stores references to uninitialized objects, used to correctly match them with the corresponding initialization instruction, and differentiate the to constructor of the super class.

Definition 9 (Abstract Stack). *Let* $\texttt{UR} = \{\texttt{UR}^{\texttt{C}}_{\texttt{pc}} | \texttt{C} \in \mathbb{C},\ \texttt{pc} \in \mathbb{N}\}$ *be the set of references to uninitialized objects with static type* $\texttt{C}$*, allocated at program counter* $\texttt{pc}$*. Let Expr be the set of expression trees in the BIR language. Then the* abstract stack *is defined as*

$$AbsStack = (Expr \cup \ \texttt{UR})*$$

The symbolic execution of the individual instructions is defined by a function $\texttt{BC2BIR}_{instr}$ that given a program counter, a JBC instruction and an abstract stack, outputs a set of BIR instructions and a modified abstract stack. In case there is no match for a pair of bytecode instruction and stack, the transformation function returns the *Fail* element, and the `BC2BIR` algorithm aborts. The function $\texttt{BC2BIR}_{instr}$ is defined as follows.

Definition 10 (BIR Transformation Function). *The rules defining the* instruction-wise transformation $\texttt{BC2BIR}_{instr} : \mathbb{N} \times instr \times AbsStack \rightarrow (instr_{BIR} * \times AbsStack) \cup Fail$ *from Java bytecode into BIR are given in Figure 5.*

As a remark, JBC instructions with similar semantics, but working on different types of operands (e.g., `adiv` and `fdiv`) are grouped as single instructions (e.g., `div`). As a convention, we use brackets to distinguish BIR instructions from their JBC counterpart. At several places, the transformation function introduces new variables $\texttt{t}^{\texttt{i}}_{\texttt{pc}}$ that maintain consistency between values on the stack and the value that it represents.

Input		Output	
Instr	Stack	Instrs	Stack
nop	as	$\emptyset$	as
pop	e:as	$\emptyset$	as
push c	as	$\emptyset$	c:as
dup	e:as	$\emptyset$	e:e:as
load x	as	$\emptyset$	x:as
add	e_1:e_2:as	$\emptyset$	e_1+e_2:as
div	e_1:e_2:as	[notzero e_2]	e_1/e_2::as

Input		Output	
Instr	Stack	Instrs	Stack
if pc’	e:as	[if e pc’]	as
goto pc’	as	[goto pc’]	as
return	as	[return]	as
vreturn	e:as	[return e]	as
athrow	e:as	[throw e]	as
new C	as	[mayinit C]	UR^{C}_{pc}:as
getfield f	e:as	[notnull e]	e:f:as

Input		Output		Condition
Instr	Stack	Instrs	Stack	
store x	e:as	[x:=e]	as	x $\notin$ as
		[t^0_{pc}:=x;x:=e]	as[t^0_{pc}/x]	x $\in$ as
putfield f	e':e:as	[notnull e;$FSave$(pc,f,as);e.f:=e']	as[t^i_{pc}/e_i]	
invokevirtual m	e'_1...e'_n:e:as	[notnull e;$Hsave$(pc,as)]		
		[e.m(e'_1...e'_n)]	as[t^j_{pc}/e_j]	m is *void*
		[t^0_{pc}:=e.m(e'_1...e'_n)]	t^0_{pc}:as[t^j_{pc}/e_j]	m not *void*
invokespecial m	e'_1...e'_n:e:as	[$Hsave$(pc,as);t^0_{pc}:=new C(e'_1...e'_n)]	as[t^j_{pc}/e_j]	e $=$ UR^{C}_{pc}
		[notnull e;$Hsave$(pc,as);e.m(e'_1...e'_n)]	as[t^j_{pc}/e_j]	otherwise

Fig. 5. Rules for BC2BIR$_{instr}$

JBC instructions if, goto, return and vreturn are transformed into corresponding BIR instructions (using the top of the stack as condition argument for the if instruction). The new instruction adds an unallocated object on the stack, and produces a mayinit instruction. The getfield f instruction reads a field from the object reference at the top of the stack. This might produce a *NullPointerException*, thus the transformation produces a notnull instruction.

For the store x instruction there are two cases. If the variable x is not yet on the stack, the assignment of the expression on the top of the stack to x is returned. Otherwise, first the current value of x is assigned to a newly created variable t^0_{pc}, and all occurrences of x on the stack are replaced by this new variable (denoted as[t^0_{pc}/x]).

The putfield f outputs a set of BIR instructions: first, a notnull assertion, to check if the accessed reference is made to a valid object. Then the auxiliary function $FSave$ introduces a set of Assignment instructions to temporary variables, for all occurrences of f on the stack; finally it creates the assignment instruction to the field (e.f).

The rule for virtual method calls (invokevirtual) generates a sequence of instructions. First there is a [notnull] assertion. Then any reference to objects on the stack that access the heap must be stored into newly introduced variables to remember its value, because objects on the heap can be altered during the method invocation. This is defined as function *Hsave*. Finally, there is the call instruction itself. If the method returns a value, a new variable is introduced to store the return value, and this is added to the abstract stack.

The transformation of invokespecial searches for an uninitialized reference on the stack after the method arguments to check if such call targets an object constructor. If such reference is not found , the transformation acts similarly to the case of virtual method calls. However if an uninitialized reference is found, it replaces the $\mathit{UR}^{\mathtt{C}}_{\mathtt{pc}}$ reference to the uninitialized object – added by the transformation of the `new` instruction – with a new variable $\mathtt{t}^{\mathtt{j}}_{\mathtt{pc}}$.

Figure 6 shows the JBC and BIR representations for the method `even`, presented in Figure 1. The example contains both a local variable (`$bcvar1`) and a new variable introduced by the transformation (`$irvar1`). We can observe reconstructed expression trees as the argument to the method invocation, and as the operand to the `[if]` instruction. The `[notnull this]` instruction is trivial, since it checks if the reference to the current object is valid, but it illustrates how assertions are placed before instructions that can raise exceptions.

```
Java bytecode                          BIR
public boolean even(int);              public bool even(int)
 0: iload_1                            0: if ($bcvar1 != 0) goto 2
 1: ifne 6                             1: vreturn 1
 4: iconst_1                           2: notnull this
 5: ireturn                            3: $irvar1 := this.odd($bcvar1-1)
 6: aload_0                            4: vreturn $irvar1
 7: iload_1
 8: iconst_1
 9: isub
10: invokevirtual boolean odd(int)
13: ireturn
```

Fig. 6. Comparison of method in JBC and BIR

4.3 Transformation from BIR into Control-Flow Graphs

The setup of the extraction algorithm is similar to that of BC2BIR. It iterates over the instructions of a method, using the transformation function $\mathtt{b}\mathcal{G}$. Each iteration outputs a set of triples of the form $V \times \mathit{Instr} \times V$. The extraction algorithm $\mathtt{b}\mathcal{G}$ takes as input a program counter and an instruction array for a BIR method. It outputs a set of edges. The set of edges can then be directly transformed in a control-flow graph as defined in Definition 6.

To define $\mathtt{b}\mathcal{G}$, we introduce auxiliary functions and definitions similar to the ones introduced in the direct extraction (in Section 3). $\bar{H}$ is the exceptions table from a given method. It contains the same entries as the JBC table, but has its control points translated to the BIR. The function $\hbar_{\bar{H}}(pc, x)$ searches for the first handler for the exception x (or a subtype) at position pc. The function $res^{\alpha}_{b}(o, n)$ returns all possible receivers for a method call, given the object reference and the method signature. The function $\bar{\mathcal{H}}^{pc}_{x}$ returns an edge after querying $\hbar$ for exception handlers. Also, $\bar{\mathcal{N}}^{pc}_{n}$ returns edges to exceptional flows for the method invocations that can terminate due to an uncaught exception, and consequently propagate it.

$$\bar{\mathcal{H}}_x^{pc} = \begin{cases} (\bullet_m^{\mathtt{pc},x}, handle, \circ_m^{t}) & \hbar_{\bar{H}} = t \neq 0 \\ (\bullet_m^{\mathtt{pc},x}, handle, \bullet_m^{\mathtt{pc},x,r}) & \hbar_{\bar{H}} = 0 \end{cases}$$

$$\bar{\mathcal{N}}_n^{pc} = \{(\circ_m^{\mathtt{pc}}, handle_n, \bullet_m^{\mathtt{pc},x}), \bar{\mathcal{H}}_x^{pc} \mid \bullet_n^{pc,x,r} \in \mathtt{b}\mathcal{G}(n), n \in res_b^{\alpha}(o,n)\}$$

$$\begin{array}{ll}
\mathtt{b}\mathcal{G}((pc,i),\bar{H}) = \emptyset & \text{if } i \in \mathit{TempAssign} \\
\mathtt{b}\mathcal{G}((pc,i),\bar{H}) = \{(\circ_m^{\mathtt{pc}}, i_b, \circ_m^{\mathtt{pc+1}})\} & \text{if } i \in \{\mathtt{[nop]},\mathtt{[mayinit]}\} \\
\mathtt{b}\mathcal{G}((pc,i),\bar{H}) = \{(\circ_m^{\mathtt{pc}}, i_b, \circ_m^{\mathtt{pc+1}})\} & \text{if } i \in \mathit{Assignment} \\
\mathtt{b}\mathcal{G}((pc,i),\bar{H}) = \{(\circ_m^{\mathtt{pc}}, i_b, \circ_m^{\mathtt{pc+1}}), \circ_m^{\mathtt{pc}}, i_b, \circ_m^{\mathtt{pc'}})\} & \text{if } i = \mathtt{[if}\ \mathit{expr}\ \mathtt{pc']} \\
\mathtt{b}\mathcal{G}((pc,i),\bar{H}) = \{(\circ_m^{\mathtt{pc}}, i_b, \circ_m^{\mathtt{pc'}})\} & \text{if } i = \mathtt{[goto\ pc']} \\
\mathtt{b}\mathcal{G}((pc,i),\bar{H}) = \{(\circ_m^{\mathtt{pc}}, i_b, \circ_m^{\mathtt{pc},r})\} & \text{if } i \in \mathit{Return} \\
\mathtt{b}\mathcal{G}((pc,i),\bar{H}) = \{(\circ_m^{\mathtt{pc}}, call(\tau,n), \circ_m^{\mathtt{pc+1}}), (\circ_m^{\mathtt{pc}}, i_b, \bullet_m^{\mathtt{pc},\varrho N})\} \cup & \text{if } i \in \mathit{NewObject} \\
\qquad\qquad\qquad\quad \{\bar{\mathcal{H}}_{\varrho N}^{pc}\} \cup \bar{\mathcal{N}}_{pc}^{n} & \\
\mathtt{b}\mathcal{G}((pc,i),\bar{H}) = \{(\circ_m^{\mathtt{pc}}, call(\tau,n), \circ_m^{\mathtt{pc+1}}) \mid \forall\tau \in res_b^{\alpha}\} \cup \bar{\mathcal{N}}_{pc}^{n} & \text{if } i \in \mathit{MethodCall} \\
\mathtt{b}\mathcal{G}((pc,i),\bar{H}) = \{(\circ_m^{\mathtt{pc}}, i_b, \bullet_m^{\mathtt{pc},x}), \bar{\mathcal{H}}_x^{pc} \mid x \in X\} & \text{if } i = \mathtt{[throw}\ X\mathtt{]} \\
\mathtt{b}\mathcal{G}((pc,i),\bar{H}) = \{(\circ_m^{\mathtt{pc}}, i_b, \circ_m^{\mathtt{pc+1}}), (\circ_m^{\mathtt{pc}}, i_b, \bullet_m^{\mathtt{pc},\bar{\chi}(i)}), \bar{\mathcal{H}}_{\bar{\chi}(i)}^{\mathtt{pc}}\} & \text{if } i \in \mathit{Assertion}
\end{array}$$

Fig. 7. Extraction rules for Control-flow graphs from BIR

Definition 11 (Control Flow Graph Extraction). *The* control-flow graph *extraction function* $\mathtt{b}\mathcal{G} : (\mathbb{N} \times \mathit{Instr}) \times \bar{H} \to \mathcal{P}((V, I_b, V))$ *is defined by the rules in Figure 7, where* $I_b = \mathit{Instr} \cup \{handle\}$*.*

The control-flow graph *for a method* m *is defined as* $\mathtt{b}\mathcal{G}(m) = \bigsqcup_{\forall i_{pc} \in instr_m} \mathtt{b}\mathcal{G}(pc, i_{pc}, \bar{H}_m)$*, where* $instr_m$ *is the instruction array for method* m*, and* i_{pc} *is the instruction with array index* pc*. The* control-flow graph *for a closed program* p *is defined as* $\mathtt{b}\mathcal{G}(p) = \bigsqcup_{\forall m \in pc} \mathtt{b}\mathcal{G}(m)$*.*

The extraction rules work as follows. Assignments to a newly introduced temporary variables, denoted by the *TempAssign* set, do not produce edges. Such instructions are produced by the BIR transformation to keep data consistent, but they do not have a correspondent edge on the direct extraction, thus we can ignore them. For the instructions in *Assignment* set, `[nop]` and `[mayinit]` a normal transition to the next control node is generated. The conditional jump `[if` *expr* `pc']` produces a branch in the CFG: control can go either to the next control point, or to the branch point `pc'`. The unconditional jump `goto pc'` adds a single transition to control point `pc'`. The `[return]` and `[vreturn` *expr*`]` instructions generate an internal transition to a return node, i.e., a node with the atomic proposition r. Notice that, although both nodes are tagged with the same `pc`, they are different, because their sets of atomic propositions are different.

The extraction rule for calls to constructors (`[new C]`) produces a single normal edge, since there is only one possible receiver for the call. Also, we produce a pair of edges relatives to *NullPointerException.* The BIR transformation does not produce a correspondent [notnull] instruction for such case, and at first we should not support such exceptional flows. However the direct algorithm contemplates such case, thus we produce these two exceptional edges for the sake of soundness. Moreover, $\bar{\mathcal{N}}_n^{pc}$ returns transitions to exceptional nodes due to uncaught exceptions, together with the appropriate exception handling transitions.

The extraction rule for method calls is similar to that of the direct extraction (in Section 3). Again, we assume that an appropriate virtual method calls resolution is used. We add a normal edge for each possible receiver returned from res_b^{α}. Again, $\bar{\mathcal{N}}_n^{pc}$ returns a pair of transitions for uncaught exceptions.

The `[throw` x`]` instruction, similarly to virtual method call resolution, depends on some kind of static analysis to find out the possible exceptions that can be thrown. The BIR transformation only provides the static type of the exception x . We define X as the set containing the static type of x and its subtypes. Thus we add one exceptional edge for each element of X, together with its correspondent edge after querying the exception table.

Finally, we cite the rule for assertion instructions. In this case, we create a normal edge, indicating that the execution was successful, one exceptional edge to mark the raise of an exception, a third edge, which shows if the instruction has an associated entry in the exceptions table.

4.4 CFG Extraction Correctness Proof

We now enunciate the correctness proof theorem for control-flow graphs extracted from the composition of `BC2BIR` and $b\mathcal{G}$ algorithms. We prove that given the same JBC program, the control-flow graph generated with the composition of algorithms simulates structurally the control-flow graph generated using the $m\mathcal{G}$ direct algorithm.

Theorem 2 (Structural Simulation of Control-Flow Graphs). *Let P be an arbitrary Java bytecode closed program. Then $b\mathcal{G} \circ BC2BIR(P)$ weakly simulates $m\mathcal{G}(P)$, considering the set $\mathcal{RE}$.*

The proof is stated using case analysis over the Java bytecode instructions set, and is available on-line [4]. Based on the previous proof that structural simulation implies behavioral simulation [9], we can conclude that the correctness of structural simulation of $m\mathcal{G}(P)$ by the control-flow graph produced in `BC2BIR`(P) implies also behavioural simulation.

5 Related Work

Sinha et. al. [13, 14] propose criteria for testing exception handling constructs in Java programs (Java source code). They consider the effect of exception propagation and exceptions type conversion. The proposed algorithm for CFG construction traverses the (Abstract Syntax Tree) AST of the program and then inter-procedural CFG (ICFG) is established. Normal CFG is constructed using algorithms proposed in [1].

[4] Available at `http://www.csc.kth.se/~pedrodcg/files/foveoos11-proof.pdf`

In a similar work Jiang [11] propose an algorithm to extract exceptional control-flow graph (ECFG) of C++ programs. In the proposed model of the programs implicit control-flow of exceptions and exceptions propagation is represented. Based on the inter-procedural ECFG (ICFG) they described techniques for path testing and definition-use testing of C++ programs.

Jo and Chang [12] propose a method to construct CFG by computing separately normal flow and exception flow of Java programs (Java source code). Using a set-constraints of exceptions and iterative fix-point method they compute exception propagation paths. They show that CFG of a program can be constructed by merging an exception flow graph onto a normal flow graph.

Our CFG extraction rules use the results of inter-procedural analysis and exception propagation from above mentioned work, however, none gives formal extraction rule and correctness proof.

6 Conclusion

This paper presents an efficient and precise control-flow graph extraction algorithm, and shows the proof outline of its correctness. To the best of our knowledge, this is the first control-flow graph extraction algorithm that has been proven correct. The proof is presented in pencil-and-paper style, but paves the ground for a second version using automated reasoning.

The algorithm is efficient and precise, because it uses an intermediate stackless representation. This allows to generate precise information about exceptional control-flow, and it keeps the generated control-flow graphs relatively small.

To prove correctness of the algorithm, i.e., to show that any behaviour of the extracted control-flow graph is an over-approximation of the program's behaviour, a second extraction algorithm is used that works directly on the bytecode. It is easy to prove correctness of this direct algorithm. To prove correctness of the indirect algorithm we show that the flow graphs it generates simulate structurally the flow graphs generated by the direct algorithm. Since structural simulation implies behavioural simulation, this gives us the desired result.

As future work, we are studying how the extraction algorithm could be adapted to a modular setting. Currently, only flow-graphs for complete programs can be extracted. However, our intention is to use the extracted flow-graphs as input for CVPP [10], a tool set for compositional verification of control-flow safety properties. In this setting, one often wishes to generate a flow-graph from an incomplete program. In addition, we are also studying how the techniques used in this paper can be used to prove correctness of an extraction algorithm that preserves some data of the original program, and how to use it for programs with multiple threads of execution.

Acknowledgments We would like to thank Dilian Gurov for his support and valuable comments. Also we thank the Celtique team in INRIA-Rennes for the cooperation and constant clarifications over the BIR language.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1986)
2. Amighi, A.: Flow Graph Extraction for Modular Verification of Java Programs. Master's thesis, KTH Royal Institute of Technology, Stockholm, Sweden (February 2011), http://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2011/rapporter11/amighi_afshin_11038.pdf, Ref.: TRITA-CSC-E 2011:038
3. Barre, N., Demange, D., Hubert, L., Monfort, V., Pichardie, D.: Sawja api documenation (June 2011), http://javalib.gforge.inria.fr/doc/sawja-api/sawja-1.3-doc/api/index.html
4. Besson, F., Jensen, T., Le Métayer, D., Thorn, T.: Model checking security properties of control flow graphs. J. of Computer Security 9(3), 217–250 (2001)
5. Demange, D., Jensen, T., Pichardie, D.: A provably correct stackless intermediate representation for java bytecode. Tech. Rep. 7021, Inria Rennes (2009), http://www.irisa.fr/celtique/demange/bir/rr7021-3.pdf, version 3, November 2010
6. Freund, S.N., Mitchell, J.C.: A type system for the java bytecode language and verifier. J. Autom. Reason. 30, 271–321 (August 2003), http://dx.doi.org/10.1023/A:1025011624925
7. Gurov, D., Huisman, M., Sprenger, C.: Compositional verification of sequential programs with procedures. Information and Computation 206(7), 840–868 (2008)
8. Hubert, L., Barré, N., Besson, F., Demange, D., Jensen, T., Monfort, V., Pichardie, D., Turpin, T.: Sawja: Static Analysis Workshop for Java. In: Formal Verification of Object–Oriented Software (FoVeOOS '10). LNCS, vol. 6528. Springer (2010)
9. Huisman, M., Aktug, I., Gurov, D.: Program models for compositional verification. In: International Conference on Formal Engineering Methods (ICFEM '08). LNCS, vol. 5256, pp. 147–166. Springer (2008)
10. Huisman, M., Gurov, D.: CVPP: A tool set for compositonal verification of control-flow safety properties. In: Formal Verification of Object–Oriented Software (FoVeOOS '10). LNCS, vol. 6528, pp. 107–121. Springer (2010)
11. Jiang, S., Jiang, Y.: An analysis approach for testing exception handling programs. SIGPLAN Not. 42, 3–8 (April 2007), http://doi.acm.org/10.1145/1288258.1288259
12. Jo, J.W., Chang, B.M.: Constructing control flow graph for java by decoupling exception flow from normal flow. In: ICCSA (1). pp. 106–113 (2004)
13. Sinha, S., Harrold, M.J.: Criteria for testing exception-handling constructs in java programs. In: Proceedings of the IEEE International Conference on Software Maintenance. pp. 265–. ICSM '99, IEEE Computer Society, Washington, DC, USA (1999), http://portal.acm.org/citation.cfm?id=519621.853364
14. Sinha, S., Harrold, M.J.: Analysis and testing of programs with exception handling constructs. IEEE Trans. Softw. Eng. 26, 849–871 (September 2000), http://portal.acm.org/citation.cfm?id=352825.352832

Static single information form for abstract compilation

Davide Ancona and Giovanni Lagorio

DISI, University of Genova, Italy
{davide,lagorio}@disi.unige.it

Abstract. In previous work we have shown that more precise type analysis can be achieved by exploiting union types and static single assignment (SSA) intermediate representation (IR) of code.
In this paper we exploit static single information (SSI), an extension of SSA proposed in literature and adopted by some compilers, to allow assignments of more precise types to variables in conditional branches. In particular, SSI can be exploited rather easily and effectively to assign more precise static types in presence of explicit runtime typechecking, a necessity that occurs rather often in statically typed object-oriented languages, and even more than often in dynamically typed ones.
We show how the use of SSI form can be smoothly integrated with abstract compilation, our approach to static type analysis. In particular, we define abstract compilation based on union and nominal types for a simple dynamically typed Java-like language in SSI form with a runtime typechecking operator, to show how precise the proposed analysis can be.

1 Introduction

In previous work [6] we have shown that more precise type analysis can be achieved by exploiting union types and static single assignment (SSA) [7] intermediate representation (IR) of code. Most modern compilers (among others, GNU's GCC [13], the SUIF compiler system [12], and JIT compilers including Java HotSpot [11], and Java Jikes RVM [9]) implement efficient algorithms for translating code in SSA IR [8], therefore focusing on analysis of SSA IR not only allows more precise results, but also favors reuse of compiler technology, and better integration with existing compilers. In particular, we have studied how *abstract compilation* [5, 4, 6] can naturally take advantage of SSA IR when union types are considered. Abstract compilation aims to reconcile static type analysis and symbolic execution: one can check whether the execution of a certain expression e is type safe, when variable contents range over possibly infinite sets of values (represented by types), by solving a goal, obtained by abstract compilation of e, w.r.t. the coinductive[1] semantics of the constraint logic program automatically generated from the source program in which the expression is executed.

[1] Coinduction allows proper treatment of recursive types and methods [5].

Abstract compilation fosters *plug-and-play* static type analysis since one can provide several compilation schemes for the same language, each corresponding to a different kind of analysis, without changing the inference engine that implements the coinductive semantics of constraint logic programs [15, 14, 4]. For instance, in previous work we have defined compilation schemes for Java-like languages based on union and structural object types, to support parametric and data polymorphism, (that is, polymorphic methods and fields) that allow quite precise type analysis, and a smooth integration with the nominal type annotations contained in the programs [5]; other proposed compilation schemes aim to detect uncaught exceptions for Java-like languages [4], or to integrate SSA IR in presence of imperative features [6].

In this paper we exploit static single information (SSI), an extension of SSA proposed in literature [2, 17], to allow assignments of more precise types to variables in conditional branches. SSI has been already adopted by compilers as LLVM [18], PyPy [3], and SUIF [16], and proved to be more effective than SSA for performing data flow analysis, program slicing, and interprocedural analysis.

In particular, we show how SSI can be exploited rather easily and effectively by abstract compilation to assign more precise static types in presence of explicit runtime typechecking, a necessity that occurs rather often in statically typed object-oriented languages [19], and even more than often in dynamically typed ones.

To this aim, we formally define the operational semantics of a simple dynamically typed Java-like language in SSI form equipped with a runtime typechecking operator, and then provide an abstract compilation scheme based on union and nominal types supporting more precise type analysis of branches guarded by explicit runtime typechecks.

The paper is structured as follows: Section 2 introduces SSA and SSI IRs and motivates their usefulness for type analysis; Section 3 formally defines the SSI IR of a dynamically typed Java-like language equipped with an operator **instanceof** for runtime typechecking. Section 4 presents a compilation scheme for the defined IR, based on nominal and union types, and Section 5 concludes with some considerations on future work. Abstract compilation of the code examples in Section 2 together with the results of the resolution of some goals can be found in an extended version of this paper.[2]

2 Type analysis with SSA and SSI

In this section SSA and SSI IRs are introduced and their usefulness for type analysis is motivated.

Type analysis with static single assignment form

Method `read()` declared below, in a hypothetical dynamically typed Java-like language, creates and returns a shape which is read through method `nextLine()`

[2] Available at ftp://ftp.disi.unige.it/person/AnconaD/foveoos11long.pdf

that reads the next available string from some input source. The partially omitted methods readCircle() and readSquare() read the needed data from the input, create, and return a new corresponding instance of Circle or Square.

```
class ShapeReader {
   ...
   nextLine() {...}
   readCircle() { ... return new Circle(...); }
   readSquare() { ... return new Square(...); }
   read() {
      st = this.nextLine();
      if(st.equals("circle")) {
         sh = this.readCircle();
         this.print("A circle with radius ");
         this.print(sh.getRadius());
      }
      else if(st.equals("square")) {
         sh = this.readSquare();
         this.print("A square with side ");
         this.print(sh.getSide());
      }
      else throw new IOException();
      this.print("Area = ");
      this.print(sh.area());
   }
}
```

Although method read() is type safe, no type can be inferred for st to correctly typecheck the method; indeed, when method area() is invoked, variable st may hold an instance of Circle or Square, therefore the most precise type that can be correctly assigned to st is Circle $\vee$ Square. However, if st has type Circle $\vee$ Square, then both sh.getRadius() and sh.getSide() do not typecheck.

There are two different kinds of approaches to solve the problem shown above. Either one defines a rather sophisticated flow-sensitive type system able to associate different types with different occurrences of the same variable, or one can typecheck the SSA IR in which the method is compiled.

In an SSA IR the value of each variable is determined by exactly one assignment statement [7]. To obtain this property, a flow graph is built, and a suitable renaming of variables is performed to keep track of the possibly different versions of the same variable; following Singer's terminology [17] we call these versions *virtual registers*. Conventionally, this is achieved by using a different subscript for each virtual register corresponding to the same variable. For instance, in the SSA IR of method read() (Figure 1) there are three virtual registers ($\texttt{sh}_0$, $\texttt{sh}_1$ and $\texttt{sh}_2$) for the variable sh.

To transform a program into SSA form, pseudo-functions, conventionally called φ-functions, have to be inserted to correctly deal with merge points. For instance, in block 5 the value of sh can be that of either $\texttt{sh}_0$ or $\texttt{sh}_1$, therefore a new virtual register $\texttt{sh}_2$ has to be introduced to preserve the SSA property. The

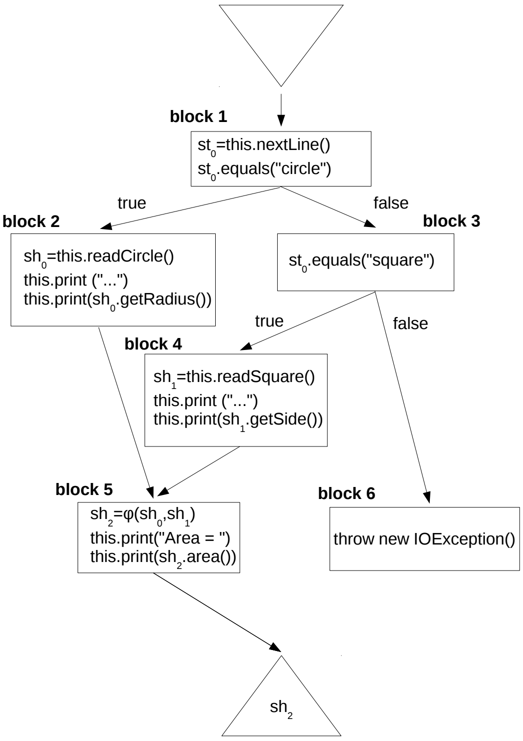


Fig. 1. Control flow graph corresponding to the body of method `read`

expression φ`(sh`$_0$`,sh`$_1$`)` keeps track of the fact that the value of `sh`$_2$ is determined either by `sh`$_0$ or `sh`$_1$.

At the level of types φ-functions naturally correspond to the union type constructor (Figure 2): arrows correspond to data flow and, as usual, to ensure soundness the type at the origin of an arrow must be a subtype of the type the arrow points to. In the figure, $\tau_0, \tau_1 \leq \tau_0 \vee \tau_1 \leq \tau_2$.

The transformation of a source program into its SSA IR is standard [7], and there exists a quite efficient algorithm to perform it [8], therefore it is more convenient to define abstract compilation for programs in SSA IR. While flow graphs are used for generating SSA IR, here we adopt a textual language more suitable for defining an abstract compilation scheme. For instance, the SSA IR of method `read()` is the following:

```
read() {
  b1:{st0 = this.nextLine();
      if(st0.equals("circle"))
         jump b2;
      else
         jump b3;}
  b2:{sh0=this.readCircle();
```

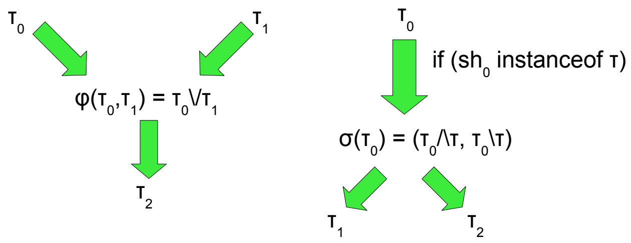

τ_0
τ_1
$\varphi(\tau_0,\tau_1) = \tau_0 \vee \tau_1$
τ_2
τ_0
if (sh_0 instanceof τ)
$\sigma(\tau_0) = (\tau_0 \wedge \tau, \tau_0 \setminus \tau)$
τ_1
τ_2

Fig. 2. Type theoretic interpretation of φ-functions and σ-functions

```
        this.print("A circle with radius ");
        this.print(sh0.getRadius());
        jump b5;}
    b3:{if(st0.equals("square"))
           jump b4;
        else
           jump b6;}
    b4:{sh1=this.readSquare();
        this.print("A square with side ");
        this.print(sh1.getSide());
        jump b5;}
    b5:{sh2=φ(sh0,sh1);
        this.print("Area = ");
        this.print(sh2.area());
        jump out;}
    b6:{throw new IOException();}
    out:{return sh2;}
}
```

The body of a method in IR is a sequence of uniquely labeled blocks; each block ends with either a conditional or unconditional `jump`, a **return** or a **throw**[3]. For simplicity, we require that only the last block[4] contains the **return** statement.

Type analysis with static single information form

Let us consider method `largerThan(sh)` of class `Square`, where **instanceof** is exploited to make the method more efficient in case the parameter `sh` contains an instance of (a subclass) of `Square`.

```
class Square {
...
   largerThan(sh) {
     if(sh instanceof Square)
```

[3] In the formal treatment that follows we omit exceptions for simplicity.

[4] This can be always obtained by introducing new virtual registers and inserting a φ-function in case of multiple returned values.

```
            return this.side > sh.side;
        else
            return this.area() > sh.area();
    }
}
```

The method is transformed in the following SSA IR:

```
  largerThan(sh$_0$) {
    b1:{if(sh$_0$ instanceof Square)
              jump b2;
          else
              jump b3;}
    b2:{r$_0$=this.side > sh$_0$.side;
          jump out;}
    b3:{r$_1$=this.area() > sh$_0$.area();
          jump out;}
    out:{r$_2$=$\varphi$(r$_0$,r$_1$);
          return r$_2$;}
}
```

Since variable sh is not updated, both blocks b2 and b3 refer to the same virtual register sh$_0$. As a consequence, the only possible type that can be correctly associated with sh$_0$ is Square, thus making the method of little use. However, this problem can be encompassed if one considers the SSI IR of the method [2, 17].

```
  largerThan(sh$_0$) {
    b1:{if(sh$_0$ instanceof Square) with (sh$_1$,sh$_2$) = $\sigma$(sh$_0$)
                                         (this$_1$,this$_2$) = $\sigma$(this$_0$)
              jump b2;
          else
              jump b3;}
    b2:{r$_0$=this$_1$.side > sh$_1$.side;
          jump out;}
    b3:{r$_1$=this$_2$.area() > sh$_2$.area();
          jump out;}
    out:{r$_2$=$\varphi$(r$_0$,r$_1$);
          return r$_2$;}
}
```

SSI is an extension of SSA enforcing the additional constraint that all variables must have different virtual registers in the branches of conditional expressions; such a property is obtained by a suitable renaming and by the insertion of σ-functions at split points. As a consequence, suitable virtual registers and σ-functions have to be introduced also for the read-only pseudo-variable **this**.

The notion of σ-function is the dual of φ-function (Figure 2); the type theoretic interpretation of σ depends on the specific kind of conditional context. If such a context is of the form (sh$_0$ **instanceof** Square) as in the example, then σ splits the type τ_0 of sh$_0$ in the type $\tau_0 \wedge$Square, assigned to sh$_1$, and in the type τ_0\\`Square`, assigned to $\texttt{sh}_2$, where the intersection and the complement operators have to be properly defined (see Section 4). For instance, if $\texttt{sh}_0$ has type `Square`$\vee$`Circle`, then $\texttt{sh}_1$ has type `(Square`$\vee$`Circle)`$\wedge$`Square`=`Square`, and $\texttt{sh}_2$ has type `(Square`$\vee$`Circle)`\\`Square`=`Circle`, therefore `Square`$\vee$`Circle` turns out to be a valid type for the parameter $\texttt{sh}_0$ of the method `largerThan`. For what concerns **this**, in this particular example no real split is performed: if we assume that $\textbf{this}_0$ has type `Square`, then `Square` is split in `(Square,Square)`, and both $\textbf{this}_1$ and $\textbf{this}_2$ have type `Square`.

3 Language definition

In this section we formally define an SSI IR for a simple dynamically typed Java-like language equipped with an **instanceof** operator for performing runtime typechecking.

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \overline{\mathit{cd}}^{n}\ e \\
\mathit{cd} & ::= & \texttt{class}\ c_1\ \texttt{extends}\ c_2\ \{\ \overline{f}^{n}\ \overline{\mathit{md}}^{k}\ \} \quad (c_1 \neq \mathit{Object}, \mathit{Bool}, c_2 \neq \mathit{Bool}) \\
\mathit{md} & ::= & m(\overline{x}^{n})\ \{\overline{b}^{n}\} \\
b & ::= & l{:}e \\
r & ::= & x_i \\
e & ::= & r \mid \texttt{false} \mid \texttt{true} \mid \texttt{new}\ c(\overline{e}^{n}) \mid e.f \mid e_0.m(\overline{e}^{n}) \mid e_1; e_2 \mid r = e \\
 & & \mid e_1.f = e_2 \mid \texttt{jump}\ l \mid r = \varphi(\overline{r}^{n}) \mid \texttt{return}\ r \\
 & & \mid \texttt{if}\ (r\ \texttt{instanceof}\ c)\ \texttt{with}\ \overline{(r', r'') = \sigma(r''')}^{n}\ \texttt{jump}\ l_1\ \texttt{else jump}\ l_2
\end{array}
$$

Syntactic assumptions: inheritance is not cyclic, method bodies are in correct SSI form and are terminated with a unique `return` statement, method and class names are disjoint, no name conflicts in class, field, method and parameter declarations, `new` $c(\overline{e}^{n})$ allowed only if $c \neq \mathit{Bool}$, and declared parameters cannot be `this`.

Fig. 3. SSI intermediate language

A program is a collection of class declarations followed by a main expression e with no free variables. The notation $\overline{\mathit{cd}}^{n}$ is a shortcut for $\mathit{cd}_1, \ldots, \mathit{cd}_n$. A class declares its direct superclass (only single inheritance is supported), its fields, and its methods. Two predefined classes are available: *Object*, the usual root class of the inheritance tree, and *Bool*, the class of the two literals `false` and `true`; such a class cannot be extended, and does not provide any constructor.

Every class, except *Bool*, comes equipped with the implicit constructor with parameters corresponding to all fields, in the same order as they are inherited and declared, but for simplicity no user declared constructors can be added.

Method bodies are sequences of uniquely labeled blocks that contain sequences of expressions. We assume that all blocks contain exactly one jump, necessarily placed at the end of the block. Three different kinds of jumps are considered: local unconditional and conditional jumps, and returns from methods. Method bodies are implicitly assumed to be in correct SSI IR: each virtual register is determined by exactly one assignment statement, and all variables must have different virtual registers in the branches of conditional expressions. Finally, all method bodies contain exactly one return expression, which is always placed at the end of the body.

The receiver object can be referred inside method bodies with the special implicit parameter `this`; besides usual statements and expressions, we consider φ and σ pseudo-function assignments.

In $r = \varphi(\overline{r}^n)$, $n \geq 2$ and all virtual registers $\overline{r}^n$ occurring in φ are assumed to refer to the same variable, denoted by $var(r_1) = \ldots = var(r_n)$.

All σ-functions are used in association with conditional jumps; each virtual register r occurring in either branches has to be split into two new distinct versions used in the blocks labeled by $\mathtt{l}_1$, and $\mathtt{l}_2$, respectively. The conditions we consider are only of the form (r `instanceof` c) since our aim is studying how type analysis can be enhanced by SSI in the presence of explicit runtime typechecks; however, more elaborated forms of conditions can be expressed in terms of the more primitive form (r `instanceof` c) by simple transformations performed by the compiler front-end. For instance, the statement

```
if (x.m1() instanceof A && y.m2() instanceof B) e₁ else e₂
```

can be transformed in the equivalent SSI IR

```
b0 : {z₀=xₖ.m1();
       if (z₀ instanceof A) with ... jump b1 else jump b3;}
b1 : {w₀=yₙ.m2();
       if (w₀ instanceof B) with ... jump b2 else jump b3;}
b2 : {e'₁}
b3 : {e'₂}
```

where $\mathtt{z}_0$ and $\mathtt{w}_0$ are fresh, e'_1 and e'_2 are the SSI IRs of e_1 and e_2, respectively, and σ-functions assignments (that depend on e_1 and e_2) have been omitted. Depending on the types and abstract compilation scheme under consideration, there could be other kinds of conditions for which SSI would improve type analysis; for instance, if an abstract compilation scheme allows analysis of null references, such an analysis could be enhanced by SSI in the case of conditional expressions with conditions of the form (x == **null**). On the other hand, for conditions of the form (x_1 < x_2) SSI does not help refine type analysis as long as the abstract compilation scheme maps numeric values to the standard primitive types **int**, **float**, and **double**; in this case the type theoretic interpretation of σ-functions is the loosest one: $\sigma(\tau)$=(τ,τ) (hence, no split is actually performed).

Semantics: To define the small step semantics of the language we first need to specify values v (see Figure 4), which are either the literals `false` and `true` (recall that they are predefined instances of the *Bool* class) or identities o of dynamically created objects. Furthermore, we add *frame expressions* $ec\{e\}$, where ec is an execution context; frame expressions are *runtime expressions*, that is, expressions that represent intermediate evaluation steps and that are needed for defining the small step semantics of method calls. An execution context ec is a pair consisting of a stack frame fr and a code address a. A frame expression $\langle fr, a\rangle\{e\}$ corresponds to the execution of a call to a method m declared in class c, where e is the residual expression (yet to be evaluated) of the currently executed block, fr is the stack frame of the method call, $a = \mu.l$ is the address of the current block, where $\mu = c.m$ is the fully qualified name of the method, and l is the label of the current block.

Stack frames fr map variables and virtual registers to their corresponding values. These frames are represented by a pair of lists of associations, $x \mapsto v$ and $r \mapsto v$, where variables and virtual registers are all distinct. Keeping track of the values of both virtual registers and their associated variable allows for a simple semantics, as discussed below.

Heaps $\mathcal{H}$ map object identifiers o to objects, that is, pairs consisting of a class name c and the set of field names f with their corresponding value v.

Figure 5 shows the execution rules. The main evaluation judgment has shape $\mathcal{H} \vdash e \to \mathcal{H}', e'$, meaning that e rewrites to e' in $\mathcal{H}$, yielding the new heap $\mathcal{H}'$. Furthermore, rule (ctx-closure) uses the auxiliary judgment $\mathcal{H}, ec \vdash e \to \mathcal{H}', ec', e'$, meaning that redex e rewrites to e' in $\mathcal{H}$ and ec, yielding the new execution context ec' and heap $\mathcal{H}'$. Both judgments, and all auxiliary functions should be parametrized by the whole executing program, $\overline{cd}^n$, which is kept implicit. The execution of the main expression of a program starts from an empty heap $\epsilon_{\mathcal{H}}$ and an empty stack frame ϵ, annotated by the pair $\langle\bot,\bot\rangle$, since the main expression is not actually contained in any block/method.

The main evaluation judgment is defined by the three rules (meth-call) (a new frame is pushed on the stack), (ctx-closure) (evaluation continues in the currently active frame), and (return) (the current active frame is popped from the stack).

In rule (meth-call), the object referenced by o is retrieved in order to find its class, c. Then, the auxiliary functions *firstBlock* and *params* return the first block of the method and its parameter names, respectively. The result of the evaluation is a frame expression, where the new stack frame maps parameters to their corresponding passed arguments, and `this` to the reference o, and the code address is the fully qualified name of the invoked method, $c.m$, plus the label of its first block, l. Finally, the resulting expression is the context applied to the body of the first block.

Rule (ctx-closure) performs a single computation step in the currently active frame (corresponding to the most nested frame expression). The execution context is extracted by *currentEC*;[5] then, if the redex e rewrites to e' yielding $\mathcal{H}'$ and ec' (see the other rules defining the auxiliary evaluation judgment), then the expression $\mathcal{C}[e]$ rewrites to $\mathcal{C}'[e']$, yielding the new heap $\mathcal{H}'$; context $\mathcal{C}'[\ ]$ is obtained from $\mathcal{C}[\ ]$ by updating the frame expression corresponding to the currently active frame with the new execution context ec'.

In rule (return) the currently active frame is removed, and the resulting expression is the context applied to the value associated with the returned virtual register r in the frame.

[5] The straightforward definitions of *currentEC* and *updateEC* have been omitted.

Rule (var) models the access to a virtual register (`this` is considered a special read-only local variable), by simply extracting the corresponding value from the stack frame fr; this works because of the way the assignment is handled in Rule (var-asn).

Variable and field assignments evaluate to their right values; rule (var-asn) models virtual register assignments, and has the side effect of updating, in the current stack frame fr, the values of both the virtual register r and its associated variable x. This implements a cache write-through strategy, where virtual registers cache values that are to be stored in the memory location corresponding to the variable to which virtual registers refer to; in this way evaluation of φ-function is simpler (see comments below).

Rule (fld-asn) models field assignments; in that case, the object referenced by o is retrieved from the heap, and its value updated.

Rule (seq) models the fact that a value is discarded when followed by another.

Rule (phi) models the assignment of a phi-function, which accesses to a (set $\overline{r}^n$ of different versions of the same) local variable x, by assigning the value contained in x (this is correct because of the write-through semantics of the Rule (var-asn)) to the virtual register r'.

Rule (new) models object creation; a new object, identified by a fresh reference o, is added to the heap $\mathcal{H}$. The fields $\overline{f}^n$ of the newly created object are initialized by the value passed to the constructor.

Rule (fld-acc) models field accesses; its evaluation is quite trivial: the object is retrieved from the heap, and the resulting expression is the value of the selected field.

Rules (jump) and (if) model unconditional and conditional jumps, respectively. These are the only rules that modify the label-part of the execution context. The evaluation of a jump, which, by construction, is known to be the last expression of a sequence, corresponds to replacing the jump expression itself with the expression e contained in the block labeled l' and updating the stack frame annotation accordingly.

The conditional jump (rule (if)) has to both choose which branch to execute and which virtual registers have to be updated, depending on whether the value of the register r (contained in $fr(r)$) is a reference to an object of a subclass of c. If the referenced object is indeed an instance of c, then the target label is l_1 and the virtual registers $\overline{r'}^n$ are updated; otherwise, the target label is l_2 and the virtual registers $\overline{r''}^n$ are updated.

4 Abstract compilation

In this section we define an abstract compilation scheme for programs in the SSI IR presented in Section 3. Programs are translated into a Horn formula Hf (that is, a logic program) and a goal B; type analysis amounts to resolve B w.r.t. the coinductive semantics (that is the greatest Herbrand model) of Hf [5].

In previous work we have defined compilation schemes based on union and structural object types, to support parametric and data polymorphism, (that

$$\begin{array}{rcl}
v & ::= & \mathtt{false} \mid \mathtt{true} \mid o \quad \text{(values)}\\
e & ::= & ec\{e\} \mid \ldots \quad \text{(runtime expressions; that is, frame expressions plus}\\
 & & \qquad\qquad\quad \text{source expressions as defined in Figure 3)}\\
ec & ::= & \langle \mathit{fr}, a\rangle \quad \text{(execution context)}\\
\mathit{fr} & ::= & \overline{x \mapsto v}^{j}\,\overline{r \mapsto v}^{k} \quad \text{(stack frames)}\\
a & ::= & \mu.l \quad \text{(block addresses, where } \mu \text{ are full method names } c.m\text{)}\\
\mathcal{H} & ::= & \overline{o \mapsto \langle c, \overline{f \mapsto v}^{j}\rangle}^{k} \quad \text{(heaps)}\\
\mathcal{C}[\cdot] & ::= & [\cdot] \mid ec\{\mathcal{C}[\cdot]\} \mid \mathtt{new}\ c(\overline{v}^{n}, \mathcal{C}[\cdot], \overline{e}^{j}) \mid \mathcal{C}[\cdot].f \mid \mathcal{C}[\cdot].m(\overline{e}^{k}) \mid v_0.m(\overline{v}^{j}, \mathcal{C}[\cdot], \overline{e}^{k})\\
 & & \mid \mathcal{C}[\cdot]; e \mid v;\mathcal{C}[\cdot] \mid x = \mathcal{C}[\cdot] \mid \mathcal{C}[\cdot].f = e \mid v.f = \mathcal{C}[\cdot]\\
 & & \mid \mathtt{if}\ (\mathcal{C}[\cdot])\ \mathtt{with}\ \overline{(x', x'') = \sigma(x''')}^{n}\ \mathtt{jump}\ l_1\ \mathtt{else}\ \mathtt{jump}\ l_2
\end{array}$$

Fig. 4. Syntactic definitions instrumental to the operational semantics

$$\text{(meth-call)}\ \frac{\begin{array}{l}\mathcal{H}(o) = \langle c, \_\rangle\\ \mathit{firstBlock}(c.m) = l : e\\ \mathit{params}(c.m) = \overline{r}^{n}\\ \mathit{fr} = \overline{r \mapsto v}^{n}, \mathtt{this}_0 \mapsto o\end{array}}{\begin{array}{l}\mathcal{H} \vdash \mathcal{C}[o.m(\overline{v}^{n})]\\ \qquad \to \mathcal{H}, \mathcal{C}[\langle \mathit{fr}, c.m.l\rangle\{e\}]\end{array}} \qquad \text{(ctx-closure)}\ \frac{\begin{array}{c}\mathit{currentEC}(\mathcal{C}[\cdot]) = ec\\ \mathcal{H}, ec \vdash e \to \mathcal{H}', ec', e'\\ \mathcal{C}'[\cdot] = \mathit{updateEC}(\mathcal{C}[\cdot], ec')\end{array}}{\mathcal{H} \vdash \mathcal{C}[e] \to \mathcal{H}', \mathcal{C}'[e']}$$

$$\text{(return)}\ \frac{}{\begin{array}{l}\mathcal{H} \vdash \mathcal{C}[\langle \mathit{fr}, a\rangle\{\mathtt{return}\ r\}]\\ \qquad \to \mathcal{H}, \mathcal{C}[\mathit{fr}(r)]\end{array}} \qquad \text{(fld-acc)}\ \frac{\mathcal{H}(o) = \langle c, \overline{f \mapsto v}^{n}\rangle \quad f = f_j}{\mathcal{H}, ec \vdash o.f \to \mathcal{H}, ec, v_j}$$

$$\text{(var)}\ \frac{}{\mathcal{H}, \langle \mathit{fr}, a\rangle \vdash r \to \mathcal{H}, \langle \mathit{fr}, a\rangle, \mathit{fr}(r)} \qquad \text{(new)}\ \frac{\begin{array}{c}o \text{ fresh in } \mathcal{H}\\ \mathit{fieldNames}(c) = \overline{f}^{n}\end{array}}{\begin{array}{l}\mathcal{H}, ec \vdash \mathtt{new}\ c(\overline{v}^{n})\\ \qquad \to \mathcal{H}[\langle c, \overline{f \mapsto v}^{n}\rangle/o], ec, o\end{array}}$$

$$\text{(seq)}\ \frac{}{\mathcal{H}, ec \vdash v_1; v_2 \to \mathcal{H}, ec, v_2} \qquad \text{(var-asn)}\ \frac{x = \mathit{var}(r) \quad x \neq \mathtt{this}}{\begin{array}{l}\mathcal{H}, \langle \mathit{fr}, a\rangle \vdash r = v\\ \qquad \to \mathcal{H}, \langle \mathit{fr}[v/r, v/x], a\rangle, v\end{array}}$$

$$\text{(fld-asn)}\ \frac{\begin{array}{l}\mathcal{H}(o) = \langle c, \overline{f \mapsto v}^{n}\rangle\\ f = f_j \quad \text{if } i = j \text{ then } v'_i = v\\ \qquad\qquad\qquad\ \ \text{else } v'_i = v_i\end{array}}{\begin{array}{l}\mathcal{H}, ec \vdash o.f = v\\ \qquad \to \mathcal{H}[\langle c, \overline{f \mapsto v'}^{n}\rangle/o], ec, v\end{array}} \qquad \text{(jump)}\ \frac{\mathit{block}(\mu.l') = l' : e}{\begin{array}{l}\mathcal{H}, \langle \mathit{fr}, \mu.l\rangle \vdash \mathtt{jump}\ l'\\ \qquad \to \mathcal{H}, \langle \mathit{fr}, \mu.l'\rangle, e\end{array}}$$

$$\text{(phi)}\ \frac{v = \mathit{fr}(\mathit{var}(r_1))}{\mathcal{H}, \langle \mathit{fr}, a\rangle \vdash r' = \varphi(\overline{r}^{n}) \to \mathcal{H}, \langle \mathit{fr}[v/r'], a\rangle, v}$$

$$\text{(if)}\ \frac{\begin{array}{l}\mathcal{H}(\mathit{fr}(r)) = \langle c', \_\rangle\\ \text{if } c' \leq c \text{ then } \ l' = l_1, \mathit{fr}' = \mathit{fr}[\overline{\mathit{fr}(r''')/r'}^{n}]\\ \qquad\qquad \text{else} \quad l' = l_2, \mathit{fr}' = \mathit{fr}[\overline{\mathit{fr}(r''')/r''}^{n}]\\ \mathit{block}(\mu.l') = l' : e\end{array}}{\begin{array}{l}\mathcal{H}, \langle \mathit{fr}, \mu.l\rangle \vdash \ \mathtt{if}\ (r\ \mathtt{instanceof}\ c)\ \mathtt{with}\ \overline{(r', r'') = \sigma(r''')}^{n}\\ \qquad\qquad\qquad \mathtt{jump}\ l_1\ \mathtt{else}\ \mathtt{jump}\ l_2\\ \qquad\qquad\qquad \to \mathcal{H}, \langle \mathit{fr}', \mu.l'\rangle, e\end{array}}$$

Fig. 5. Small-step semantics

is, polymorphic methods and fields) that allow quite precise type analysis, but pose problems in terms of termination of the resolved goals. In this paper we present a simpler compilation scheme based on union and nominal object types, that allows a less precise type analysis on the one hand (but still more precise than that obtained with the standard type systems of mainstream languages as Java and C#) but, on the other hand, avoids non termination problems with goal resolution, since the space of all possible valid types for a given program is always bounded. Therefore, while coinductive constraint logic programming [4] (where constraints are expressed in terms of the subtyping relation) is needed for structured types to ensure termination of the resolution of some goals, here would only allow more precise and efficient analysis; however, here subtyping is treated as an ordinary predicate to make the presentation simpler.

Summarizing, here we use nominal rather than structured types for the following reasons: SSI allows more precise type analysis even in the presence of less expressive types, the presentation is simpler, and, last but not least, we take advantage of the *plug-and-play* facility offered by the abstract compilation approach by providing yet another compilation scheme; in practice, more advanced compilations schemes could be adopted, including structural [5, 6] and exception types [4] (see further comments in the conclusion).

The compilation of programs, class, and method declarations is defined in Figure 6. We follow the usual syntactic conventions for logic programs: logical variable names begin with upper case, whereas predicate and functor names begin with lower case letters. Underscore denotes anonymous logical variables that occur only once in a clause; [ ] and $[e|l]$ respectively represent the empty list, and the list where e is the first element, and l is the rest of the list.

$$(\text{prog})\frac{\forall\, i=1..n\ cd_i \rightsquigarrow \mathit{Hf}_i \qquad e \rightsquigarrow (t\,|\,B)}{\overline{cd}^n\ \ e \rightsquigarrow (\mathit{Hf}^d \cup \overline{\mathit{Hf}}^n | B)}$$

$$(\text{class})\frac{\forall\, i=1..k\ md_i\ \textbf{in}\ c_1 \rightsquigarrow \mathit{Hf}_i \quad \mathit{inhFields}(c_1)=\overline{f'}^h}{\begin{array}{l}\texttt{class}\ c_1\ \texttt{extends}\ c_2\ \{\ \overline{f}^n\ \ \overline{md}^k\ \} \rightsquigarrow \overline{\mathit{Hf}}^k \cup \\ \left\{\begin{array}{l} \mathit{class}(c_1) \leftarrow \mathit{true}. \\ \mathit{extends}(c_1, c_2) \leftarrow \mathit{true}. \\ \overline{\mathit{dec\_field}(c_1, f)}^n \leftarrow \mathit{true}. \\ \mathit{new}(CE, c_1, [\overline{T'}^h, \overline{T}^n]) \leftarrow \mathit{new}(CE, c_2, [\overline{T'}^h]), \overline{\mathit{field\_upd}(CE, c_1, f, T)}^n. \end{array}\right\}\end{array}}$$

$$(\text{meth})\frac{\overline{b}^n \rightsquigarrow (t\,|\,B)}{\begin{array}{l} m(\overline{r}^n)\{\overline{b}^n\}\ \textbf{in}\ c \rightsquigarrow \\ \left\{\begin{array}{l} \mathit{dec\_meth}(c, m) \leftarrow \mathit{true}. \\ \mathit{has\_meth}(CE, c, m, [\mathit{This}_0, \overline{r}^n], t) \leftarrow \mathit{subclass}(\mathit{This}_0, c), B. \end{array}\right\}\end{array}}$$

$$(\text{body})\frac{\forall\, i=1..n\ b_i \rightsquigarrow B_i}{\overline{b}^n\ \ l{:}\texttt{return}\ r \rightsquigarrow (r\,|\,\overline{B}^n)}$$

Fig. 6. Compilation of programs, class, and method declarations and bodies.

Each rule defines a different compilation judgment. The judgment $\overline{cd}^n\ e \rightsquigarrow (\mathit{Hf}^d \cup \overline{\mathit{Hf}}^n | B)$ means that the program $\overline{cd}^n\ e$ is compiled into the pair $(\mathit{Hf}^d \cup \overline{\mathit{Hf}}^n | B)$, where $\mathit{Hf}^d \cup \overline{\mathit{Hf}}^n$ is a Horn formula (that is, a set of Horn clauses), and B is a goal[6]. Static analysis of the program corresponds to resolving the goal B w.r.t. the coinductive semantics of $\mathit{Hf}^d \cup \overline{\mathit{Hf}}^n$. The Horn formula Hf^d contains all generated clauses that are invariant w.r.t. the program (see Figure 8), whereas each Hf_i is obtained by compiling the class declaration cd_i (see below); the goal B is obtained from the compilation of the main expression e (see below); the term t corresponding to the returned type of e is simply ignored here, but it is necessary for compiling expressions (see comments on Figure 7).

The compilation of a class declaration $\texttt{class}\ c_1\ \texttt{extends}\ c_2\ \{\ \overline{f}^n\ \ \overline{md}^k\ \}$ is a set of clauses, including each clause Hf_i obtained by compiling the method md_i (see below), clauses asserting that class c_1 declares field f_i, for all $i = 1..n$, and three specific clauses for predicates *class*, *extends*, and *new*. The clause for *new* deserves some explanations: the atom $\mathit{new}(ce, c, [\overline{t}^n])$ succeeds iff the invocation of the implicit constructor of c with n arguments of type $\overline{t}^n$ is type safe in the global class type environment ce. The class environment ce is required for compiling field access and update expressions (see Figure 7): it is a finite map (simply represented by a list) associating class names with field records, which are finite maps (again simply represented by lists) associating all fields of a class with their corresponding types. Class environments are required because of nominal types: abstract compilation with structural types allows data polymorphism on a per-object basis, whereas here we obtain only a very limited form of data polymorphism on a per-class basis. Type safety of object creation is checked by ensuring that object creation for the direct superclass c_2 is correct, where only the first h arguments corresponding to the inherited fields (returned by the auxiliary function *inhFields* whose straightforward definition has been omitted) are passed; then, predicate *field_upd* defined in Figure 8 checks that all remaining n arguments, corresponding to the new fields declared in c_1, have types that are compatible with those specified in the class environment. The clause dealing with the base case for the root class *Object* is defined in Figure 8.

The judgment $m(\overline{r}^n)\{\overline{b}^n\}\ \mathbf{in}\ c \rightsquigarrow \mathit{Hf}$ means that the method declaration $m(\overline{r}^n)\{\overline{b}^n\}$ contained in class c compiles to Horn clauses Hf. Just two clauses are generated per method declaration: the first simply states that method m is declared in class c (and is needed to deal with inherited methods, see Figure 8), whereas the second is obtained by compiling the body of the method. The atom $\mathit{has\_meth}(ce, c, m, [t_0, \overline{t}^n], t)$ succeeds iff in class environment ce method m of class c can be safely invoked on target object of type t_0, with n arguments of type $\overline{t}^n$ and returned value of type t. The predicate *subclass* (defined in Figure 8) ensures that the method can be invoked only on objects that are instances of c or of one of its subclasses. For simplicity we assume that all names (including

[6] For simplicity we use the same meta-variable B to denote conjunctions of atoms (that is, clause bodies), and goals, even though more formally goals are special clauses of the form $\mathit{false} \leftarrow B$.

this) are translated to themselves, even though, in practice appropriate injective renaming should be applied [5].

The compilation of a method body $\overline{b}^n$ $l{:}\texttt{return}\ r$ consists of the type of the returned virtual register r, and the conjunction of all the atoms generated by the compilation of blocks $\overline{b}^n$.

Figure 7 defines abstract compilation for blocks, and expressions.

$$(\text{block})\,\frac{e \rightsquigarrow (t \mid B)}{l{:}e \rightsquigarrow B} \qquad (\text{seq})\,\frac{e_1 \rightsquigarrow (t_1 \mid B_1) \qquad e_2 \rightsquigarrow (t_2 \mid B_2)}{e_1; e_2 \rightsquigarrow (t_2 \mid B_1, B_2)}$$

$$(\text{c-jmp})\,\frac{\begin{array}{l}\text{if } \mathit{var}(r_i''') = \mathit{var}(r)\\ \ \text{then } t_i' = T, t_i'' = F\\ \ \text{else } t_i' = r_i''', t_i'' = r_i''' \quad T, F \text{ fresh}\end{array}}{\begin{array}{c}\texttt{if}\ (r\ \texttt{instanceof}\ c)\ \texttt{with}\ \overline{(r', r'') = \sigma(r''')}^n\ \texttt{jump}\ l_1\ \texttt{else jump}\ l_2\ \texttt{in}\ \rightsquigarrow\\ (\mathit{void} \mid \mathit{inter}(r, c, T), \mathit{diff}(r, c, F), \overline{\mathit{var\_upd}(r', t'), \mathit{var\_upd}(r'', t'')}^n)\end{array}}$$

$$(\text{var-upd})\,\frac{e \rightsquigarrow (t \mid B)}{r = e \rightsquigarrow (t \mid B, \mathit{var\_upd}(r, t))} \qquad (\text{jmp})\,\frac{}{\texttt{jump}\ l \rightsquigarrow (\mathit{void} \mid \mathit{true})}$$

$$(\text{field-upd})\,\frac{e_1 \rightsquigarrow (t_1 \mid B_1) \qquad e_2 \rightsquigarrow (t_2 \mid B_2)}{e_1.f = e_2 \rightsquigarrow (t_2 \mid B_1, B_2, \mathit{field\_upd}(CE, t_1, f, t_2))}$$

$$(\text{phi})\,\frac{}{r = \varphi(\overline{r}^n) \rightsquigarrow (\vee\overline{r}^n \mid \mathit{var\_upd}(r, \vee\overline{r}^n))}$$

$$(\text{new})\,\frac{\forall\, i = 1..n\ e_i \rightsquigarrow (t_i \mid B_i)}{\texttt{new}\ c(\overline{e}^n) \rightsquigarrow (c \mid \overline{B}^n, \mathit{new}(CE, c, [\overline{t}^n]))}$$

$$(\text{field-acc})\,\frac{e \rightsquigarrow (t \mid B) \quad R \text{ fresh}}{e.f \rightsquigarrow (R \mid B, \mathit{field}(CE, t, f, R))}$$

$$(\text{invk})\,\frac{\forall\, i = 0..n\ e_i \rightsquigarrow (t_i \mid B_i) \quad R \text{ fresh}}{e_0.m(\overline{e}^n) \rightsquigarrow (R \mid B_0, \overline{B}^n, \mathit{invoke}(CE, t_0, m, [\overline{t}^n], R))}$$

$$(\text{var})\,\frac{}{r \rightsquigarrow (r \mid \mathit{true})} \qquad (\text{bool})\,\frac{v \in \{\texttt{true}, \texttt{false}\}}{v \rightsquigarrow (\mathit{bool} \mid \mathit{true})}$$

Fig. 7. Compilation of blocks and expressions

Compiling a block $l{:}e$ returns the conjunction of atoms obtained by compiling e; the type t of e is discarded.

The compilation of $e_1; e_2$ returns the type of e_2 and the conjunction of atoms generated from the compilation of e_1 and e_2.

The compilation of an unconditional jump generates the type *void* and the empty conjunction of atoms *true*. A conditional jump has type *void* as well, but a non empty sequence of predicates is generated to deal with the splitting performed by σ-functions; predicates *inter* and *diff* (defined in Figure 9) correspond to intersection T and difference F between the type of r and c, respectively, and predicate *var_upd* (defined in Figure 8) ensures that the type of virtual registers r_i' and r_i'' are compatible with the pairs of types returned by the σ-functions. In case r_i''' refers to the same variable of r the types of such a pair are the computed intersection T and difference F, respectively, otherwise the pair (r_i''', r_i''') is returned (hence, no split is actually performed).

Compilation of assignments to virtual registers and fields yields the conjunction of the atoms generated from the corresponding sub-expressions, together with the atoms that ensure that the assignment is type compatible (with predicates *var_upd* and *field_upd* defined in Figure 8). The returned type is the type of the right-hand side expression.

Compilation of φ-function assignments to virtual registers is just an instantiation of rule (var-upd) where the type of the expression is the union of the types of the virtual registers passed as arguments to φ.

Compilation rules for object creation, field selection, and method invocation follow the same pattern: the type of the expression is a fresh logical variable (except for object creation) corresponding to the type returned by the specific predicate (*new*, *field*, and *invoke* defined in Figure 8). The generated atoms are those obtained from the compilation of the sub-expressions, together with the atom specific of the expression.

Rules (var) and (bool) are straightforward.

Figure 8 and Figure 9 define the set Hf^d used in compilation rule (prog) corresponding to all generated clauses that are invariant w.r.t. the compiled program.

Clauses in Figure 8 deserve some comments for what concerns the subtyping relation (predicate *subtype*); as expected, classes c_1 and c_2 are both subtypes of $c_1 \vee c_2$, but no subclass of c_1 or c_2 is a subtype of $c_1 \vee c_2$, because subclassing is not subtyping, since no rules are imposed on method overriding. Consider for instance the following source code snippet:

```
class Square {
...
equals(s){return this.side==s.side;}
...
}
class ColoredSquare extends Square {
...
equals(cs){return this.side==cs.side&&this.color==cs.color;}
...
}
```

According to our compilation scheme, the expression `s1.equals(s2)` has type `Bool` if `s1` and `s2` have type `Square` and `Square`$\vee$`ColoredSquare`, respectively, but the same expression is not well-typed if `s1` has type `ColoredSquare` (hence, `ColoredSquare`$\not\leq$`Square`), since `s2` may contain an instance of `Square` for which field `color` is not defined. Subtyping is required for defining the predicates *var_upd* and *field_upd* for virtual register and field updates: the type of the source must be a subtype of the type of the destination.

$$
\begin{array}{l}
\mathit{class}(\mathit{object}) \leftarrow \mathit{true}.\\
\mathit{class}(\mathit{bool}) \leftarrow \mathit{true}.\\
\mathit{extends}(\mathit{bool}, \mathit{object}) \leftarrow \mathit{true}.\\
\mathit{subclass}(X, X) \leftarrow \mathit{class}(X).\\
\mathit{subclass}(X, Y) \leftarrow \mathit{extends}(X, Z), \mathit{subclass}(Z, Y).\\
\mathit{subtype}(T, T) \leftarrow \mathit{true}.\\
\mathit{subtype}(T1 \vee T2, T) \leftarrow \mathit{subtype}(T1, T), \mathit{subtype}(T2, T).\\
\mathit{subtype}(T, T1 \vee T2) \leftarrow \mathit{subclass}(T, T1).\\
\mathit{subtype}(T, T1 \vee T2) \leftarrow \mathit{subclass}(T, T2).\\
\mathit{field}(CE, C, F, T) \leftarrow \mathit{has\_field}(C, F), \mathit{class\_fields}(CE, C, R), \mathit{field\_type}(R, F, T).\\
\mathit{field}(CE, T1 \vee T2, F, FT1 \vee FT2) \leftarrow \mathit{field}(CE, T1, F, FT1),\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{field}(CE, T2, F, FT2).\\
\mathit{class\_fields}([C : R|CE], C, R) \leftarrow \mathit{no\_def}(C, CE).\\
\mathit{class\_fields}([C1 : \_|CE], C2, R) \leftarrow \mathit{class\_fields}(CE, C2, R), C1 \neq C2.\\
\mathit{field\_type}([F{:}T|R], F, T) \leftarrow \mathit{no\_def}(F, R).\\
\mathit{field\_type}([F1 : \_|R], F2, T) \leftarrow \mathit{field\_type}(R, F2, T), F1 \neq F2.\\
\mathit{no\_def}(\_, [\,]) \leftarrow \mathit{true}.\\
\mathit{no\_def}(K1, [K2 : \_|Tl]) \leftarrow \mathit{no\_def}(K1, Tl), K1 \neq K2.\\
\mathit{invoke}(CE, C, M, A, RT) \leftarrow \mathit{has\_meth}(CE, C, M, [C|A], RT).\\
\mathit{invoke}(CE, T1 \vee T2, M, A, RT1 \vee RT2) \leftarrow \mathit{invoke}(CE, T1, M, A, RT1),\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{invoke}(CE, T2, M, A, RT2).\\
\mathit{new}(\_, \mathit{object}, [\,]) \leftarrow \mathit{true}.\\
\mathit{has\_field}(C, F) \leftarrow \mathit{dec\_field}(C, F).\\
\mathit{has\_field}(C, F) \leftarrow \mathit{extends}(C, P), \mathit{has\_field}(P, F), \neg \mathit{dec\_field}(C, F).\\
\mathit{has\_meth}(CE, C, M, A, R) \leftarrow \mathit{extends}(C, P), \mathit{has\_meth}(CE, P, M, A, R),\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \neg \mathit{dec\_meth}(C, M).\\
\mathit{var\_upd}(T1, T2) \leftarrow \mathit{subtype}(T2, T1).\\
\mathit{field\_upd}(CE, C, F, T2) \leftarrow \mathit{field}(CE, C, F, T1), \mathit{subtype}(T2, T1).
\end{array}
$$

Fig. 8. Clauses defining the predicates used by the abstract compilation

Predicate *field* looks up the type of a field in the global class environment, and is defined in terms of the auxiliary predicates *has_field*, *class_fields*, *field_type*, and *no_def*. In particular, predicate *has_field* checks that a class has actually a certain field, either declared or inherited. The definitions of *class_fields*, *field_type*, and *no_def* are straightforward (*no_def* ensures that a map does not contain multiple entries for a key), whereas the clause for *has_field* dealing with inherited fields is similar to the corresponding one for *invoke* (see below).

If the target object has a class type c, then the correctness of method invocation is checked with predicate *has_meth* applied to class c and to the same list of arguments where, however, the type c of **this** is added at the beginning. If the target object has a union type, predicate *invoke* checks that method invocation is correct for both types of the union, and then merges the types of the results into a single union type.

Finally, the clause for *has_meth* deals with the inherited methods: if class c does not declare method m, then *has_meth* must hold on the direct superclass of c.

$$\begin{array}{l}
\mathit{inter}(\mathit{C1}, \mathit{C2}, \mathit{C1}) \leftarrow \mathit{subclass}(\mathit{C1}, \mathit{C2}).\\
\mathit{inter}(\mathit{T1} \vee \mathit{T2}, C, \mathit{IT1} \vee \mathit{IT2}) \leftarrow \mathit{inter}(\mathit{T1}, C, \mathit{IT1}), \mathit{inter}(\mathit{T2}, C, \mathit{IT2}).\\
\mathit{inter}(\mathit{T1} \vee \mathit{T2}, C, \mathit{IT1}) \leftarrow \mathit{inter}(\mathit{T1}, C, \mathit{IT1}), \neg \mathit{inter}(\mathit{T2}, C, \_).\\
\mathit{inter}(\mathit{T1} \vee \mathit{T2}, C, \mathit{IT2}) \leftarrow \mathit{inter}(\mathit{T2}, C, \mathit{IT2}), \neg \mathit{inter}(\mathit{T1}, C, \_).\\
\mathit{diff}(\mathit{C1}, \mathit{C2}, \mathit{C1}) \leftarrow \mathit{class}(\mathit{C1}), \neg \mathit{subclass}(\mathit{C1}, \mathit{C2}).\\
\mathit{diff}(\mathit{T1} \vee \mathit{T2}, C, \mathit{IT1} \vee \mathit{IT2}) \leftarrow \mathit{diff}(\mathit{T1}, C, \mathit{IT1}), \mathit{diff}(\mathit{T2}, C, \mathit{IT2}).\\
\mathit{diff}(\mathit{T1} \vee \mathit{T2}, C, \mathit{IT1}) \leftarrow \mathit{diff}(\mathit{T1}, C, \mathit{IT1}), \neg \mathit{diff}(\mathit{T2}, C, \_).\\
\mathit{diff}(\mathit{T1} \vee \mathit{T2}, C, \mathit{IT2}) \leftarrow \mathit{diff}(\mathit{T2}, C, \mathit{IT2}), \neg \mathit{diff}(\mathit{T1}, C, \_).
\end{array}$$

Fig. 9. Clauses defining type intersection and difference

Predicates *inter* and *diff* define type splitting for σ-functions; the asymmetric definition for *inter* is due to the fact that subclass is not subtyping: if r has type c_1, then it means that it contains an object that is an instance of c_1, therefore the condition (r **instanceof** c_2) is false when c_2 is a proper subclass of c_1. Both predicates fail if the returned type is empty, therefore a conditional jump is not considered correct if either branches are dead (that is, not reachable). In practice, it would be better to avoid this kind of failures by introducing an explicit empty type constant to be able to detect dead code without any failure. Such an alternative option does not pose any technical issue, but since it is more verbose (a new clause dealing with the empty type must be added for most predicates) has not been considered here, just for space limitations.

5 Conclusion

We have defined the small step operational semantics of a simple Java-like language in SSI IR, equipped with an **instanceof** operator for runtime typechecks, and shown how precise type analysis of branches guarded by runtime typechecks can be achieved by abstract compilation in the presence of union and nominal types, and by suitably defining two predicates *inter* and *diff* that provide the type theoretic interpretation of σ-functions.

Despite the use of nominal types, the analysis is more precise than that we would get from the type system of a statically typed language as Java and C#; however, using structural types to trace the type of each field object leads to a more precise analysis, as already shown in previous work [5, 6]. Here we have preferred to keep the presentation simpler, but we envisage no problems in extending the compilation scheme and the definition of the predicates *inter* and *diff* to accommodate structural types.

For what concerns future developments, we are planning to extend the abstract compilation scheme proposed here to support subtyping constraints, to make the analysis more precise and efficient; we do not expect major problems in implementating in CHR [10] a constraint solver for subtyping between set of nominal types [1].

The approach proposed here seems promising also for other kinds of conditions occurring often in object-oriented programs; for instance, SSI can enhance static analysis of null pointer references when branches are guarded by conditions of the form `(`r` == `**null**`)`.

References

1. A. Aiken. Introduction to set constraint-based program analysis. *SCP*, 35:79–111, 1999.
2. C. S. Ananian. The static single information form. Technical Report MITLCS-TR-801, MIT, 1999.
3. D. Ancona, M. Ancona, A Cuni, and N. Matsakis. RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *DLS'07*, pages 53–64. ACM, 2007.
4. D. Ancona, A. Corradi, G. Lagorio, and F. Damiani. Abstract compilation of object-oriented languages into coinductive CLP(X): can type inference meet verification? In *FoVeOOS 2010, Revised Selected Papers*, volume 6528 of *LNCS*. Springer Verlag, 2011.
5. D. Ancona and G. Lagorio. Coinductive type systems for object-oriented languages. In *ECOOP 2009*, volume 5653 of *LNCS*, pages 2–26. Springer Verlag, 2009. Best paper prize.
6. D. Ancona and G. Lagorio. Idealized coinductive type systems for imperative object-oriented programs. *RAIRO - Theoretical Informatics and Applications*, 45(1):3–33, 2011.
7. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13:451–490, 1991.
8. D. Das and U. Ramakrishna. A practical and fast iterative algorithm for phi-function computation using DJ graphs. *TOPLAS*, 27(3):426–440, 2005.
9. B. Alpern et. al. The jalapeño virtual machine. *IBM Systems Journal*, 39, 2000.
10. T. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.
11. R. Griesemer and S. Mitrovic. A compiler for the java hotspottm virtual machine. In *The School of Niklaus Wirth, "The Art of Simplicity"*, pages 133–152, 2000.
12. G. Holloway. The machine-SUIF static single assignment library. Technical report, Harvard School of Engineering and Applied Sciences, 2001.
13. D. Novillo. Tree SSA - a new optimization infrastructure for GCC. In *GCC Developers' Summit*, pages 181–193, 2003.
14. L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In *ICALP 2007*, pages 472–483, 2007.
15. L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive logic programming. In *ICLP 2006*, pages 330–345, 2006.
16. J. Singer. Static single information form in machine SUIF. Technical report, University of Cambridge Computer Laboratory, UK, 2004.
17. J. Singer. *Static Program Analysis based on Virtual Register Renaming.* PhD thesis, Christs College, 2005.
18. A. Tavares, F.M. Pereira, M. Bigonha, and R. Bigonha. Efficient SSI conversion. In *SBLP 2010*, 2010.
19. J. Winther. Guarded type promotion (eliminating redundant casts in Java). In *FTfJP 2011.* ACM, 2011.

Modeling and Analyzing the Interaction of C and C++ Strings

Gogul Balakrishnan[1], Naoto Maeda[2], Sriram Sankaranarayanan[3],
Franjo Ivančić[1], Aarti Gupta[1], and Rakesh Pothengil[4]

[1] NEC Laboratories America, Princeton, NJ
[2] NEC Corporation, Kanagawa, Japan
[3] University of Colorado, Boulder, CO
[4] NEC HCL ST, Noida, India

Abstract. Strings are commonly used in a large variety of software. And yet, they are a common source of bugs involving invalid memory accesses arising due to misuses of the string manipulation API. These bugs are often remotely exploitable, leading to severe consequences. Therefore, the static detection of invalid memory accesses due to string manipulations has received much attention, especially for C programs using the standard C library functions. More recently, software development is increasingly being performed in object-oriented languages such as C++ and Java. However, the need to interact with legacy C code and C-based system-level APIs often necessitates the use of a mixed programming paradigm that combines features of high-level object-oriented constructs with calls to standard C library functions. While such programs are commonplace, there has been little research on static analysis of such code. We present heap-aware memory models for C++ programs, with an emphasis on modeling features such as dynamically allocated memory, use of null-terminated buffers as strings, C++ standard template library (STL) classes and interactions between these features. We use standard verification tools such as abstract interpretation and model checking to verify properties over these models to find potential bugs. Our tool can find *several previously unknown bugs* in open-source projects. These bugs are primarily due to the intricate C++ programming model and subtle interactions with legacy C string functions.

1 Introduction

Buffer overflows are common in systems code. They can lead to memory corruption and application crashes. They are particularly dangerous if they can be exploited by malicious users to deny service by crashing a system or escalate privileges remotely. A large number of overflows are present in deployed commercial as well as open-source software [18]. A significant volume of research on buffer overflow prevention has focused on the detection of overflows in C code.

Software development teams have shifted their development from C to object-oriented languages including C++ and Java. The benefits of using an object-oriented language include reusability, better maintainability, encapsulation and the use of inheritance. In particular, C++ is often chosen due to its ability to interact with legacy C-based systems, including system-level C libraries. Thus, development in C++ often necessitates a mixed programming style combining object-oriented constructs with lower-level C code. Whereas a large volume of work on verification has focused on C programs, there has been comparatively little work on the verification of C++ programs. The modeling of objects in the heap is a key component of such verification. In this paper, we present heap-aware static analysis techniques that can verify memory safety of C/C++ programs. Our approach focuses on the modeling of strings in C/C++ and buffer overflow errors due to the interaction and misuse of string manipulation functions.

```
class Object
{
A:  /* Returns object not ref. */
B:  std::string section_name(unsigned int shndx)
C:  { return this->do_section_name(shndx); }
  ...
};

class Relobj : public Object { ... } ;
```

```
1: void Icf::find_identical_sections(
2:   const Input_objects* input_objects,Symbol_table* symtab){
    ...
4:   for (Input_objects::Relobj_iterator p =
             input_objects->relobj_begin();
         p != input_objects->relobj_end(); p++) {
      ... /* (*p) is of type RelObj*  */
6:     const char* section_name=(*p)->section_name(i).c_str();
 /*(*p)->section_name(.) resolved to Object::section_name()*/
7:     if ( !is_section_foldable_candidate(section_name) )
                                       /* invalid use */
    ...
  }
```

Fig. 1. Motivating example from GNU binutils v2.21.

Motivating example. A typical "interaction bug" is shown in Figure 1. The code snippet is taken from the gold project, part of the GNU binutils (binary utilities) package (v2.21). *Gold* is a linker that is more efficient for large C++ programs than the standard GNU linker. For convenience, we have added labels to denote line numbers of interest. Consider the call to `c_str()` in line `6` of the function `find_identical_sections`. The call `(*p)->section_name(i)` creates a *temporary object* (see labels `A-C` in class `Object`). The call to the `c_str()` method thus obtains a pointer to a C string, pointing into the temporary object.

However, the subsequent uses of that string, stored in the variable `section_name`, are invalid. The temporary object (including the pointed to C string) is destroyed immediately following the call to `c_str()`. Under certain conditions the freed memory may be re-used, leading to segmentation fault or memory corruption. Thus, the call to `is_section_foldable_candidate()`, and further uses of the variable not shown here, produce unexpected behavior.

This example shows some typical C++ code. Note that just considering the call to `c_str()` is not enough to find this bug. If `Object::section_name()` (lines `A-C`) had returned a reference, this use of `c_str()` would likely have been legal. Due to the hidden side effects in C++ and the interaction with legacy C APIs, such bugs are easy to commit and hard to find. Furthermore, the bug in binutils had gone unnoticed for at least a year in spite of rigorous testing (the bug was introduced before the release of v2.20, which was officially released in October 2009). It is likely that under normal runtime deployment or during unit testing, the pointer assigned to `section_name` still contains the original string even after it is destroyed. However, under large resource constraints, this bug may manifest itself likely through a segmentation fault upon a later use of `section_name`. Finally, note that a static analysis needs to handle numerous C++ specific issues including STL classes, complex inheritance, and iterators.

Our Approach. Given a program and the properties to check, we use an *abstraction* to model the memory used by arrays, pointers, and strings. The memory model abstraction only tracks the attributes and operations that are relevant to the properties under consideration. We focus on providing precise and scalable memory model for the usage of C and C++ strings. In particular, we address the intricate interplay between C and C++ strings.

Instead of providing a universal memory model, we partition the set of potential bugs into various classes, and use different models for the different classes. Tailoring the memory models to the class of bugs makes the analysis and verification more scalable. For instance, while checking for NULL-pointer dereferences and use-after-free bugs, we use an abstraction that only tracks the status of the pointer, and does not keep track of buffer sizes and string lengths. On the other hand, we use a more precise analysis model that keeps track of allocated memory regions and string lengths for checking buffer overflows.

One particular distinguishing feature of our memory models is that we provide a *unified* framework that addresses correct usage of C-based strings, the C++ STL string class, as well as the interaction between the C++ string class and C strings through conversions from one to the other. Whereas heap aware models for C programs have been well studied [10,19,20,22,26,30], our model handles C++ objects including memory allocation using `new/delete`, the string class in STL and the interaction of these features in C++. To deal with the interaction of C and C++ strings, we introduce a notion of *non-transferable ownership* of a C-string. We utilize this ownership notion to find dangling pointer accesses of C-strings that were obtained through a conversion from a C++ string.

The memory models are weaved into the program under consideration and is then verified using various static analysis and model checking techniques. First, we employ *abstract interpretation* [16] to prove properties using a variety of numerical abstract domains [15,17,28]. The proved properties are eliminated, which enables us to simplify the model of the program. Then, we use a model checker, in particular a bit-accurate SAT-based bounded model checker [7,14], to find proofs and violations for the remaining properties. The model checker outputs concrete witnesses that demonstrate (a) the path taken through the program to produce the violation and (b) concrete values for the program variables.

The major contributions of this paper are as follows:

❒ We present sophisticated, yet scalable, heap-aware memory models for analyzing overflow properties of C and C++ programs that use features including arrays, strings, pointer arithmetic, dynamic allocation, multiple inheritance, exceptions, casting, and standard library usage.

❒ Our approach tackles the interaction of C and C++ strings, thus enabling our tool to discover subtle bugs in the interaction between the different string kinds. We separate the checks into two classes: a pointer-validity-based checking class and a string-length-based checking class. We also introduce a notion of *non-transferable ownership* or *origination* of a C-string for strings obtained through conversion from the C++ string class.

❒ We implemented our models and demonstrate their usefulness on real code, where we found previously unknown bugs in open-source software. To find these bugs, our tool uses abstract interpretation for proving properties and bit-precise model checking for finding concrete witness traces.

2 Preliminaries

We provide an overview of our analysis framework for C, and present a taxonomy of bugs related to the usage of C++ strings. This taxonomy will be used to guide our subsequent modeling of the string class and its interaction with C strings.

2.1 Overview of Analysis Framework

In the past, we have developed a general analysis framework for C programs called F-Soft [26]. It uses both abstract interpretation and bounded model checking to find bugs in the source code under analysis. F-Soft contains a number of "checkers" for various memory safety issues. These include a memory leak checker (MLC), a *pointer validity* checker (PVC) and an *array buffer overflow* checker (ABC). These checkers use different levels of abstraction, and thus, explore different trade-offs between scalability and their ability to reason about intricate pointer accesses. For example, PVC targets bugs such as use-after-free, accesses of a NULL pointer, freeing of a constant string, etc. On the other hand, ABC targets violations that require reasoning about sizes of arrays and strings, and whether strings are null-terminated. To improve scalability of ABC, properties that could be checked using PVC are not considered by ABC. In this paper, we omit discussion of other checkers available in our tool for sake of brevity. These include checkers for the *use of uninitialized memory* (UUM) and an exception analysis (EXC) that computes exceptional control flow paths in C++ programs, for example. The EXC checker can also find *uncaught exception* violations [32]. Ultimately, all checkers generate a model of the program with embedded properties that can be checked by the subsequent analysis engines.

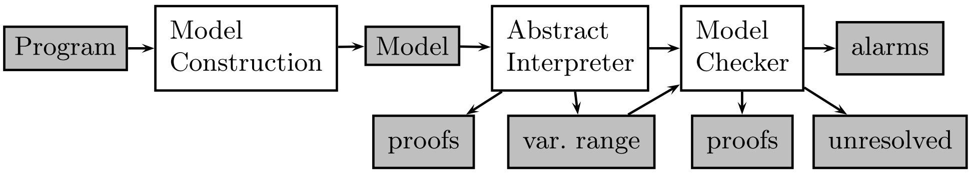


Fig. 2. Main analysis components

Figure 2 depicts the major analysis modules used in our tool. The overall flow is geared towards maximizing the number of property proofs and concrete witnesses of property violations. After model construction for a given program, we analyze the model using abstract interpretation in an attempt to prove that assertions are never violated. Assertions that can be proved safe are removed from the model, and the final model is sliced based on the checks that remain unresolved. In practice, the sliced model is considerably smaller than the original. The model is then analyzed by a series of model-checking engines, including a SAT-based bounded model checker. At the end of model checking, we obtain concrete traces that demonstrate property violations in the model. These violations are mapped back to the source code and displayed using an HTML-based interface or a programming environment such as *Eclipse*(tm). We briefly describe the major components in the flow:

Abstract Interpreter Abstract interpretation [16] is used in our flow as the main *proof engine*. Our abstract interpreter is inter-procedural, flow- and context-sensitive. Currently, we have implementations of abstract domains such as *constants*, *intervals* [15], *octagons* [29], and *polyhedra* [17]. These domains are organized in increasing order of complexity. After each analysis is run, the proved properties are removed and the model is simplified using slicing. The resulting model is then analyzed by a more complex domain.

Model Checker The model checker creates a finite state machine model of the simplified program after abstract interpretation. Each integer variable is treated as a 32 bit entity, character variables as 8 bits and so on. However, the range information provided by the abstract interpreter for program variables is used to reduce the number of bits significantly. We use bit-accurate representations of all operators, ensuring that arithmetic overflows are modeled faithfully.

The model checker verifies the symbolic model for the reachability of the embedded properties. We primarily use SAT-based *bounded model checking* [7]. This technique unrolls the program upto some depth $d > 0$ and searches for the presence of a bug at that depth by compilation into a SAT problem. The depth

d is increased iteratively until a bug is found or resources run out. The model checker generates a counterexample (witness trace) which vastly simplifies the user inspection and evaluation of the error.

2.2 C++ String Class Usage Issues

C++ STL strings provide a safer alternative to developers when compared to C strings. However, as shown in the motivating example (see Figure 1), mistakes are still easy to make, especially in the interaction with C-based standard library functions. The string class contains a number of built-in features such as modification routines (`append, replace`, etc), operations such as substring generation, iterators, and others. Additionally, the methods `c_str()` and `data()` can be called to obtain a buffer containing a C string, which is null-terminated for `c_str()` and not null-terminated for `data()`. Our description focuses on the `c_str()` method, but is applicable to `data()` as well. Moreover, the `data()` method is even more error-prone due to it returning a non null-terminated string. We classify common bugs related to the use of strings below:

(1) Generic bugs: Memory leaks, uncaught exceptions (eg., `std::bad_alloc`) [32].
(2) String class manipulation errors:
  (a) Out of bounds access. `std::out_of_range` exception thrown by the `at` and `operator[]` methods of the `string` class.
  (b) Use of a string object after it has been destroyed.
  (c) Use of a stale string iterator.
(3) Interaction between C and C++ strings
  (a) Access of C-string returned by `string::c_str()`, after the corresponding C++ object is destroyed.
  (b) Certain C library functions called on strings obtained through `c_str()`.
  (c) Manipulation of a C-string returned by `string::c_str()`.
  (d) C-based buffer overflows on C-string obtained through `string::c_str()`.

3 Program Modeling and Memory Checkers

We now discuss the memory models used in our approach. Our approach supports a hierarchy of memory models ranging from models that simply track few bits of allocation status for each pointer to the full-fledged tracking of allocated bounds, string sizes, region aliasing of arrays, and so on. We describe two models within this spectrum: *the pointer validity model* that uses simple pointer type states, and *the pointer bounds model* that attempts to track allocated bounds, positions of various sentinels, and contents of cells accurately.

3.1 Pointer Validity Model

The validity model instruments for each pointer a validity status $\mathsf{ptrVal}(p)$ to denote the type of the location pointed-to in memory. These values include null

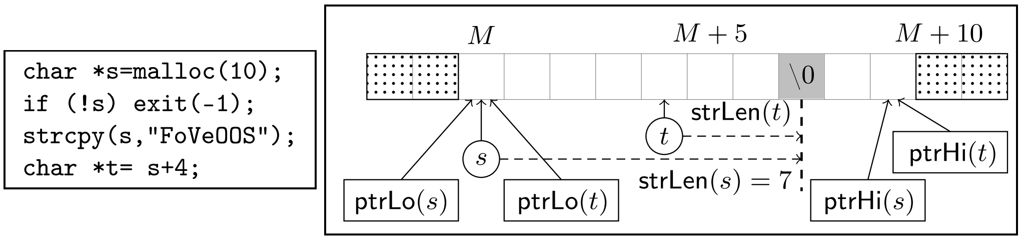
indicating a null pointer; invalid for a non-NULL pointer whose dereference may cause a segmentation violation; static for pointers to global variables, arrays and static variables; stack for pointers to local variables, `alloca` calls, local arrays, formal arguments; heap for pointers to dynamically allocated memory on the heap; and code for code sections, such as string constants.

The validity model does not track addresses of pointers. It also ignores address arithmetic. A pointer expression $p+i$ has the same validity status as its base pointer p. A dereference $*p$ yields an assertion check that is violated if ptrVal(p) is null or invalid. Similarly, relevant checks are done for other operations. We distinguish between null and invalid in order to allow `delete NULL`, which is allowed per C++ standard, as well as optionally allow `free(NULL)`, which is handled gracefully by standard compilers such as `gcc`. Finally, note that it is easy to extend this model to find invalid de-allocations, such as the case where memory that was allocated using `new` is released using `free`. This can be accomplished by separating the validity status heap into sub-regions according to their allocation method, such as heap − malloc, heap − new and heap − new[].

3.2 Pointer Bounds Model

The bounds model tracks various attributes for each pointer, including allocation sizes and sentinel positions, which subsumes information tracked by the validity model. For a pointer p, the main modeling attributes are as follows (see Figure 3):

1. ptrLo(p), which corresponds to the base address of a memory region that p currently points to;
2. ptrHi(p), which corresponds to the last address in the memory region currently pointed to by p that can be accessed without causing a buffer overflow;
3. strLen(p) which corresponds to the remaining string length of the pointer p, which is the distance to the next null-termination symbol starting at p.



```
char *s=malloc(10);
if (!s) exit(-1);
strcpy(s,"FoVeOOS");
char *t= s+4;
```

Fig. 3. The memory model for the pointer bounds model after successfully executing the four statements on the left-hand side: The successful allocation returns a pointer to some new address M, and the lower bound addresses ptrLo(s) = ptrLo(t) = M. The higher bound addresses are ptrHi(s) = ptrHi(t) = $M+9$. Finally, the string lengths are determined using the size abstraction, namely strLen(s) = 7 and strLen(t) = 3. The dotted memory region denotes out-of-bound memory regions for the pointers s and t.

For each pointer p, we track its "address", and its bounds $[\mathsf{ptrLo}(p), \mathsf{ptrHi}(p)]$, representing the range of values of the pointer p such that p may be legally dereferenced in our model. If $p \in [\mathsf{ptrLo}(p), \mathsf{ptrHi}(p)]$ then $p[i]$ *underflows* iff $p + i < \mathsf{ptrLo}(p)$. Similarly, $p[i]$ *overflows* iff $p + i > \mathsf{ptrHi}(p)$.

Dynamic Allocation We assign bounds for dynamic allocations with the help of a special counter $\mathsf{pos}(L)$ for each allocation site L in the code. It keeps track of the maximum address currently allocated. Upon each call to a function such as `p := malloc(n)`, our model assigns the variable $\mathsf{pos}(L)$ to p and $\mathsf{ptrLo}(p)$. It increments $\mathsf{pos}(L)$ by n, and sets $\mathsf{ptrHi}(p) + 1$ to this value.

C String Modeling Conventionally, strings in C are represented as an array of characters followed by a special null-termination symbol. String library functions such as strcat, and strcpy rely on their inputs to be properly null-terminated and the allocated bounds to be large enough to contain the results. We extend our model to check for such buffer overflows using a size abstraction along the lines of CSSV [22]. The major differences are described in Section 6.

For each character pointer p, we use an attribute $\mathsf{strLen}(p)$ to track the position of the first null-terminator character starting from p. The updates to string length can be derived along similar lines as those for the pointer bounds. For instance, calls to the method `strcat` that append its second argument to the first lead to assertion checks in terms of the pointer bounds and string lengths that guarantee its safe execution. Next, the update to the strLen attribute of the first argument is instrumented. Our approach currently has instrumentation support for about 650 standard library functions. It provides support for parsing constant format strings in order to model effects of functions such as `sprintf`. We elide the details for lack of space and focus here on the modeling of C++ strings and their interaction with C.

4 Modeling the STL String Class

We now present a model for C++ strings that allows us to capture common bugs arising from the misuse of STL strings. Note that, for the sake of brevity, we omit the presentation of string iterator related issues in this paper. Furthermore, we will not discuss issues due to uncaught exceptions when utilizing the C++ string class. Details on our exception handling can be found in [32].

As in Section 3, we separate verification into a light-weight pointer validity-based checker and a more heavy-weight buffer overflow checker tracking accurate string lengths using an extension of the pointer bounds model. Finally, it should be noted that we model a wider class of C++ STL strings than alluded to so far. For example, we also model the templated class `std::basic_string<T>`, of which `std::string` is just a particular instantiation.

4.1 Pointer and String Object Validity

Section 3.1 introduced a memory model that focusses on validity of pointers. Here, we extend it by introducing a new validity status that is used to model

status	*	`free`	`delete`	`delete[]`	`if-NULL`	`return`	˜()
null	✠	(✠?) null	null	null	null	null	✠
invalid	✠	✠	✠	✠	✠	invalid	✠
stack	stack	✠	✠	✠	N/A	invalid	invalid
global	global	✠	✠	✠	N/A	global	✠
code	✠on write	✠	✠	✠	N/A	code	✠
env.	env.	invalid	invalid	invalid	null	env.	invalid
heap-malloc	heap-malloc	invalid	✠	✠	N/A	heap-malloc	✠
heap-new	heap-new	✠	invalid	✠	N/A	heap-new	✠
heap-new[]	heap-new[]	✠	✠	invalid	N/A	heap-new[]	✠
ownerM.	✠on write	✠	✠	✠	N/A	ownerM.	✠

Table 1. Overview of pointer and string object validity model. This table shows the effect of operations on different validity statuses. A potential error is marked using the symbol ✠. Upon error, the validity status changes to invalid. If the update is safe, the table provides the resulting status after the client code operation. The entry `N/A` denotes that a particular step is not possible in our model. Operation "*" denotes a pointer or object read/write, "`return`" denotes the end of a functional scope, "`if-NULL`" denotes a pointer equals null check, "˜()" denotes a destructor call. Allocation (`malloc, new`), initialization operations (constructor calls), and other details are omitted for brevity.

the interaction of C++ strings with C-based strings. We check most issues related to the interaction of C++ and C strings by developing an extended pointer and string object validity checker rather than additionally burdening the pointer bounds model. To do so, we model calls to `string::c_str()` such that they return C strings whose validity status is set to a new status that behaves roughly like the code status, denoting constant strings. A key difference is that the *owning class instance*, which returned the string in the first place, is allowed to manipulate this string, while no manipulations are permissible for constant strings.

This naturally leads to a notion of *ownership* [9,12] of pointers that is a common programming idiom. Thus, we introduce a new status ownerMutable. Prior work used transferable ownership models to find memory leaks in C++ code [23]. However, we only consider C-strings obtained from C++-strings. Thus, we limit ourselves to a *non-transferable ownership* model, which tracks the relationship between originating C++-string and owned C-string. This allows us to declare such ownerMutable strings as stale (that is, invalid), when the originating C++ object that owns it is modified using a method call.

We summarize the pointer and string object validity checker in Table 1. It shows the effect of various operations in the client code on the defined validity statuses. The handling of many operations including initialization, allocation, destructor calls and so on are omitted from the table in order to avoid clutter.

Figure 4 shows a partial sketch of our custom string object validity model. The internal assertion checks are represented as calls to a member function `isValid(operation)`, which can be thought of as utilizing the information in

```
class string { /* pointer and string object validity model */
private: char *p ;
public: ...
  string() {
    p = new char[1];                  /* assumed to not fail */
  }
  string(const string &s) {
    ASSERT(s.isValid(READ-OP));
    p = new char[1];                  /* assumed to not fail */
  }
  ~string() {
    ASSERT(this.isValid(DESTRUCT));
    delete[] p;
  }
  string substr(size_t p=0,size_t n=MAX) const{
    ASSERT(this.isValid(READ-OP);
    return string() ;
  }
  void push_back(char c) {
    ASSERT(this.isValid(WRITE-OP));
    delete[] p ;      /* used to invalidate stale pointers */
    p = new char[1] ;                 /* assumed to not fail */
  }
  const char *c_str() const {
    ASSERT(this.isValid(READ-OP));
    setValid(p,OWNER_MUTABLE);
    return (const char *) p;
  }
};
```

Fig. 4. Partial string object validity model sketch

Table 1. The sketch shows the use of a `setValid(void*,status)` method that can be thought of as setting the validity status for arbitrary pointers. The non-const function `push_back(c)` shows how we invalidate C-strings that may have been obtained through `c_str()` earlier. Finally, note that we separate the issue of allocation failures through `new` from the string validity checking. As mentioned in the comments, we assume that each `new` operation succeeds.

Example 1. Figure 5 shows a simple C++ function that manipulates a C++ string and converts it to a C string. It proceeds to call `strlen` on this C string. A variety of intermediate transformations are performed on the C++ source code including transformations that make calls to constructors and destructors explicit. Figure 5 also shows the result of this transformation for method `cutLen`, which we call `cutLenX`. Note the use of a temporary variable as a result of our transformation, which is initialized using the copy-constructor, and then

```
// Simple C++ string use
int cutLen(const string &s,size_t i,size_t n){
  const char *str = s.substr(i,n).c_str();
  return strlen(str);
}
```

```
// Simplified representation of cutLen
int cutLenX(const string &s,size_t i,size_t n){
  const string tmp = string(s.substr(i,n)) ;
  const char *str = tmp.c_str();
  tmp.~string() ; //also invalidates str!
  return strlen(str);
}
```

Fig. 5. A simple example illustrating the interaction of C and C++ strings.

stmt	**&s**	**&tmp**	**str**
substr(.,.)	stack	stack	—
c_str()	stack	stack	ownerM.
tmp.~string()	stack	invalid	invalid
strlen(.)	stack	invalid	✠

(a)

stmt	**&s**	**s.p**	**str**
initially	stack	heap-new[]	—
str=s.c_str()	stack	ownerM.	ownerM.
s.pushback('a')	stack	heap-new[]	invalid
strlen(str)	stack	heap-new[]	✠

(b)

Fig. 6. (a) Updates to the validity status for the simplified code shown in Figure 5, assuming that the input string `s` was initially allocated on the stack. The destruction of the temporary object `tmp.~string()` also invalidates the pointer `str` through aliasing. (b) Updates to the validity status for another sequence of statements shown in the column labeled stmt. The `pushback` operation first passes the required assertion, then invalidates the pointer `str`, and finally resets the internal pointer `s.p` to a fresh allocated region. The subsequent call to `strlen(str)` thus raises an error.

destroyed using an explicit call to `~string()`. The bug in the code can thus be detected using the model of Figure 4 (see Figure 6 (a)).

4.2 Pointer and string bounds model

The array bounds model for C strings is extended by tracking the logical size of each C++ string. This size is used to handle calls to `string::c_str()` and `string::data()`. Therein, we create valid C strings of the appropriate string length and allocation size, and null-termination status.

Figure 7 shows a simplified model for the `c_str()` method. Note that we do not check whether the string object is valid during calls to `c_str()` in this checker. These checks are already performed in the pointer validity model. Similarly, we do not worry that this model leaks memory for calls to `c_str()` since it is only used for ABC. It should be highlighted that due to the use of the efficient validity checker, we can simplify the model for the array bound checking model

```
class string {                /* array bound model */
private: size_t size ;
public: ...
  const char *c_str const {
    char *res=new[size+1];      /* should not fail */
    strLen(res)=size;     /* thus null-terminated */
    return (const char *) res;
  }
};
```

Fig. 7. Array bound model for the `string::c_str` method.

to only consider the size abstraction. Issues that are related to failed allocations are, as mentioned before, relegated to the special purpose exception checker.

5 Experiments

We have implemented our methods in an in-house extension of CIL [31] called `CILpp`, which handles C++ programs. We present a number of experiments on some C and C++ benchmarks, and describe some of the previously unknown bugs in C++ programs discovered by our analysis.

The models described thus far are able to find a wide variety of memory related issues in C/C++ source code. Since the focus of this paper is on the modeling of the interaction of C and C++ strings, we first present experiments that target only this particular aspect. To do so, we have performed experiments on open-source software packages that contain such interactions. Our analysis is performed in a *scope-bounded fashion* [5,25,34]. A simple pre-processing technique is used to identify potential error sites. For the interaction analysis, these are centered around calls to string library functions and error-prone functions such as calls to the `string::c_str()` method. This enables us to choose a set of objects and methods to be analyzed. We present a number of bugs that have been uncovered by our experiments, thus far. As our tool is being improved, we are applying our techniques to more open-source software.

Motivating example Recall the code fragment presented as Figure 1 in Section 1. The released version of the GNU binutils package at the time of the experiments was v2.21 (official releases are available at `ftp.gnu.org/gnu/binutils`), which was released in December 2010. The bug described earlier was already present in v2.20 released in October 2009. Our tool discovered the bug in March 2011. The developers of the gold package confirmed this bug. However, the developers have been aware of this bug internally about a month before our report. A fix for this bug was finally released with v2.21.1 in June 2011.

Stale uses of `c_str`-created C-strings In our experiments, we found that the issue of dangling pointer accesses due to stale uses of C++-to-C converted strings is the main bug category of interest. We have found many incarnations

```
void IO::FixSlashes(char *str) {
  for (uint8 i=0; i<=strlen(str); i++) {
    if ((str[i]=='\\' || str[i]=='/') &&
        str[i+1]=='\0') {
      str[i] = '\0' ;    /* invalid write */
      return ;
    }
    if (str[i] == '\0') return ;
  }
}

void IO::FixPatches() {
  ...
  FixSlashes((char *)cfg->mysqlpath.c_str());
  FixSlashes((char *)cfg->wowpath.c_str());
}
```

Fig. 8. Invalid string manipulation

of this bug pattern in addition to the motivating example, which can be found using the validity-based abstraction model.

We have observed the same issue in a variety of other open-source benchmarks, including in unit tests for *ICU4C* (see icu-project.org/apiref/icu4c/), which provides portable unicode handling capabilities for software globalization requirements. Similarly, we noticed three uses of a dangling C-string pointer obtained through `string::c_str()` in *Mosh*, a fast interpreter for Scheme as specified in R6RS, which is the latest revision of the Scheme standard. After we informed the developers of this actively maintained project about these three dangling pointer violations, they have confirmed the issue and have fixed them in the source repository (see `http://bit.ly/gCdwva`).

Manipulation of `ownerMutable` strings We also observed rare cases of direct string manipulation of C-strings obtained through `c_str()`. As discussed earlier, this is in explicit violation of the STL C++ string specification. Multiple such scenarios occurred in the *datatrap* project, one of which is shown in Figure 8.

Buffer overflows due to string conversions In our experiments, we have also observed rare cases of potential buffer overflows using strings obtained from a C++ string object. One such example is shown in Figure 9, which is from a library that transliterates text between different representations. Note that this warning awaits confirmation, since in our scope-bounded analysis we are not aware of any global constraint on the maximum size of a string to be converted.

Erlang/OTP Case Study Erlang (see erlang.org) is a programming language used to build massively scalable soft real-time systems with requirements on high availability. Erlang's runtime system has built-in support for concurrency, distribution and fault tolerance. OTP is a set of Erlang libraries pro-

```
char *convert(const char *in, mode_t mode,
              parse_func_t parse, output_func_t output) {
  static char buf[4096];
  ...
  std::ostringstream sout;
  (*output)(tokens,sout,mode);
  sout << '\0' ;
  std::strcpy(buf,sout.str().c_str()); /* buffer overflow? */
  return buf;
}
```

Fig. 9. Potential buffer overflow

viding middle-ware to develop such systems. It includes a distributed database, applications to interface towards other languages, debugging tools, etc.

We analyzed relevant C and C++ source code of the current Erlang/OTP release R14B01 (December 2010). The *Orber* application is a CORBA compliant Object Request Broker (ORB), which provides CORBA functionality in an Erlang environment. Essentially, the ORB channels communication or transactions between nodes in a heterogeneous environment.

```
typedef std::stringstream STRINGSTREAM;
typedef std::stringbuf STRINGBUF;
void InitialReference::createIOR(
    STRINGSTREAM& byte, long length) {
  STRINGBUF *stringbuf;
  STRINGSTREAM string;
  int i;
  const char *c;
  const char *bytestr = byte.str().c_str();
  for(i = 0,c = bytestr; i<length; c++, i++){
    b = *c;        /* invalid access */
      ...
  }
  delete bytestr ; /* invalid call to delete */
  /* iorString is a member field */
  iorString = (char *)string.str().c_str();
}
```

Fig. 10. Erlang/OTP Orber application code

Figure 10 shows a part of the C++ source code for the `InitialReference` class in Orber. The code generates a reference for an *Interoperable Object Reference (IOR)*, which simplifies the initial reference access from C++. However, the

C++ interface contains a number of invalid uses of C-strings from a C++ string object in the central `createIOR` method. Our analysis discovered the invalid access inside the `for`-loop, and also reported the invalid call to `delete`.

We analyzed the complete C++ code inside the Orber module. As is typical for C++, complexity of the analysis is increased due to standard header files. While the Orber module only contained about 300 LOC, the effective LOC after including the relevant headers is about $3k$ LOC. Our tool analyzed 7 functions of interest, and reported only the above 2 witnesses using the pointer and string object validity checker. The array bound checker did not find any witnesses in this case study. For one of the functions, the analysis using abstract interpretation and bounded model checking timed out (we limit the analysis for each function to 10 minutes). Overall, for 14 function and checker pairs, our tool reported over 40 property proofs, and spent about 20 minutes for the analysis.

However, our tool did not report a third issue, where a dangling pointer is assigned to the `iorString` member field. We discovered this issue when inspecting neighboring code to reported warnings. This likely violation of the object invariant, that all member fields be pointing to valid memory regions, was not discovered since our scope-bounded analysis did not find a read of the `iorString` field. In the future, we would like to extend our analysis to automatically check for object consistency after method invocations, in order to discover such issues.

The c-icap project The *c-icap* project is an open-source implementation of ICAP (Internet Content Adaptation Protocol), a protocol aimed at supporting HTTP content adaptation. ICAP allows arbitrary content-filtering and on-the-fly content modification. A common application running ICAP are anti-virus scanners, for example. The development of the *c-icap* project started in 2004, and the project is still actively maintained (see c-icap.sourceforge.net).

Bug category	**Checker**	**Reported**	**Known**	**Fixed**	**Important**
NULL access	PVC	23	0	22	1
Memory leak	MLC	7	1	6	0
Uninitialized condition	UUM	2	0	2	1
Array underflow	ABC	1	0	1	0
Partially initialized memory	UUM	1	0	1	1
Total		34	1	32	3

Table 2. Experimental results for *c-icap* for various checkers (see Section 2.1)

We analyzed the complete *c-icap* project with our tool, by analyzing individual modules separately. The tool analyzed over $24k$ lines of source code written in C, which includes about $4k$ lines of header files. The complete analysis, in a scope-bounded fashion, using abstract interpretation and model checking for all checkers completes in a few hours. The full investigation of all witnesses found by the model checker took one expert user about 3 hours.

The experimental results are summarized in Table 2. The investigation yielded 34 unique bugs that were communicated to the developer of *c-icap*. 32 of the 34 reported issues have been fixed so far. Three of the reported bugs were deemed very important by the developer, including one deep inter-procedural NULL access. The two bugs that have not been fixed yet have been acknowledged as bugs as well, and are to be addressed in future releases. Further details are available at www.nec-labs.com/~ivancic/bugs/c-icap.htm.

The MeCab project The *MeCab* project provides a customizable Japanese morphological analyzer, which is applied to a variety of natural language processing tasks. Its source code (without any header files) contains 6.6k LOC of C++ code. A verification engineer discovered four bugs using this approach. This includes 3 paths with invalid NULL accesses, which were found using the pointer validity checker. Additionally, one uninitialized memory read was discovered.

6 Related Work

Buffer overflows can cause memory corruption which may be hard to detect instantly. Cowan et al. survey different buffer overflow attacks and some attempts at prevention and detection [18]. Static approaches use pointer analysis, range analysis and constraint solvers at various degrees of precision. Wagner et al. transform the overflow check elimination problem into one of solving interval constraints over integers [36]. Rugina and Rinard provide a powerful summary-based approach that reduces interval analysis problems into linear programming [33]. Many of the early approaches do not completely handle complications involving dynamic memory allocation, heap data-structures, array contents, type-casting, etc. Recently, there has been work on more comprehensive approaches, that handle many of the complications mentioned above [4,11,22,35]. However, we are not aware of any prior work on addressing buffer overflows due to the interaction of C++ and C string usage. Recently, size-based abstractions for strings have been proposed for other languages, such as PHP, as well [37].

The CSSV tool [22] implements a comprehensive approach to overflow detection of C code. It constructs a memory model that tracks pointer bounds, and string lengths of arrays. A precise region-based points-to analysis handles overlaps between strings. Our memory model is fundamentally similar to that of CSSV. By combining abstract interpretation with SAT-based model checking in a scope-bounded fashion [25], we obtain scalable analysis for programs that are much larger than those reported by Dor et al.

Our approach uses the theory of abstract interpretation [16] along with numerical domains such as Intervals [15], Octagons [28], Polyhedra [17] and other numerical domains of intermediate precision and complexity. Abstract interpretation has been used in tools such as *PolySpace* [3], *Astrée* [8], and so on. These tools focus on checking embedded applications with special features such as simple aliasing, no dynamic allocation, simple control flow and no recursion. However, our approach is designed to be more general purpose. The CoVerity verifier [1] has also been successfully applied to large industrial and open-source projects. From published reports, most uncovered defects pertain to static buffers and are intraprocedural. Our effort is more ambitious in nature; we focus on accurate memory modeling to detect more complex bugs. CodeSonar from GrammaTech [2] is another related commercial tool. Recently, the static analysis of STL container classes was proposed [21]. However, we are not aware of any tool that directly targets the interaction of C and C++ strings.

There have been past approaches to model check programs for buffer overflows using various model checking techniques. The CBMC tool due to Clarke et al. [14] uses SAT-based *bounded model checking* (BMC) to unroll a given program upto a fixed depth into a SAT problem, which is checked for the presence of a violation upto that depth [7]. Our tool uses SAT-based BMC at its backend. However, we also use abstract interpretation up front to vastly simplify the model and obtain a more scalable approach. Predicate abstraction using automatic counterexample-guided abstraction refinement (CEGAR) [13,27] has lead to important tools such as SLAM [6], BLAST [24], and many others. These tools have been mainly used to find API usage violations. However, our own experience with predicate abstraction refinement suggests that for properties such as buffer overflows and strings, the automatic refinement leads to a large number of predicates and too many refinement iterations.

References

1. Coverity Inc. program verifier. `www.coverity.com`.
2. GrammaTech CodeSonar. `www.grammatech.com/products/codesonar`.
3. PolySpace program analysis tool. `www.polyspace.com`.
4. X. Allamigeon, W. Godard, and C. Hymans. Static Analysis of String Manipulations in Critical Embedded C Programs. In *SAS*. Springer, 2006.
5. D. Babić and A. J. Hu. Structural abstraction of software verification conditions. In *CAV*, volume 4590 of *LNCS*, pages 366–378. Springer, 2007.
6. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *PLDI'01*, pages 203–213. ACM Press, 2001.
7. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, volume 1579 of *LNCS*, pages 193–207, 1999.
8. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, volume 548030, pages 196–207. ACM Press, June 2003.
9. C. Boyapati, R. Lee, and M. C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.
10. S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A reachability predicate for analyzing low-level software. In *TACAS*, LNCS. Springer, 2007.
11. A. Christensen, A. Møller, and M. Schwartzbach. Precise analysis of string expressions. In *SAS 2003*, volume 2694 of *LNCS*, pages 1–18. Springer, 2003.
12. D. G. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.
13. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.
14. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *LNCS*. Springer, 2004.

15. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In 2^{nd} *Intl. Symp. on Programming*, pages 106–130. Dunod, France, 1976.
16. P. Cousot and R. Cousot. Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.
17. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among the variables of a program. In *POPL*, pages 84–97. ACM, Jan. 1978.
18. C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proc. DARPA Information Survivability Conference and Expo (DISCEX)*. IEEE, 1999.
19. M. Das. Unleashing the power of static analysis. In *SAS*, volume 4134 of *LNCS*, pages 1–2. Springer, 2006.
20. M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI*, pages 57–68. ACM Press, 2002.
21. I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. In *POPL*, pages 187–200. ACM, 2011.
22. N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proc. PLDI.* ACM Press, 2003.
23. D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI*, pages 168–181. ACM, 2003.
24. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70. ACM, 2002.
25. F. Ivančić, G. Balakrishnan, A. Gupta, S. Sankaranarayanan, N. Maeda, H. Tokuoka, T. Imoto, and Y. Miyazaki. DC2: A framework for scalable, scope-bounded software verification. In *ASE*, 2011.
26. F. Ivančić, I. Shlyakhter, A. Gupta, M. Ganai, V. Kahlon, C. Wang, and Z. Yang. Model checking C programs using F-Soft. In *ICCD*, pages 297 – 308. IEEE, 2005.
27. R. Kurshan. *Computer-aided Verification of Coordinating Processes: the automata-theoretic approach.* Princeton University Press, 1994.
28. A. Miné. A new numerical abstract domain based on difference-bound matrices. In *PADO II*, volume 2053 of *LNCS*, pages 155–172. Springer, May 2001.
29. A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.
30. Y. Moy. *Automatic Modular Static Safety Checking for C Programs.* PhD thesis, Université Paris-Sud, Jan. 2009.
31. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of C programs. In *CC*, 2002.
32. P. Prabhu, N. Maeda, G. Balakrishnan, F. Ivančić, and A. Gupta. Interprocedural exception analysis for C++. In *ECOOP*. Springer, 2011.
33. R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *PLDI*, pages 182–195. ACM, 2000.
34. D. Shao, S. Khurshid, and D. E. Perry. An incremental approach to scope-bounded checking using a lightweight formal method. In *FM*, pages 757–772. Springer, 2009.
35. A. Simon and A. King. Analyzing String Buffers in C. In *Proc. AMAST'02*, volume 2422 of *LNCS*, pages 365–379. Springer, September 2002.
36. D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proc. Network and Distributed Systems Security Conference*, pages 3–17. ACM Press, 2000.
37. F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for PHP. In *TACAS*, volume 6015 of *LNCS*, pages 154–157. Springer, 2010.

Integration of Bounded Model Checking and Deductive Verification

Bernhard Beckert, Thorsten Bormer, Florian Merz, and Carsten Sinz

Department of Informatics,
Karlsruhe Institute of Technology, Germany
http://{formal|verialg}.iti.kit.edu

Abstract. Modular deductive verification of software systems is a complex task: the user has to put a lot of effort in writing module specifications that fit together when verifying the system as a whole. In practice, this involves several iterations of writing and correcting auxiliary specifications, as well as guiding the verification system in finding a correctness proof. In particular at the beginning of this process, when not many auxiliary specifications have been written yet, it is hard to get sensible feedback from the verification tool.

In this paper, we propose a combination of deductive verification and software bounded model checking (SBMC), where SBMC is used to support the user in the specification and verification process, while deductive verification provides the final correctness proof. SMBC provides early – as well as precise – feedback to the user. Unlike modular deductive verification, the SBMC approach is able to check annotations beyond the boundaries of a single module – even if other relevant modules are not annotated (yet). This allows to test whether the different module specifications in the system match the implementation at every step of the specification process.

1 Introduction

Deduction-based methods for software verification and systematic debugging have seen a tremendous progress in recent years. Up to the nineties, the main focus in these areas was on fundamental research – the methods developed during this period were applicable only to small, academic software systems. Since the beginning of this century, a set of new techniques emerged that puts into reach the application of these techniques to real-world software systems.

What is needed now are methods and tools to help the user in writing modular specifications for complex systems and support the verification process. The size of real-world systems, together with a possibly huge amount of interdependencies of functions and data structures in the system make them hard to specify – even if they are decomposed into small modules. Modular verification is a commonly used technique to help the user in verifying large systems. It is supported (or even required) in most current deductive verification tools. Despite this, tracking interdependencies between a large amount of modules still seems to be beyond the compass of the human mind.

The disadvantage of verifying each module in isolation is that the specification of a module may not fit the other parts of the system where the module is used. This may result in many iterations of changes to the specification of all modules until a fix-point is reached that allows verification of the full system.

On the other hand, approaches like software bounded model checking are able to analyze a system beyond module boundaries, providing a more global view of the system at hand. This ability, however, does not come for free: The specification languages of BMC tools are not as expressive, and they have less precision as compared to deductive verification systems.

In this paper, we propose a combination of deductive verification and software bounded model checking (SMBC), where SBMC is used to support the user in the specification and verification process while deductive verification provides the final correctness proof. For this, auxiliary specifications that the user adds as annotations to the system are translated into input for the SBMC tool. As the user's annotations are aimed at deductive verification, the SBMC tool may not be able handle them, but in many cases it can provide early feedback. In particular, SBMC can check the appropriateness of the specifications beyond the boundaries of a single module – even if other relevant modules are not specified (yet). This allows to test early on whether the different module specifications in the system match the implementation at every step of the specification process.

In Section 2, we give a brief introduction to deductive verification. This is followed by a description of the problems one encounters when deductively verifying software systems in practice (Section 3). Section 4 contains a short explanation of software bounded model checking and its relation to deductive verification. Section 5 presents the main contribution of this work, namely the integration of annotation-based deductive verification and software bounded model checking. This is followed by a brief description of how to translate from the annotation-based specification used in deductive verification into input of the BMC framework in Section 6. Subsequently, in Section 7 an extended example is given describing the advantages of this combination. Finally, Sections 8 and 9 are concerned with related work, conclusion, and future work.

As the basis for the work presented in this paper and for the experiments, we used the deductive verification tool VCC [4] and the software bounded model checker LLBMC [14]. Accordingly, the verification targets we use are C programs, and annotations are written in VCC's specification language. Nevertheless, the ideas presented here should apply equally well to other annotation-based deductive verification tools and languages.

2 Basics of Deductive Verification

2.1 The Annotation-based Verification Paradigm

Annotation-based deductive verification allows to obtain a rigorous mathematical proof for the correctness of a software system w.r.t. its formal functional specification. Verification tools that fall into this category are based on a specific style of user interaction, called *auto-active* [9]. Proof construction is not interactive, but all information needed for finding a correctness proof, including all auxiliary specifications, have to be provided by the user before the verification tool is run – there is no provision for manual intervention during the proof construction process.

Specifications are written directly in the source code as annotations in a way that does not alter the normal compilation process nor execution of the software system. Often, these annotations feature a syntax that is close to the syntax of expressions of the target programming language, enriched by constructs of first-order logic.

The auto-active paradigm has several implications:

- specification and verification has to be modular in order for the back-end prover to perform well, and
- proof guidance, as well as auxiliary specifications (like loop invariants) have to be provided by the user as annotations.

A typical example for an annotation-based verification system is VCC [4], developed by Microsoft Research. VCC is a deductive verification tool for concurrent C programs. It uses the Boogie tool [1] to generate verification conditions in first-oder logic. The generated FOL formula has the property that it is unsatisfiable iff the program fulfills its specification. An automated theorem prover, in this case Z3 [5], is then used to show unsatisfiability of the formula. If the formula is satisfiable, Z3 can often find models for it, which can be translated back to traces of the program that violate the specification [11]. As said above, VCC was used as the basis for the work presented in this paper.

2.2 Verification Targets

In the following, we consider the verification targets to be C programs, containing a set of function definitions. We consider these functions to be the modules of the program. Our approach is not restricted to procedural programming languages like C, though. In particular, it can be applied to verify programs written in object-oriented programming languages like Java or C++. Abstract types (interfaces), e.g., can be dealt with by providing suitable contracts for these interfaces or by giving a set of concrete instantiations. Dynamic typing can be taken into account by replacing method calls by case discrimination over the possible dynamic types. The fundamental problem of how to engineer suitable annotations for verification remains largely unchanged compared to procedural languages.

We say that a C function f_A depends on a function f_B iff the function body of f_A (syntactically) contains a function call to f_B. This dependency relation, together with the set of functions of the system forms a directed graph. These graphs may contain arbitrary cycles depending on the implementation of the functions. In the following, for simplicity, we assume the graphs to be acyclic. In practice, mutually recursive functions would have to be specified together in one step and the verification methodology has to ensure that no cyclic reasoning occurs. For longer cycles in the call graph, techniques such as program slicing would allow us to split the graph and consider acyclic parts separately.

Similar to the notion of a root in a tree, we define as the roots in the dependency graph any node without a parent. The depth of a node is defined as the length of the longest path from this node to any of the roots in the graph. The set of all functions that a function f depends on is called $\mathit{children}(f)$; conversely, the set of all functions that depend on f is called $\mathit{parents}(f)$.

The depth of a node can be used to introduce a (topological) ordering $\prec$ on the nodes of the graph: $f_1 \prec f_2$ iff $\mathit{depth}(f_1) < \mathit{depth}(f_2)$. In the following, we identify functions with their corresponding nodes in the dependency graph, and we use the terminology of order theory for functions where appropriate.

2.3 Annotations and their Semantics

In modular deductive verification, the specification SPEC of a software system S is composed of the specifications of its modules (C functions) and data structures. We assume that SPEC is a set of annotations, where each annotation consists of (a) one or more expressions of the specification language (pre-/post-conditions, invariants, assertions, etc.) and (b) the position of the annotation in the program. We further assume that each annotation is local to a single function of the specified system, i.e., SPEC is the disjoint union $\mathit{SPEC} = \mathit{SPEC}_1 \cup \ldots \cup \mathit{SPEC}_n$ of specifications SPEC_i for each of the functions f_i of which S consists.

The binary relation $\models$ between programs and (sets of) annotations denotes the semantics of specifications, i.e., $S \models \mathit{SPEC}$ iff the software system S satisfies the specification SPEC according to the definition of the specification language. Since the specification languages we consider are modular, we have

$$S \models \mathit{SPEC} \quad \text{iff} \quad f_1 \models \mathit{SPEC}_1, \ldots, f_n \models \mathit{SPEC}_n \ .$$

Also, the relation $\models$ is monotonic w.r.t. adding annotations:

$$S \models \mathit{SPEC} \cup \mathit{SPEC}' \ \text{implies} \ S \models \mathit{SPEC} \ .$$

This monotonicity condition requires, for example, that a pre-condition is not considered to be an annotation on its own but only in combination with a post-condition. Adding a lone pre-condition may weaken a specification while adding a pre-/post-condition pair always strengthens it.

2.4 Deductive Verification Systems

To prove that a software system satisfies its specification, we use a verification system V (in our case the VCC tool). The relation $S \vdash_V \mathit{SPEC}$ denotes that V is able to prove that S satisfies SPEC. We assume the verification system to be sound, i.e.,

$$S \vdash_V \mathit{SPEC} \ \text{implies} \ S \models \mathit{SPEC}$$

Any such sound verification system has to be incomplete as any non-trivial system property is undecidable due to Rice's Theorem. Instead, verification systems are supposed to be *relatively complete*, in the sense that they would be complete if there was available an oracle for validity of first-order formulas with arithmetic. In practice, this is rarely an issue: the amount of verification problems where there doesn't exist a proof is negligible compared to the far larger class of problems where the performance of the verification system is the reason a proof is not found in time. For the latter case, the user of a verification tool is prepared to give the prover further hints in form of auxiliary annotations in order to be able to verify a software system to satisfy its specification.

Moreover, all of today's deductive verification systems presuppose certain types of additional, non-requirement annotations to be given by the user. It is neither given nor expected that an annotation-based verification system is relatively complete. In practice, completeness of a verification system means that if the program is correct w.r.t. its *given* requirement specification REQ, then some auxiliary specification AUX *exists* allowing to prove this. In our terminology, $SPEC$ covers both types of annotations, thus in the following $SPEC$ is a synonym for $REQ \cup AUX$.

The goal of the process presented in this work is to obtain a full functional specification $SPEC$ of the system S. For this, we assume that the user can refer to some kind of natural language specification of the system in order to produce the requirement specification REQ in form of annotations.

2.5 The Verification Task

Given a software system S, consisting of the functions $f_1, \ldots, f_n$ and a requirement specification REQ_S, such that $S \models REQ_S$, the task of the user is to find a set of annotations AUX_S, s.t.

$$S \vdash REQ_S \cup AUX_S \ .$$

Typically, these auxiliary annotations include loop and object invariants, lemmas, as well as program code that updates a separate specification memory (which is not visible from the C program during execution).

We tacitly assume that an already proved auxiliary annotation $a \in AUX_S$ can be used in subsequent proofs for other annotations in REQ_S and AUX_S. Thus, by using such auxiliary annotations as *lemmas*, proofs for elements of REQ_S may be greatly simplified or may even become possible at all in a given verification system.

Note that both REQ_S and AUX_S are composed of specifications for each of the functions f_i in S, i.e.,

$$REQ_S = REQ_1 \cup \ldots \cup REQ_n \text{ and } AUX_S = AUX_1 \cup \ldots \cup AUX_n \ .$$

Assuming soundness of the verification system,

$$S \vdash REQ_S \cup AUX_S \text{ implies } S \models REQ_S \cup AUX_S \ ,$$

and due to the requirement that $\models$ is monotonic w.r.t. adding annotations, a solved verification task (i.e. $S \vdash REQ_S \cup AUX_S$) implies

$$S \models REQ_S \ .$$

If, on the other hand, S does not satisfy REQ_S, the verification task has no solution, i.e., no appropriate AUX_S exists. In that case, the verification system may still give the user feedback that helps to correct the requirement specification and/or the implementation.

2.6 The Modular Verification Process

The set of annotations of a function f consists of two parts: one part can be used in the correctness proof for calling functions of f (e.g., pre-/post-conditions of f), while the rest can only be used in the verification of the function f itself (e.g., loop invariants). We call the former set of annotations the *external* specification f, while the latter is named the *internal* specification of f. Which kind of annotation belongs to which of these categories is determined by the verification methodology built into the verification tool and the verification task at hand.

When verifying a function f using a modular verification approach, the external specifications of the children f' of f are used in the correctness proof of f instead of their implementation. Thus, the external parts of the auxiliary specification AUX' of f' are not only relevant for the verification of f' but can also be used as lemmas in the proof of other functions, which in some sense breaks the modularity of the verification process. There exist dependencies between the auxiliary annotations for the different functions, which makes finding a complete set of auxiliary annotations to solve a verification task a difficult problem.

2.7 Top-down and Bottom-up Verification

The user of a deductive verification system may chose different orders in specifying and verifying the modules of a system. The extreme cases are:

Top-down verification The process starts with specifying and verifying the top-level functions with minimal depth in the dependency tree, before proceeding to verify functions with greater depth.

Bottom-up verification The process starts with specifying and verifying functions with maximal depth (leaves in the dependency tree) and proceeds to functions with smaller depth.

In an ideal world, given a prover that never fails due to time-outs, the modular software verification process would proceed top-down, starting with the requirement specification of a top-level function f. All children of f are then specified using the strongest possible contract (which by definition must be sufficient if any annotation is sufficient). Then, f can be proven to be correct with the help of auxiliary internal annotations given by the user. The process repeats with the children of f, until all functions of the system are verified to be correct w.r.t. their specifications.

In a similar fashion, this process could be performed bottom-up: the user annotates each leave in the dependency tree with its strongest possible contract. This contract is always sufficient to prove any parent function correct w.r.t. its specification (if there is no bug in the program or specification). Again, the annotation process is repeated until all maximal functions have been verified to be correct w.r.t. its specification.

Unfortunately, we do not live in an ideal world, and using strongest contracts is not a good idea in practice. They are (a) hard to find and (b) hard to prove. So, in practice, the solution to a verification task is a set of auxiliary annotations that are just (barely) strong enough. To support the user in the long process of finding a solution and making the search less chaotic is the goal of the work presented in this paper.

3 Deductive Verification of Large Software Systems

In practice, a verification attempt may fail for a number of reasons. One significant problem is the performance of available verification tools, which may lead to time-outs. A verification attempt can have the following possible outcomes:

1. Verification of the program w.r.t. its specification succeeds.
2. Verification fails and a counterexample is returned by the prover. In this case, either the program does not satisfy the specification or the auxiliary annotations are not sufficient for the existence of a proof.
3. Verification fails because of a lack of resources (memory or time) and no further indication is given whether the program is correct w.r.t. its specification.

Recall that in modular verification a function is verified using the external specifications of its children. If the verification of a function f succeeds (Case 1 above), then that does not imply that its children are correct w.r.t. their specifications. In case a child function f' does not satisfy its specification, there may or may not be a different auxiliary specification for f' that is both satisfied by f' and sufficient to verify f.

In a similar manner, in Case 2 above, the external specifications of a child f' may be insufficient to verify f. Again, there may or may not be an alternative specification for f' that solves the problem.

When adhering strictly to a bottom-up verification process, one will never encounter the case that one of the children does not satisfy its specification, but it may very well happen that the specification of a child is insufficient to verify its parent. On the other hand, when verifying top-down, one will never end up with insufficient specifications of the children, but a specification of a child f' that is not satisfied by f' may very well occur. That is, independently of the order in which functions are specified and verified, one of the two problems remains.

```
int sqrt(int x)
_(requires x ≥ 0)
_(ensures
  \result^2 ≤ x ∧
  (\result+1)^2 > x)
```

```
int sqrt(int x)
_(requires x ≥ 0)
_(ensures
  ∀y; (y ≤ \result ⇒ y^2 ≤ x^2) ∧
  ∀z; (z > \result ⇒ z^2 > x^2))
```

Fig. 1: Two alternative contracts for the `sqrt` function.

Even always using the strongest possible contract for all functions is not an option here: while providing a stronger contract for a function f' may help in the verification of the parents of f', it also makes verifying f' more difficult. In practice, the user has to provide a specification for f' that is strong enough to verify all parent functions of f' and weak enough to verify f' itself.

Moreover, the logical strength of a contract is not its only relevant property but its syntactic form is just as important.

Example 1. Consider the two contracts for the function `sqrt` that computes (an approximation of) the square root of an integer shown in Fig. 1. While both contracts are logically equivalent, i.e., specify the same behaviour of `sqrt`, one of the two contracts may be much more useful than the other one, depending on the verification tool used and the properties that are needed in the verification of a caller of `sqrt`.

As it is hard to foresee which specification of a function f is the most appropriate without paying attention to all call sites in the parents, in practice, neither a strict top-down nor a strict bottom-up approach is applied. Instead during the process a continuous adaptation of the specification takes place during which the specifications of calling and called functions are changed in alternation until verification of the software as a whole succeeds – this process is shown in Fig. 2a. A further possibility, which is not shown in the figure, is that the verification process fails because the implementation does not satisfy the requirement specification (in which case refining the annotations cannot help). Then, the implementation and/or the requirement specification need to be changed and the verification process restarted. The iterative process shown in Figure 2a is often applied locally, i.e., only one pair of caller and callee is considered at a time. As other functions may also use the callee and depend on its contract, changes in this contract may have to be propagated to various other parts of the system.

4 Software Bounded Model Checking

Software bounded model checking (SBMC), like deductive verification, is a formal method aimed at reasoning about behavior and correctness of software systems. In contrast to deductive software verification techniques, it is based on an exhaustive search for a counterexample to the desired properties, rather than on constructing a deductive proof.

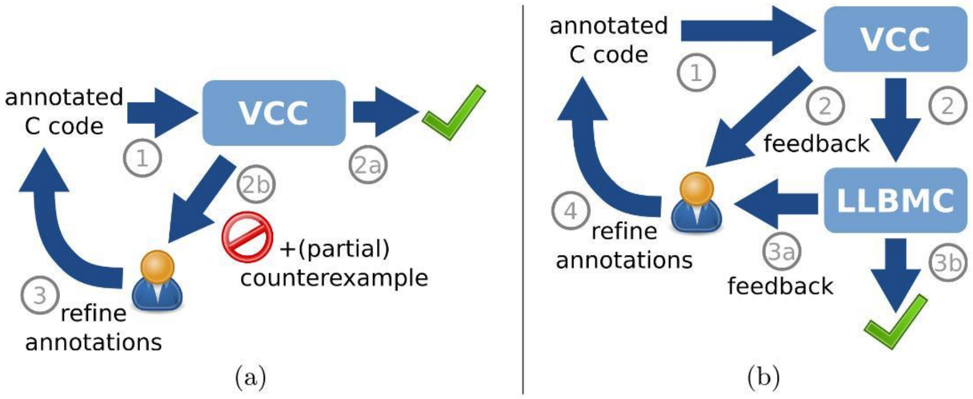


Fig. 2: (a) Normal VCC workflow and (b) counterexample guided *manual annotation* refinement (CEGMAR).

The SBMC implementation used in this work is LLBMC [14], the Low-Level Software Bounded Model Checker developed at the Karlsruhe Institute of Technology. It implements software bounded model checking of C and C++ programs and provides bit-precise reasoning, comprehensive support of the C language and a precise memory model supporting dynamic memory allocation.

LLBMC takes an LLVM bitcode file as input and first performs loop unrolling on all relevant functions. Starting with the main function, the result is then translated into an intermediate logic representation (ILR). Function calls are inlined on the fly during conversion, and logic encodings of the properties to be proven are inserted into the formula. The resulting ILR formula is then simplified, translated to SMT and passed to an SMT solver. If the formula is satisfiable, the SMT solver's model is translated back into assignments for the variables on the LLVM level. In the following, when we say SBMC we mean the method as implemented by LLBMC.

In SBMC, assertions are inserted into the code at strategic places. These assertions can be synthesized by the SBMC tool itself (e.g., for division-by-zero checks or memory access checks), or they can be provided by the user. User-provided assertions are similar to `assert` statements in C, in that they can be used to check certain properties of the software system at the specific state that the system is in when the assertion is executed. But in contrast to C's `assert` statement, which only performs run-time checks for an execution at hand, SBMC statically checks assertions for all possible input values at once. The software system is considered correct only if for every set of possible input values the assertion holds. In addition to assertions – and in contrast to C – SBMC also provides so-called assumptions. These are used similarly to assertions, except that execution paths where the assumption is violated are pruned and not analyzed any further. This means that any violation of an assertion following a violation of an assumption (for a certain execution trace) is ignored.

Assumptions and assertions in combination provide a simple means for expressing specifications, where assumptions are used to express pre-conditions and assertions are used for post-conditions. Compared with VCC's specification language, this specification method is quite restricted. It only allows what is expressible in C, which means, e.g., that quantifiers are not (directly) supported. For an existential quantifier it is often possible, however, to mimic its behavior by providing an explicit construction of the element satisfying the given property. Finite universal quantifiers might be replaced by a loop ranging over all elements. This comes at an increased cost in checking the property, though.

LLBMC implements SBMC by first removing cycles in the control flow. This is done by inlining all function calls and by unrolling all loops. For this first step to terminate, a fixed upper bound on the number of inlining and unrolling operations is required. Thus SBMC is only applicable to fixed-bound execution traces. It is a complete verification methodology only if for a given software system a bound exists, such that no loop is executed more often than this bound, and the function call depth is never deeper than that bound. Creating an acyclic control flow graph is essential for the high performance of SBMC, but it also makes SBMC non-modular.

5 The Integrated Verification Process

In the following, we describe how to integrate software bounded model checking into the annotation-based deductive verification process, thereby taking advantage of the strengths of both methods. As said above, we use the tools VCC and LLBMC to illustrate our approach.

The central idea of our method is to use SBMC to support the process of finding a set of auxiliary annotations, AUX, for a given system S that allows the deductive verification tool to prove that S satisfies its requirement specification REQ. The resulting integrated process is illustrated in Fig. 2b. The name CEGMAR is inspired by the counterexample-guided abstraction refinement (CEGAR) technique [3] used in model checking – though in CEGMAR annotations are refined instead of abstractions.

Note, that we do not use the term refinement in its strict mathematical meaning here. Instead we have a more colloquial interpretation in mind, where refinement simply means iterations towards a specification which is fit for its purpose. Also note that this kind of refinement contains a manual component, which makes CEGMAR a machine-supported verification process, not a fully automatic algorithm.

CEGMAR aims at finding suitable auxiliary specifications for the full system S, but at any given point in time, some function f is in the focus of the process. The process starts from the given requirement specification REQ_f for a function f and a (possibly empty) set AUX_f of auxiliary annotations.

In Step 1 (Fig. 2b), the annotated C code relevant for proving f correct w.r.t. its (requirement and auxiliary) specification is passed to VCC for verification. The result of VCC's verification attempt for f is given to LLBMC (Step 2 in Fig. 2b). Then, in case both VCC and LLBMC agree that f satisfies its specification, the refinement-loop for f ends successfully (Step 3b). After this, some other function moves into focus or, if all functions have been verified, the verification task has been successfully completed.

Otherwise, if one of the tools (or both) fails to verify f, the user has to refine some of the auxiliary annotations (Step 4), using the feedback of VCC and LLBMC (from Step 2 resp. 3a). After refining the auxiliary annotations in Step 4, the next iteration starts with Step 1. If changing the auxiliary specifications is not sufficient according to the feedback from the tools, i.e., there is a problem in the implementation or the requirement specification, then the refinement loop for f terminates and can only be restarted after the implementation and/or the requirements have been fixed.

In Step 2, using VCC, the correctness of f is only proven *locally*, i.e., the external specifications for $\mathit{children}(f)$ are used without checking that they are satisfied. Feedback from VCC is either (a) the statement that f is correct w.r.t. its specification or (b) a list of annotations that cannot be proven (possibly together with counterexamples). In contrast, LLBMC is used in our integrated approach to check correctness of a function f *globally*, i.e., the implementation of all functions called by f (directly or indirectly) is taken into account. We identified three different properties to be checked with LLBMC:

A. f satisfies its specification;
B. all functions in $\mathit{children}(f)$ satisfy their external specifications;
C. the pre-condition of f holds at all points where f is called in $\mathit{parents}(f)$ (invocation contexts).

Each of the three checks A–C has three possible outcomes: either (a) the property in question holds up to a certain bound on the length of traces, or (b) LLBMC provides an error trace falsifying the property, or (c) there is a time out.

The three checks differ in which part of the implementation and annotations are given to LLBMC, as well as the consequences of the check for the verification process, as described in the following.

A: Checking that f satisfies its specification. For this, the implementations of f and all descendants of f are passed to LLBMC for model checking. The implementation of f is checked w.r.t. all annotations of f that VCC reports to be violated. LLBMC can provide the user with feedback on which unproven specifications are indeed violated by the implementation and which are likely satisfied (because no counterexample was found within the given bound), but just not provable by VCC without refining the annotations.

Even if VCC could verify that f is correct (based on the external specification of functions called by f), LLBMC is still applied. This allows to discover cases where VCC's correctness proof for f only succeeded due to an erroneous external specification of a child of f.

B: Checking that child functions satisfy their external specification. The implementations of all descendants of f are passed to LLBMC for model checking. The functions in $\mathit{children}(f)$ are checked to satisfy their external specifications. This check helps to rule out correctness proofs for f that are erroneous because they rely on faulty specifications of f's children.

C: Checking Invocation Contexts. For each function $g \in \mathit{parents}(f)$, the implementations of g and all descendants of g are passed to LLBMC. Here, the property to check is the pre-condition of f. Checking the invocation contexts helps to avoid writing specification for f that cannot be used in the proofs for other functions in the system.

Note that in all these cases, LLBMC is not used to check whether a function satisfies an annotation in general, but to check that the function satisfies the annotation in the context in which it is called. The context may be defined by the function's pre-condition or by the context in one of its parents.

The benefit of the proposed integration of SBMC into deductive verification results mainly from the fact that SBMC is not modular. During the verification of a function f, LLBMC uses the implementation of the children and parents of f instead of (only) using the external specification of the children as VCC does. Because of this difference, LLBMC and VCC can provide the user with different information about the functions and their annotations (e.g. counterexamples). For the verification process, the information provided by LLBMC is a valuable addition to the information provided by VCC.

In Section 7, we give an example on how this additional information improves the verification process.

6 Translation of Specification into LLBMC Input

As explained in Section 4, LLBMC's specification formalism is different from VCC's, so VCC specifications have to be translated into equivalent LLBMC input. Because LLBMC's assertions cannot contain quantifiers, all quantifier-expressions have to be emulated by C loops that explicitly enumerate the domain of quantification. By applying these transformations, VCC's declarative specification style is turned into an imperative specification style.

For pre- and post-conditions, VCC's `ensures` are translated into one or more `assert` statements, and VCC's `requires` are translated into one or more `assume`s. Usually, a special *verification driver* function is written for this (to a large fraction this can be automatized), which has the same function parameters as the function under check. Thereby, all parameters are considered as uninitialized variables, and the check runs over all possible values for these variables. The driver function first executes the `assume`s corresponding to the pre-condition, then the function itself, and finally the `assert` statements corresponding to the post-condition. LLBMC is then applied to this verification driver function.

Other annotations are similarly translated and inserted into the C code. For example, if loop invariants need to be checked, corresponding `assert` statements are inserted before the loop, and at the end of each loop body execution, making sure that the loop invariant indeed holds.

For a first evaluation, the translation from VCC specifications to LLBMC's `assume`s and `assert`s was accomplished manually. To fully exploit the strength of the integrated process, we are planning to automatize this translation process as far as possible (automatic loop generation for quantifiers may pose a restriction, however).

A partial VCC specification of the function `copyNoDuplicates` (from the example in Section 7) and the translation of that specification into LLBMC input is shown in Fig. 3.

```
//no 'new' items in result
_(ensures \result != NULL ==>
  ∀ uint i; i < \result->count ==>
    (∃ uint j; j < source->count ∧
      \result->items[i] ==
        source->items[j]))
//no duplicate items in result
_(ensures \result != NULL ==>
  ∀ uint i; ∀ uint j;
   (i < \result->count ∧
    j < \result->count ∧
    \result->items[i] ==
      \result->items[j]) ==> i == j)
```

```
cnt = result->count;
//no 'new' items in result
if (result != NULL)
  for (i = 0; i < cnt; ++i) {
    int found = 0;
    for (j = 0; j < cnt; ++j)
      if (result->items[i] ==
                source->items[j])
        result = 1;
    assert(found == 1);
  }
//no duplicate items in result
if (result != NULL)
  for (i = 0; i < cnt; ++i)
    for (j = 0; j < cnt; ++j)
      if (result->items[i] ==
                result->items[j])
        assert(i == j);
```

Fig. 3: Partial requirement specification of `copyNoDuplicates` (left) and its translation into LLBMC input (right).

7 A Typical Specification Scenario

7.1 The System to be Verified

In the following we present an example that demonstrates the issues of modular verification mentioned before and how integration of software bounded model checking into the verification process can help attenuate these.

Consider the following C data structure implementing a sequence data type:

```
typedef struct queue_t {
    int *items;
    int count, capacity;
} queue, *pQueue;
```

Here, `count` denotes the length of the sequence and `capacity` the fixed size of memory that has been allocated to store the items of the sequence. In our

```
pQueue copyNoDuplicates(pQueue source) {
  pQueue dest = initQueue(5);
  for (int i = 0; i < source->count; i++)
  {
    int sVal = source->items[i];
    int j = 0, contained = 0;
    while(j < dest->count && dest->items[j] <= sVal) {
      if (dest->items[j] == sVal) {
        contained = 1;
        break;
      }
      j++;
    }
    if (!contained) insert(dest, sVal);
  }
  return dest;
}
```

Fig. 4: Implementation of `copyNoDuplicates`.

case, the items of the sequence are integers and are stored in the array that starts at the memory address `items`.

The top-level function we want to verify is `copyNoDuplicates` (see Fig. 4), but in total there are three functions involved in the verification process:

- `pQueue copyNoDuplicates(pQueue q)`
  Given a queue q, this function returns a new queue q', which is a copy of q, except that duplicate elements of q occur only once in q'.
- `pQueue initQueue(int capacity)` (not shown)
  Allocates memory for a new queue structure as well as the appropriate amount of memory for storing `capacity` number of items. It also initializes the queue data structure to correspond to the empty sequence. This function is called by `copyNoDuplicates`.
- `void insert(pQueue q, int val)` (shown in Fig. 5)
  Inserts an item *val* into a queue q in such a way that if the queue is in ascending order, it remains ordered. This function is called by `copyNoDuplicates`.

There are two peculiarities about this implementation of `copyNoDuplicates` that the verification engineer might not be aware of:

1. Queues maintain their elements in sorted order, and the algorithm for copying without duplicates relies on `insert` retaining sortedness of the queue. This is because the algorithm stops searching for a matching element as soon as a greater element is encountered.
2. Inserting into a queue fails silently if the capacity of the queue is reached.

In the following, we will use the example to show why a user who is not supported by software bounded model checking will have trouble identifying

```
void insert(pQueue q, int val)
{
  if (q->count == q->capacity) return;

  int i,j;
  for (i = 0; i < q->count && val > q->items[i]; i++) {}
  for (j = q->capacity-1; j>i; j--)
    q->items[j] = q->items[j-1];

  q->items[i] = val;
  q->count++;
  return;
}
```

Fig. 5: Implementation of `insert`.

these problems during verification, independently of whether a top-down or a bottom-up approach is chosen.

7.2 Bottom-up Verification of `copyNoDuplicates`

When verifying bottom-up, the first two functions that are to be verified correct in our case are `initQueue` and `insert`. Because we are mainly interested in interaction between `copyNoDuplicates` and `insert`, from now on we assume that `initQueue` has been verified and does not need to be considered further.

The second issue mentioned in the previous subsection (finite capacity of the queue) is identified quickly with a bottom-up approach and therefore not further discussed. The first issue (sortedness) is considerably harder to identify, though.

Suppose that no requirement specification is given for `insert`, so any specification of `insert` is auxiliary. It is likely that the user correctly specifies that the item passed to `insert` is indeed inserted in the given queue. However, it is also likely that the verification engineer on the first try is not aware of the importance of sortedness and, consequently, the specification does not mention that insertion of elements retains sortedness of the queue. In that case, the specification of `insert` is not strong enough and verification of `copyNoDuplicates` will fail. But that is only noticed after `insert` has been successfully verified and the verification process has moved on to `copyNoDuplicates`.

Once `copyNoDuplicates` could not be proven correct, the verification engineer has to identify the cause for this. The too weak specification of `insert` is hard to spot, and the user may be tempted to believe `copyNoDuplicates` is not correctly implemented. The counterexample provided by VCC usually does not contain all necessary information to understand the issue.

LLBMC on the other hand states, using Check A from Section 5, that `copyNoDuplicates` does indeed satisfy its requirement specification and the relevant loop invariants – at least up to a certain size of the queue[1]. This indicates to the user that copyNoDuplicates is likely correct and the problem is either that the specification of insert is too weak or the auxiliary annotations are not enough to allow VCC to verify the property. This is an important cue towards the right direction and can therefore speed up the verification process.

7.3 Top-down Verification of copyNoDuplicates

In a top-down verification approach, the top-level function copyNoDuplicates is the first to be verified. The first issue (sortedness of the queue) is found early in the process, as the verification of copyNoDuplicates will already uncover it. Instead, the second issue (finite capacity of the queue) is now causing problems.

Consider a requirement specification of copyNoDuplicates consisting of an empty pre-condition and the following post-condition stating that all elements of the source queue are also contained in the resulting queue (in fact, this is only part of the actual requirement specification because it does not state that the result should not contain duplicates):

```
∀i; i ≥ 0 ∧ i < source->count  ⟹
    ∃j; j ≥ 0 ∧ j < \result->count ∧
        source->items[i] == \result->items[j]
```

In order to verify the implementation of copyNoDuplicates to satisfy this requirement, the user has to provide auxiliary specifications for the helper functions initQueue and insert. The verification of these auxiliary specifications is postponed in the top-down approach until after the contract of copyNoDuplicates is proven. Nevertheless they are already used in the proof for copyNoDuplicates.

If the user annotates insert he/she may easily overlook the case where the queue has reached its capacity and insertion of yet another element fails (signaled by insert by returning an error code). Now, because of this omission, the specification of insert is too strong, which allows VCC to prove the contract of copyNoDuplicates – even though it is in fact not satisfied. Only when the verification of the contract of insert fails, this error is detected.

Then, after fixing the specification of insert, the verification engineer has to go back to copyNoDuplicates and re-verify that function, taking the modified specification of insert into account. In practice, the top-down approach results in numerous iterations until a function and all of its children are verified.

Using LLBMC can help resolve this issue early on, as LLBMC directly takes the implementation of insert into account, and not just its (too strong) specification. LLBMC uncovers the problem as soon as copyNoDuplicates is checked – even though VCC cannot detect any problem at this point.

[1] This size is determined by the bound applied during model checking.

8 Related Work

Our work is related to previous work about combinations of model checking and deductive verification, improvements to the software verification process, as well as to tools and techniques that help understand failed verification attempts.

Various combinations of model checking and deductive verification have been proposed and studied in the past, e.g., [2, 8]. An extensive overview of the work published until 2000, with a focus on the verification of reactive systems,[2] is given in [16]. Since then, research in this area seems to have slowed down.

Some papers use a combination of model checking and deductive verification techniques to improve performance and generality of existing verification tools, such as [13]. Others use deductive methods specifically to extend model checking approaches to infinite-state systems, e.g. [10, 15, 6].

Most of these papers focus on improving performance of the tools or creating more powerful verification tools. In contrast to this we focus entirely on improving the process of annotation-based deductive software verification.

On a different note, Müller and Ruskiewicz use debuggers to address the problem of understanding failed verification attempts [12]. They, too, provide concrete counterexamples that illustrate why verification did not succeed. While error traces are an important part of our contribution to the deductive verification process, the proposed process is not restricted to counterexamples.

Similarly, Vanoverberghe et al. use symbolic execution techniques to generate test cases when the prover fails [17]. They also generate test cases that can be executed in a debugger to analyze the failed verification for a concrete example.

Alex Groce et al.'s tool `explain` [7] provides additional information about counterexamples generated by the bounded model checker CBMC. The tool itself uses CBMC's software bounded model checking engine to generate executions similar to an existing counterexample that do not fail, and additional counterexamples that are as different as possible but do still fail. The approach does not seem to be directly applicable to deductive verification techniques due to the lack of a concrete counterexample to start with.

9 Conclusion and Future Work

Integrating software bounded model checking into the deductive verification process can give the user of deductive verification tools early feedback, thereby decreasing the verification effort and improving the overall verification process. We showed that SBMC can help in finding insufficient or wrong annotations. An example was provided showing that SBMC can help both in top-down and bottom-up verification. It can also help in identifying specifications that are either too weak or too strong. In the future we plan to further integrate VCC and LLBMC, so that specifications in VCC can be fully automatically translated

[2] The combination of temporal logic properties and an infinite-state system makes reactive systems a fitting application for combinations of model checking and deductive verification.

into to LLBMC input, so that larger case studies can be carried out. Ideas from the area of instrumenting code for run-time checking specifications will be useful here. We also expect that it will be necessary to extend LLBMC's specification language and possibly LLBMC itself in order to allow for a more complete translation of VCC specifications. In the long term, LLBMC could also be used to automatically derive simple annotations – such as non-nullness of pointers and simple ownership relations – thereby speeding up the specification process.

References

1. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005*, pages 364–387, 2005.
2. S. Berezin, K. McMillan, and C. B. Labs. Model checking and theorem proving: a unified framework. Technical report, Carnegie Mellon Univ., 2002.
3. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV'00*, pages 154–169, 2000.
4. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, T. Santen, M. Moskal, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs'09*, pages 23–42, 2009.
5. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08*, pages 337–340, 2008.
6. G. Delzanno and A. Podelski. Constraint-based deductive model checking. *Software Tools for Technology Transfer*, 3(3):250–270, 2001.
7. A. Groce, D. Kroening, and F. Lerda. Understanding counterexamples with explain. In *CAV'04*, pages 453–456, 2004.
8. Y. Kesten, A. Klein, A. Pnueli, and G. Raanan. A perfect verification: Combining model checking with deductive analysis to verify real-life software. In *FM'99*, pages 173–194, 1999.
9. K. R. M. Leino and M. Moskal. Usable auto-active verification. Technical Report Manuscript KRML 212, Microsoft Research, 2010.
10. Z. Mann, A. Anuchitanukul, N. Bjorner, A. Browne, E. Chang, M. Colon, L. de Alfaro, H. Devarajan, H. Sipma, and T. E. Uribe. STeP: The Stanford Temporal Prover. Technical report, Stanford Univ., 1994.
11. M. Moskal. *Satisfiability Modulo Software.* PhD thesis, Univ. of Wrocław, 2009.
12. P. Müller and J. N. Ruskiewicz. Using debuggers to understand failed verification attempts. In *FM'11*, pages 73–87, 2011.
13. A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In *CAV'96*, pages 184–195, 1996.
14. C. Sinz, S. Falke, and F. Merz. A precise memory model for low-level bounded model checking. In *SSV'10*, 2010.
15. H. Sipma, T. E. Uribe, and Z. Manna. Deductive model checking. *Formal Methods in System Design*, 15(1):49–74, 1999.
16. T. E. Uribe. Combinations of model checking and theorem proving. In *FroCoS'00*, pages 151–170, 2000.
17. D. Vanoverberghe, N. Bjørner, J. D. Halleux, W. Schulte, and N. Tillmann. Using dynamic symbolic execution to improve deductive verification. In *SPIN'08*, pages 9–25, 2008.

A Framework for Object-Oriented Modeling and Analysis of Probabilistic Open Distributed Systems*

Lucian Bentea and Olaf Owe

Department of Informatics, University of Oslo
{lucianb,olaf}@ifi.uio.no

Abstract. Creol is an executable, formally defined modeling language with advanced object-oriented features, tailored for modeling software systems consisting of physically distributed components, each running on its own processor and communicating by means of asynchronous method calls. In this paper we propose a probabilistic extension of Creol, called *PCreol*, and give its operational semantics by means of probabilistic rewrite theories. The extension is motivated by the need to model: a) communication over networks of different quality, b) software components running randomized algorithms, c) independent processor speeds, or d) an open environment exhibiting probabilistic behavior. The syntax of PCreol therefore includes a probabilistic choice operator, random assignment and means for modeling random numbers. The semantics of Creol is also extended to model lossy communication. We give details on the implementation of a prototype PCreol interpreter—on top of the existing one for Creol—which is executable in Maude. Furthermore, we integrate PCreol with the VeStA tool, to support probabilistic reasoning of PCreol models by statistical model checking and quantitative analysis. This way, representative runs of a PCreol model are more easily obtained by discrete-event simulation and the model checking problem of large models becomes feasible. We also provide concrete examples of PCreol models and show how VeStA can be used to reason about them.

1 Introduction

Software systems of today are often distributed, consisting of independent and concurrently executing units which communicate over networks of different quality, and which are supposed to work in open and evolving environments. Due to nondeterminism, it is non-trivial to design, model and program such systems, and in particular to analyze system properties and reliability. For such systems, non-functional properties expressing probabilistic behaviour are valuable.

At the modeling level there is a need for high-level syntactic constructs making interaction and process control more manageable. There is also a need for

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

methodology and tools that can be used to investigate system properties and robustness. Creol is an executable modeling language introduced in [12, 15, 13] that tries to address these challenges. It is tailored for modeling software systems made up of physically distributed components, each running on its own processor and communicating with one another by means of asynchronous method calls. A Creol system runs in an open environment in which components may appear or disappear. In addition, components may change their functionality during execution. The language features conditional and unconditional processor release points, allowing an object to suspend the execution of a process until a later time, and resume the execution of another (enabled) process.

In this paper, we extend the syntax and semantics of Creol with probabilistic features, to be able to model lossy communication, independent processor speeds, software components running randomized algorithms, or an open environment exhibiting probabilistic behaviour; this extension is given the name *PCreol*. We introduce new syntactic constructs—probabilistic choice operators, random assignment, and means for modeling random numbers—and give their operational semantics by means of probabilistic rewrite theories [17].

Furthermore, model checking Creol models can currently only be achieved using Maude's LTL model checker [9] and its breadth-first state exploration facilities. The disadvantage is that even average-sized Creol models lead to state space explosion, making it infeasible to model check them using Maude. We have therefore integrated PCreol with VeStA [22], which allows for probabilistic reasoning of PCreol models via statistical model checking and quantitative analysis—an approach that scales well with the size of the model. This integration is essentially achieved by refining the original Creol operational semantics, so that *representative* runs of a Creol model are more easily obtained by discrete-event simulation. The model checking problem of large PCreol models therefore becomes feasible since breadth-first search is avoided and replaced by repeated simulation of different representative runs, controlled probabilistically.

PMaude [3] is a language for specifying probabilistic object systems by means of equations and probabilistic rewrite rules. The technique that we use to refine the operational semantics of PCreol is essentially the same as in [3] for obtaining executable PMaude modules from possibly nondeterministic ones, and includes adding an explicit notion of time to the global configuration of the model, as well as scheduling objects to execute at random instants of time. This resolves all nondeterminism in the interpreter, allowing VeStA to run discrete-event simulations and do statistical analysis of PCreol models.

Related Work. The generalized probabilistic choice operator, as well as the random assignment in PCreol are inspired by similar syntactic constructs in a probabilistic version of ProMeLa [11] called ProbMeLa [6]. For instance, our generalized probabilistic choice operator can be expressed using the **pif** ... **ifp** construct in ProbMeLa. However, PCreol is an object-oriented modeling language with advanced features like inheritance, future variables and asynchronous communication, which ProbMeLa lacks. PRISM [18] is another similar modeling language without object-oriented features, but which comes with powerful, exact probabilistic model checking tools that PCreol is missing since VeStA is a non-exact (statistical) probabilistic model checker.

The paper is structured as follows: Section 2 contains preliminaries on probabilistic rewrite theories, PMaude and VeStA, and Section 3 provides an overview of the Creol modeling language, focusing on the features that we extend to the probabilistic setting. Section 4 gives the syntax and the operational semantics of PCreol by means of probabilistic rewrite theories. In Section 5 we provide concrete examples of PCreol models and show how VeStA can be used to reason about them. Section 6 provides details on the actual Maude implementation of the PCreol interpreter, and Section 7 suggests several topics for future work.

2 Preliminaries

Using rewriting logic [19] the static parts of a system (e.g., data types, functions, etc.) can be defined equationally and its transitions can be specified by labeled conditional rewrite rules of the form $l:\ t \longrightarrow t'$ **if** $cond$, where t and t' are *terms* over typed variables and function symbols, l is a rule label, and $cond$ is a conjunction of equalities, sort memberships, and rewrites. Typing is given in terms of sorts and subsorts, and each sort belongs to a kind. Such rules specify conditional local transitions from an instance of the term t to the corresponding instance of the term t', where the condition $cond$ must be satisfied by the substitution instance in order for the transition to take place. In what follows, we give a formal definition to rewrite theories and their probabilistic extension.

Given a set K of *kinds*, a *many-kinded signature* Σ contains a set of function declarations of the form $f : k_1 \dots k_n \to k$, where $n \geq 0$ and $k_1, \dots, k_n, k \in K$. In *membership equational logic* (MEL) [20] each kind k has an associated set of sorts S_k. A *MEL theory* consists of a MEL signature $(K, \Sigma, \{S_k \mid k \in K\})$, also denoted Σ, and a set E of (possibly) conditional equations $(\forall \vec{x})\ \ t = t'$ **if** $cond$ and membership axioms $(\forall \vec{x})\ \ t : s$ **if** $cond$, where t and t' are Σ-terms of the same kind k, s is a sort of kind k, $cond$ is a conjunction of equalities and sort memberships, and $\vec{x}$ denotes the set of variables in these axioms. We write $vars(t)$ for the set of variables occurring in a term t; if $vars(t) = \emptyset$, then t is called a *ground term.* If $(\Sigma, E \cup A)$ is a MEL theory, where A is a collection of *structural axioms* specifying properties of function symbols, like commutativity, associativity, etc., and E is terminating, confluent and sort-decreasing modulo A, then $Can_{\Sigma,E/A}$ denotes the algebra of A-equivalence classes of ground terms fully simplified by the equations E, i.e., $Can_{\Sigma,E/A}$ is the algebra of canonical terms, or "normal forms." We denote by $[t]_A$ the A-equivalence class of a fully simplified term t. Under suitable assumptions, each normal form can therefore be assigned the least possible sort among those of all other equivalent terms. An equation is sort-decreasing if the sort of the right-hand side is smaller than that of the left-hand side.

An E/A*-canonical ground substitution* for a set of variables $\vec{x}$ is a function $[\theta]_A : \vec{x} \to Can_{\Sigma,E/A}$ that assigns a fully simplified ground term to each variable in $\vec{x}$. We denote by $CanGSubst_{E/A}(\vec{x})$ the set of all such functions. A *rewrite* *theory* [19] is a tuple $\mathcal{R} = (\Sigma, E, L, R)$, where Σ is a MEL signature, (Σ, E) is a MEL theory (E also contains the set of structural axioms A), L is a set of labels, and R is a set of labeled conditional rewrite rules

$$(\forall \vec{y}) \quad l : \ t \longrightarrow t' \ \ \textbf{if} \ \ cond, \tag{1}$$

where $l \in L$ is a label, t and t' are terms of the same kind, *cond* is a conjunction of equalities, memberships and rewrites, and $\vec{y} = vars(t) \cup vars(t') \cup vars(cond)$. Maude [9] can be used to verify and simulate systems specified as rewrite theories.

Let Ω be a nonempty set. If Ω is countable, a *probability mass function*, or *probability distribution* over Ω is any mapping $p : \Omega \to [0, 1]$ with the property that $\sum_{\omega \in \Omega} p(\omega) = 1$; we denote by $Dist(\Omega)$ the set of all probability distributions over the set Ω. In [17] rewrite theories are extended to *probabilistic rewrite theories*, in which the right-hand side t' of a rewrite rule $l : \ t \longrightarrow t' \ \ \textbf{if} \ \ cond$ may contain variables $\vec{p}$ that do not occur in t. These new variables are assigned values according to a probability distribution taken from a *family* of probability distributions—one for each instance of the variables in t—associated with the rule. Formally, a probabilistic rewrite theory is a pair $(\mathcal{R}, \pi)$, where $\mathcal{R}$ is a rewrite theory and π is a function which assigns to each rule $r \in R$ of the form (1), with $vars(t) = \vec{x}$, $vars(cond) \subseteq vars(t)$, and $vars(t') \setminus vars(t) = \vec{p}$, a mapping[1]

$$\pi_r : [\![cond(\vec{x})]\!] \to Dist\left(CanGSubst_{E/A}(\vec{p})\right),$$

where $[\![cond(\vec{x})]\!]$ is the set of all E/A-canonical ground substitutions for $\vec{x}$ that satisfy the condition *cond*. That is, for each substitution θ of the variables in t which satisfies *cond*, we get a probability distribution $\pi_r\left([\theta]_A\right)$ that defines how the new variables $\vec{p}$ are instantiated. A rewrite rule $r \in R$ of the form (1) with $vars(t') \setminus vars(t) \neq \emptyset$, together with its associated probability distribution function π_r is called a *probabilistic rewrite rule* and is written $l : \ t \longrightarrow t' \ \ \textbf{if} \ \ cond \ \ \textbf{with probability} \ \ \pi_r$. If $[\![cond(\vec{x})]\!] = \emptyset$ due to $\vec{p}$ being empty, we say that π_r defines a *trivial distribution*. Probabilistic rewrite theories can therefore express both probabilistic and fully nondeterministic behavior.

Probabilistic rewrite rules are nondeterministic due to the new variables in their right-hand sides, rendering them nonexecutable in Maude. However, Maude can be used to *simulate* probabilistic rewrite theories, provided that all the new variables are replaced with actual values *sampled* from the corresponding probability distribution. In [3] *actor PMaude* modules are introduced, which can be used to create executable PMaude specifications, that are free from any nondeterminism. This is achieved by considering the current state of the system as a multiset of *actors* and *messages*, in which time is made explicit through a global real value. When creating an executable PMaude specification from a nondeterministic one, all actors in the original specification are *scheduled* to execute at random moments of time, with the interval between two consecutive

[1] In [17] the definition of probabilistic rewrite theories uses the more general notion of *probability measures* on a σ-algebra over $CanGSubst_{E/A}(\vec{p})$. To simplify the exposition, we only consider probability mass functions.

Syntactic categories.	*Definitions.*
C, I, m in Names	$IF ::= \texttt{interface}\ I\ [\texttt{inherits}\ \overline{I}]\texttt{begin}\ [\texttt{with}\ I\ \overline{Sg}]\ \texttt{end}$
t in Label	$CL ::= \texttt{class}\ C\ [(\overline{x : I})]\ [\texttt{inherits}\ \overline{C}]\ [\texttt{implements}\ \overline{I}]$
g in Guard	$\quad \texttt{begin}\ \overline{\texttt{var}\ x : I [:= e]}\ \overline{[\texttt{with}\ I]\ \overline{M}}\ \texttt{end}$
s in Stmt	$M ::= Sg == [\texttt{var}\ \overline{x : I [:= e]};]\ s$
x in Var	$Sg ::= \texttt{op}\ m\ ([\texttt{in}\ \overline{x : I}][\texttt{out}\ \overline{x : I}])$
e in Expr	$g ::= b \mid t? \mid g \wedge g \mid g \vee g$
o in ObjExpr	$s ::= \texttt{begin}\ \overline{s}\ \texttt{end} \mid \overline{s}; \overline{s} \mid x := e \mid x := \texttt{new}\ C\ [(\overline{e})]$
b in BoolExpr	$\mid \texttt{if}\ b\ \texttt{then}\ \overline{s}\ [\texttt{else}\ \overline{s}]\ \texttt{end} \mid \texttt{while}\ b\ \texttt{do}\ \overline{s}\ \texttt{end} \mid \texttt{await}\ g$
	$\mid [t]![o.]m(\overline{e}) \mid t?(\overline{x}) \mid \texttt{await}\ g \mid [\texttt{await}][o.]m(\overline{e}; \overline{x})$

Fig. 1. The syntax of Creol. The terms denoted by $\overline{e}$, $\overline{x}$, and $\overline{s}$ represent lists over terms of the corresponding syntactic categories, and [...] denotes optional elements. Elements in a list are separated by a comma, while statements in a statement list are separated by semicolon.

executions following a continuous, exponential probability distribution with a fixed rated parameter. Therefore, the probability that two actors are scheduled at the same time is zero.

The VeStA tool [22] can be used to generate execution traces from executable actor PMaude modules in which all nondeterminism has been resolved. It allows to *statistically model check* these modules against probabilistic temporal formulas expressed in Continuous Stochastic Logic [5], giving an alternative to Maude's search and model checking commands. Due to its statistical nature, the result of the model checking procedure is given with some level of confidence, which can be tweaked; higher confidence requires a larger number of execution traces. VeStA also allows the quantitative analysis of executable PMaude modules, via quantitative temporal expressions given in the QuaTEx logic [3] which is more expressive than Probabilistic Computation Tree Logic (PCTL) [10]. These expressions relate the current state of the system to a numerical quantity, through a formula defined equationally in rewriting logic. VeStA estimates the average value of such an expression, within a given confidence interval.

3 Overview of Creol

We provide a brief introduction to the syntax and semantics of the Creol object-oriented modeling language. A summary of the language syntax is given in Figure 1, as it appears in [13]. For a detailed introduction to Creol we refer to [12]. In this paper, we extend the syntactic category Stmt of Creol statements.

Among the main Creol statements we mention those which we extend to the probabilistic setting: *nondeterministic choice* with the syntax $\mathrm{s}_1 \ \Box \ \mathrm{s}_2$, where s_1 and s_2 are statement lists, and *asynchronous method calls* with the syntax $t!o.m(\overline{e})$, where o is the callee, m the called method with actual parameter $\overline{e}$, and t is a label variable that can be used to query for the return value of this method call. A label variable t is similar to a *future variable* [1], except that it is used locally and cannot be passed as a parameter. However, in the operational semantics it appears as an implicit parameter, called `label`. Creol methods are declared in the context of a *cointerface*, given by the surrounding `with` clause. In a Creol method, the predefined `caller` parameter gives access to the caller object, allowing type-correct call-backs, assuming `caller` supports the cointerface. In what follows, we give the operational semantics for these Creol statements that we extend with probabilistic features.

The operational semantics of Creol is given in rewrite logic (see [12]) and its implementation is executable through Maude [9]. Following the Actor model [2], the system configuration of a Creol model is a multiset of objects, classes and messages, where the objects concurrently execute (remaining parts of) local method activations (called *processes*), and communicate by means of asynchronous message passing. We use empty syntax to denote the associative and commutative multiset concatenation operator. At each execution step, the Creol model makes a transition from one configuration to another, which results from all possible local transitions between its subconfigurations. Local transitions of the model are therefore expressed by conditional rewrite rules of the form:

$$\text{subconfiguration}_1 \;\longrightarrow\; \text{subconfiguration}_2 \quad \textbf{if} \quad \text{condition}.$$

When defining the operational semantics of Creol by means of rewrite theories, *objects* are denoted by constructs of the form

$$< O : Ob \mid \mathit{Cl},\ \mathit{Pr},\ \mathit{PrQ},\ \mathit{Att},\ \mathit{Lvar},\ \mathit{EvQ},\ \mathit{Lcnt} >$$

where O is the object's identifier of sort `Oid`, *Cl* its corresponding class, *Pr* its active process code, *PrQ* a multiset of pending processes, *Att* the object's attributes, *Lvar* the local variables including method parameters, *EvQ* a multiset of unprocessed messages, and *Lcnt* a counter used to generated label values identifying method calls. Creol *classes* are defined by terms of the form

$$< C : \mathit{Cl} \mid \mathit{Par},\ \mathit{Att},\ \mathit{init},\ \mathit{Mtds},\ \mathit{Ocnt} >$$

where C is the class identifier, *Par* is the list of class parameters, *Att* is the list of attributes, *init* contains the Creol code for the *constructor* of class C, and *Mtds* is the multiset of class methods, including *run*, a special method that is automatically executed after the class constructor. Also, *Ocnt* gives the current number of instances of class C. Messages sent between objects are the asynchronous *invocation* and *completion* messages with the syntax *invoc*(o_1, o_2, m, ℓ, in) and *comp*(o_1, ℓ, out), respectively, where ℓ is a label. The meaning of such a pair of messages is that object o_1 calls method m of object o_2, with arguments *in*, and the result is stored in the *out* parameter of the completion message. We omit the object o_1 sending or receiving a message, whenever it is understood from the context.

The semantics for the nondeterministic choice operator is given through the following conditional rewrite rule, using the commutativity and associativity properties of this operator,

$$\begin{array}{l}
< O : Ob \;\mid Pr : (\textsc{s}_1 \,\Box\, \textsc{s}_2)\,;\, \textsc{s}_3,\ PrQ : \textsc{q},\ Lvar : \textsc{l},\ Att : \textsc{a} > \\
\longrightarrow \\
< O : Ob \;\mid Pr : \textsc{s}_1\,;\, \textsc{s}_3,\ PrQ : \textsc{q},\ Lvar : \textsc{l},\ Att : \textsc{a} > \\
\textbf{if}\ ready(\textsc{s}_1,\ (\textsc{a}\ ;\ \textsc{l}),\ \textsc{q})
\end{array}$$

where *O* is an arbitrary object and *ready* is a predicate whose value tells whether the given process $\textsc{s}_1$ is ready for execution, in the context of the variable bindings ($\textsc{a}$; $\textsc{l}$) and the object's multiset of pending processes $\textsc{q}$. Note that, when giving the semantics by rewrite rules, we omit irrelevant attributes in the style of Full Maude [9]. The above rewrite rule therefore applies to instances *O* of any class, since the class attribute *Cl* is omitted.

The operational semantics for asynchronous *invocation* messages is given by

$$\begin{array}{l}
< O : Ob \mid Pr : t!x.m(\textsc{e});\ \textsc{s},\ Lvar : \textsc{l},\ Att : \textsc{a},\ Lcnt : n > \\
\longrightarrow \\
< O : Ob \mid Pr : t := n;\ \textsc{s},\ Lvar : \textsc{l},\ Att : \textsc{a},\ Lcnt : next(n) > \\
invoc(eval(\textsc{x},\ (\textsc{a};\ \textsc{l})),\ m,\ (O,\ n,\ eval(\textsc{e},\ (\textsc{a};\ \textsc{l}))))
\end{array}$$

where *n* is the label value used to identify the future variable *t* and where *eval* is a function used for evaluating expressions. A separate rule takes the invocation message into the process queue *PrQ* of the called object. Similarly, the semantics for asynchronous *completion* messages is given through the rewrite rule

$$\begin{array}{l}
< O : Ob \mid Pr : return(\textsc{v});\ \textsc{s},\ Lvar : \textsc{l},\ Att : \textsc{a} > \\
\longrightarrow \\
< O : Ob \mid Pr : \textsc{s},\ Lvar : \textsc{l},\ Att : \textsc{a} > \\
comp(eval((\texttt{caller},\ \texttt{label},\ \textsc{v}),\ (\textsc{a};\ \textsc{l})))
\end{array}$$

which adds a new completion message to the current configuration. This message only appears in the right-hand side of the rule, but not in its left-hand side, to specify the "production" of a new term that is added to the configuration. The `caller` and `label` are reserved formal parameter names (bound in $\textsc{l}$) referring to the caller and label values of the call, and $\textsc{v}$ contains the formal return values. A separate rule takes a completion message *comp(O, n, out)* into the event queue *EvQ* of the calling object *O*, thereby enabling guards on a label with value *n*.

4 Syntax and Semantics of PCreol

This section provides an overview of the main probabilistic features that extend the syntax and semantics of Creol. We start with the `random` statement, which allows us to specify random values in a Creol model. Then we provide the operational semantics for a probabilistic choice operator together with some possible generalizations, followed by the description of random assignment statements, and the operational semantics for probabilistic lossy communication.

Modeling random numbers. We introduce a `random` statement, which models uniformly distributed random numbers in the unit interval, i.e., such that all values in $[0, 1)$ have an equal chance to be sampled. This is implemented in the interpreter by a rewrite rule that generates fresh pseudo-random numbers with each use of the `random` command.

Probabilistic choice operator. We first consider adding an infix *probabilistic choice* operator $\Box_p$ to the syntactic category of Creol statements. This operator has the syntax $\textsc{s}_1 \Box_p \textsc{s}_2$ where $p \in [0, 1]$ is a fixed real value in the unit interval, and $\textsc{s}_1$, $\textsc{s}_2$ are two arbitrary *lists* of statements. The informal semantics of $\textsc{s}_1 \Box_p \textsc{s}_2$ is that, whenever it is encountered throughout the control flow, the list of statements $\textsc{s}_1$ is selected for execution with probability p, while $\textsc{s}_2$ is selected with probability $1 - p$. However each list of statements is executed provided that it is *ready*, i.e. if its corresponding process may be woken up, which is checked through the *ready* predicate.

Denote by BERNOULLI(p) the Bernoulli discrete probability distribution with parameter p, that samples the value `true` with probability p, and `false` with probability $1 - p$. If both statements in the probabilistic choice are ready for execution, the formal semantics for the $\Box_p$ operator is given by the following probabilistic conditional rewrite rule:

$$\begin{array}{l}
\texttt{<}\ O : Ob \mid Pr : (\textsc{s}_1 \Box_p \textsc{s}_2) \,;\, \textsc{s}_3,\ Lvar : \textsc{l},\ Att : \textsc{a},\ EvQ : \textsc{q}\ \texttt{>} \\
\longrightarrow \\
\textbf{if } B \textbf{ then} \\
\quad \texttt{<}\ O : Ob \mid Pr : \textsc{s}_1 \,;\, \textsc{s}_3,\ Lvar : \textsc{l},\ Att : \textsc{a},\ EvQ : \textsc{q}\ \texttt{>} \\
\textbf{else} \\
\quad \texttt{<}\ O : Ob \mid Pr : \textsc{s}_2 \,;\, \textsc{s}_3,\ Lvar : \textsc{l},\ Att : \textsc{a},\ EvQ : \textsc{q}\ \texttt{>} \\
\textbf{fi} \\
\textbf{if } \mathit{ready}(\textsc{s}_1,\ (\textsc{a}\ ;\ \textsc{l}),\ \textsc{q}) \textbf{ and } \mathit{ready}(\textsc{s}_2,\ (\textsc{a}\ ;\ \textsc{l}),\ \textsc{q}) \\
\textbf{with probability } B := \text{BERNOULLI}(p)
\end{array}$$

When only one of the statements is ready for execution, this statement is automatically selected and the suspended one is dropped. This is achieved by simplification with respect to the conditional equations

$$\begin{array}{l}
\texttt{<}\ O : Ob \mid Pr : (\textsc{s}_1 \Box_p \textsc{s}_2) \,;\, \textsc{s}_3,\ Lvar : \textsc{l},\ Att : \textsc{a},\ EvQ : \textsc{q}\ \texttt{>} \\
= \\
\texttt{<}\ O : Ob \mid Pr : \textsc{s}_1 \,;\, \textsc{s}_3,\ Lvar : \textsc{l},\ Att : \textsc{a},\ EvQ : \textsc{q}\ \texttt{>} \\
\textbf{if } \mathit{ready}(\textsc{s}_1,\ (\textsc{a}\ ;\ \textsc{l}),\ \textsc{q}) \textbf{ and not}(\mathit{ready}(\textsc{s}_2,\ (\textsc{a}\ ;\ \textsc{l}),\ \textsc{q}))
\end{array}$$

for the case when only the first statement is ready, and

$$\begin{array}{l}
\texttt{<}\ O : Ob \mid Pr : (\textsc{s}_1 \Box_p \textsc{s}_2) \,;\, \textsc{s}_3,\ Lvar : \textsc{l},\ Att : \textsc{a},\ EvQ : \textsc{q}\ \texttt{>} \\
= \\
\texttt{<}\ O : Ob \mid Pr : \textsc{s}_2 \,;\, \textsc{s}_3,\ Lvar : \textsc{l},\ Att : \textsc{a},\ EvQ : \textsc{q}\ \texttt{>} \\
\textbf{if not}(\mathit{ready}(\textsc{s}_1,\ (\textsc{a}\ ;\ \textsc{l}),\ \textsc{q})) \textbf{ and } \mathit{ready}(\textsc{s}_2,\ (\textsc{a}\ ;\ \textsc{l}),\ \textsc{q})
\end{array}$$

when only the second statement is ready. The case when neither one of the statements is ready for execution is not handled, neither through conditional rewrite rules nor via conditional equations. Hence, a probabilistic choice is only made as soon as at least one of the statements becomes ready for execution.

In the following, we consider generalizing the probabilistic choice operator, motivated by the need to naturally express random selection of a statement list from a set of statement lists, and refer to the technical report [8] for the semantics of these generalized operators. For example, in order to randomly choose between four assignments $x := 3$, $x := 5$, $x := 7$ and $x := 11$, each with an equal chance of being selected, the binary probabilistic choice operator can be used as follows:

$$x := 3 \quad \Box_{1/4} \quad (x := 5 \quad \Box_{1/3} \quad (x := 7 \quad \Box_{1/2} \quad x := 11)). \tag{2}$$

However, this does not naturally express the fact that the four assignments are selected for execution with the same probability. Instead, the probabilities 1/4, 1/3 and 1/2 in (2) need to be *derived* from the uniform distribution:

$$\begin{pmatrix} 3 & 5 & 7 & 11 \\ 1/4 & 1/4 & 1/4 & 1/4 \end{pmatrix}.$$

A more natural solution is to consider a mixfix *uniform probabilistic choice* operator $\Box_u$ that takes as input a variable number of statement lists and selects either one of them for execution, each with equal probability. Thus, the fair selection statement (2) can more easily be expressed using the $\Box_u$ operator as:

$$x := 3 \quad \Box_u \quad x := 5 \quad \Box_u \quad x := 7 \quad \Box_u \quad x := 11. \tag{3}$$

Furthermore, we may introduce a mixfix *generalized probabilistic choice* operator that takes as input a list of values in $[0, 1]$ summing up to a value in $[0, 1]$, as well as the statement lists whose probabilities of being selected are given by these values. Therefore, the binary probabilistic choice operator is generalized to $n \geq 3$ statements, with the syntax

$$\mathrm{s}_1 \ \Box_{p_1} \ \mathrm{s}_2 \ \Box_{p_2} \ \ldots \ \mathrm{s}_{n-1} \ \Box_{p_{n-1}} \ \mathrm{s}_n \tag{4}$$

for values $p_1, p_2, \ldots, p_{n-1} \in [0, 1]$ such that $\sum_{i=1}^{n-1} p_i \leq 1$. The informal semantics for this operator is a natural generalization both of the binary case $n = 2$ and of the case of uniform random selection, when $p_1 = p_2 = \ldots = p_{n-1} = 1/n$. Therefore, whenever an expression of the form (4) is encountered in the control flow, the statement list s_i is selected with probability p_i, for each $i \in \{1, 2, \ldots, n-1\}$, provided that it is ready for execution. The last statement list s_n is selected for execution with probability $1 - \sum_{i=1}^{n-1} p_i$. Note that using parentheses in (4) to put together two statement lists may cause the binary probabilistic choice operator to be used instead. Therefore, we recommend that an expression involving the generalized probabilistic choice operator should contain no parentheses.

As an example, consider the problem of expressing the random selection from the four assignments considered before $x := 3$, $x := 5$, $x := 7$ and $x := 11$, where this time the probabilities for assigning each of the given values to the variable x are given by the following non-uniform distribution:

$$\begin{pmatrix} 3 & 5 & 7 & 11 \\ 1/6 & 1/3 & 1/6 & 1/3 \end{pmatrix}.$$

In this case, the generalized probabilistic choice operator can be used as follows:

$$x := 3 \quad \Box_{1/6} \quad x := 5 \quad \Box_{1/3} \quad x := 7 \quad \Box_{1/6} \quad x := 11. \tag{5}$$

Random assignment. We add an *uniform random assignment* operator with the syntax

$$x := \text{random}([e_1,\, e_2,\, \ldots,\, e_n]),$$

that randomly selects an expression from the list $E = [e_1,\, e_2,\, \ldots,\, e_n]$ and assigns it to the specified variable x, where each expression in E has an equal chance of being selected. Thus, the fair selection statement (3) using the uniform probabilistic choice operator $\Box_u$ can be more easily expressed as:

$$x := \text{random}([3,\, 5,\, 7,\, 11]).$$

This operator may also be generalized, by considering arbitrary, possibly non-uniform distributions over the list E. The syntax for this *generalized random assignment* is

$$x := \text{random}([e_1,\, e_2,\, \ldots,\, e_n], [p_1,\, p_2,\, \ldots,\, p_n]),$$

where $p_i \in [0,1]$ denotes the probability of assigning e_i to x, for each $i \in \{1, 2, \ldots, n\}$. Also, to satisfy the axioms of probability theory, it must also be the case that $\sum_{i=1}^{n} p_i = 1$. Similar to uniform random assignment, we can use the generalized random assignment in order to express the non-uniform random assignment (5) in a more compact way:

$$x := \text{random}([3,\, 5,\, 7,\, 11], [1/6,\, 1/3,\, 1/6,\, 1/3]).$$

Due to space constraints, we refer to [8] for the operational semantics of these random assignment commands.

Lossy communication. In order to specify lossy inter-object communication, the semantics for the *invocation* and *completion* messages need to be extended, by redefining the rewrite rules for lossless communication. We have two cases, depending on the nature of the messages. *Lossy invocation* with $\alpha \in [0,1]$ probability of successful delivery has the following semantics

```
< $O$ : $Ob$ | $Pr$ : $t!x.m(\textsc{e})$; $\textsc{s}$, $Lvar$ : $\textsc{l}$, $Att$ : $\textsc{a}$, $Lab$ : $n$ >
$\longrightarrow$
if $B$ then
  < $O$ : $Ob$ | $Pr$ : $t := n$; $\textsc{s}$, $Lvar$ : $\textsc{l}$, $Att$ : $\textsc{a}$, $Lab$ : $next(n)$ >
  $invoc(eval(\textsc{x}, (\textsc{a}; \textsc{l})), m, (O, n, eval(\textsc{e}, (\textsc{a}; \textsc{l}))))$
else
  < $O$ : $Ob$ | $Pr$ : $\textsc{s}$, $Lvar$ : $\textsc{l}$, $Att$ : $\textsc{a}$, $Lab$ : $n$ >
fi
with probability $B$ := BERNOULLI($\alpha$)
```

where n is a new label used to identify the future variable t. Also, *lossy completion* with $\beta \in [0,1]$ probability of successful message delivery is given by the probabilistic rewrite rule:

```
$< O : Ob \mid Pr : return(\textsc{v}); \textsc{s}, Lvar : \textsc{l}, Att : \textsc{a} >$
$\longrightarrow$
if $B$ then
  $< O : Ob \mid Pr : \textsc{s}, Lvar : \textsc{l}, Att : \textsc{a} >$
  $comp(eval((\texttt{caller}, \texttt{label}, \textsc{v}), (\textsc{a}; \textsc{l})))$
else
  $< O : Ob \mid Pr : \textsc{s}, Lvar : \textsc{l}, Att : \textsc{a} >$
fi
with probability $B := \text{BERNOULLI}(\beta)$
```

5 Examples

A client-server example. Consider modeling the interaction between a client and a server, in the following scenario. The first object that is created is an instance of class `Main`. This instance, in turn, creates a `Server` and a `Client` object. The `Client` class is parameterized by an integer parameter `value` and a `Server` class parameter `s`, representing the server with which the client is going to communicate. The `Server` class has an attribute `v`, storing its current value. As soon as the `Client` object starts running, it executes a `while` loop of 10 iterations. At each iteration the client makes a call to the server's `add_or_sub` method with its argument equal to the `value` parameter of the `Client` class. The server responds either by adding or by subtracting its current value with the `value` parameter sent by the client. A probabilistic choice is made between the two alternatives. In this example we assigned a probability of 0.8 for selecting addition and a probability of $1 - 0.8 = 0.2$ for selecting subtraction. Below we give a complete listing of the PCreol model corresponding to this example:

```
interface Server  begin  with Any  op add_or_sub(in value: Int)  end
interface Client  begin  end

class Server implements Server
begin
  var v : Float
  op init == v := 0
  with Any
    op add_or_sub(in value: Int) == v := v + value □0.8 v := v - value
end

class Client(value : Int, s: Server) implements Client
begin
  var i : Int
  op run ==
    i := 0;
    while i < 10 do  s.add_or_sub(value;); i := i + 1  end
end

class Main
begin
  var s : Server, c : Client
  op init == s := new Server; c := new Client(1, s)
end
```

We use a QuaTEx query asking for the *expected value* of the `Server` object, as soon as the execution terminates. VeStA gives the following answer:

```
Result: 6.14
Running time: 82.344 seconds
States sampled: 15500
```

The value stored by the `Server` object when the execution terminates is equal to the value of a simple random walk over the integers, starting at 0 (the value that the `Server` object is initialized to in its constructor) and taking 10 unit steps, where each step is either taken to the right with probability 0.8 (addition) or to the left with probability 0.2 (subtraction).

We also consider a predicate $G : S \to \{\texttt{false}, \texttt{true}\}$ on the set of states of the model that returns `true` provided that the server's value is above -1 and `false` otherwise. We used VeStA to statistically model check our model against the CSL formula $\mathcal{P}_{\geq 0.9}[\Diamond\ G]$, with the meaning that *the server's value eventually becomes greater than* -1 *with probability greater than* 0.9. Notice that the $\Diamond$ operator is unbounded, i.e., it does not have any time constraints. The result of the statistical model checking is

```
Result: true
Running time: 77.931 seconds
States sampled: 16318
```

which agrees with the intuition that, given a random walk which takes positive steps with probability 0.8 and negative steps with probability 0.2, it is more likely that the value of the random walk after 10 steps is found in the positive part of the interval $[-10, 10]$ than in its negative part.

A simple authentication protocol. We also show how a simplified version of the Needham-Schroeder authentication security protocol [21] can be specified as an object-oriented PCreol model, where messages can be lost with some probability due to a probabilistic environment. We consider a protocol in which two agents–Alice and Bob–must decide on a shared key, and they proceed as follows. Alice generates a new random key K, encrypts it together with her ID tag, using the public key of Bob, and sends it to Bob. (Only Bob is able to decrypt this message.) Bob decrypts the message and learns the shared key K; he also knows that the message came from Alice by looking at the ID tag. Then Bob answers to Alice by sending her the key K together with his ID, encrypted with Alice's public key. Formally, this simple protocol can be described as follows

$$\begin{aligned} A \to B &: \langle A, \{K\}_{\mathrm{pk}(B)}\rangle \\ B \to A &: \langle B, \{K\}_{\mathrm{pk}(A)}\rangle \end{aligned}$$

and its PCreol specification is given below, where the content of all messages is assumed to be of type `Float`, and the variables and methods that come before the `with Any` statement in class `Agent` are private to that class. Notice the use of the `random` command in the `generateKey` method and that of the probabilistic choice operator $\Box_{0.6}$ in the `sendKey` method, modeling a lossy communication channel between Alice and Bob such that messages are lost with probability 0.4.

```
interface Agent
begin with Any
    op generateKey                  // generates a random shared key
    op getPublicKey(out key: Float) // returns the public key of the current agent
    op sendKey(in someAgent: Agent) // sends the encrypted shared key to the given agent
    op receiveMessage(in msg: Float) // processes the message on the communication channel
    op waitMsg                      // waits for a message on the communication channel
end

class Agent(publicID: Float, publicKey: Float) implements Agent
begin
  var secretKey: Float := 0.1568983, // the secret key of this agent
      sharedKey: Float,              // the key to share with another agent
      hasNewMsg: Bool := false,      // true if a new message appears on the channel
      newMsg: Float                  // the new message

  op encrypt(in msg: Float, encKey: Float; out encMsg: Float) ==
    encMsg := ... // an encryption scheme
  op decrypt(in msg: Float; out decMsg: Float) ==
    decMsg := ...  // the corresponding decryption scheme

  with Any
    op generateKey == sharedKey := random
    op getPublicKey(out key: Float) == key := publicKey
    op sendKey(in x: Agent) ==
      var encKey: Float, encMsg: Float;
      x.getPublicKey(; encKey); encrypt(sharedKey, encKey; encMsg);
      x.receiveMessage(encMsg; ) □ 0.6 skip
    op receiveMessage(in msg: Float) == hasNewMsg := true; newMsg := msg
    op waitMsg ==                  // listen on the channel and wait for a new message
      await hasNewMsg;             // (explicit processor release point)
      decrypt(newMsg; sharedKey); // decrypt the message and store it as the shared key
      hasNewMsg := false           // set the flag to wait for another message
end

class Main
begin
  op run ==
    var a: Agent, b: Agent; a := new Agent(1, 0.67125); b := new Agent(2, 0.58769);
                  // create two agents with some public keys
    a.generateKey(;); // a generates a random key to share with b
    !b.waitMsg();     // b waits for a message from a (asynchronous method call)
    a.sendKey(b;);    // a sends the encrypted key to b
    b.sendKey(a;)     // b sends back the encrypted key to a
       // this method call does not execute before b receives a message from a,
       // due to the await statement in the "waitMsg" method
end
```

Notice that the execution of the above model terminates with probability 0.6 due to the probabilistic choice $\Box_{0.6}$ in method `sendKey` and the processor release point in method `waitMsg`. Indeed, we used VeStA to statistically model check our model against the formula $\mathcal{P}_{\geq 0.5}[\Diamond\ T]$ meaning that *the execution eventually terminates with probability greater than* 0.5 (T is a predicate for checking termination), and we obtained the following result, with a confidence of 99.5%:

```
Result: true
Running time: 3450.891 seconds
States sampled: 160106
```

6 Implementation of the PCreol Interpreter

This section describes the Maude implementation of the PCreol operational semantics, which is available for download from [7] and can be executed using the latest versions of Maude. We use a refinement technique based on the stochastic time model introduced in [3] to resolve all nondeterminism in the interpreter and allow the VeStA tool to statistically model check and analyze quantitative properties of PCreol models.

Our prototype PCreol interpreter is implemented on top of the current one for Creol, and it allows us to test part of the features described in Section 4, namely modeling random numbers, binary probabilistic choice and lossy communication (invocation, completion). The probabilistic rewrite theories framework that we use allows us to formulate a natural probabilistic extension to Creol's operational semantics, in which all of Creol's operational rules are kept the same, without the need to translate them. In this respect, the implementation of our probabilistic extension to Creol can be seen as a *patch* to the Creol interpreter, making it easy to keep in sync with the latest version of Creol.

The following paragraphs give precise meaning to the mechanism that we use to schedule the execution of PCreol objects, which is essentially the same as for scheduling objects in actor PMaude [3]. As in Creol, each object has a sequence of statements to execute, running on its own processor. The execution traces of the concurrent system are represented by interleavings of executions performed by different objects. Standard model checking goes through all possible interleavings of the concurrent model, causing state space explosion. One possible solution to this problem is to obtain a series of random interleavings by discrete-event simulation. These can then be used in statistical model checking and quantitative analysis algorithms, as implemented in VeStA. It can easily be shown that a sufficient condition for the state space corresponding to a concurrent object-based system to be checked in a fair manner using statistical model checking is that the waiting times of all objects must be exponentially distributed with a fixed rate parameter.

We briefly give the implementation details for this stochastic time model. First, we make `Float` a subsort of the configuration, to be able to explicitly specify *time* as part of the global system state. We then define *execution marks* as terms of sort `ExecMark`, introduced as a subsort of the configuration:

```
subsort ExecMark < Configuration .
op execute(_) : Oid -> ExecMark .
```

To control the execution of objects in PCreol, we introduce execution marks, which identify the currently *active* object in the system configuration. An object is active if it has an associated execution mark and has at least one enabled rewrite rule. In the operational semantics, execution marks will now be part of the left-hand side of each rewrite rule. The execution of non-active objects is controlled by *scheduled execution marks*, which include a time parameter indicating when an object should become active:

```
subsort ScheduledExecMark < Configuration .
op [_,_] : Float ExecMark -> ScheduledExecMark .
```

By randomly assigning time values to each scheduled execution marks, using a continuous probability distribution, we ensure that only one object can execute at a given time. In this way, non-determinism is resolved in the operational semantics. Finally we add a *tick* operation that makes the system evolve by unwrapping the scheduled execution marks into unscheduled ones and rendering exactly one object active

```
op tick : Config -> Config .
```

where `Config` is a sort whose terms are obtained from terms of sort `Configuration` by adding a pair of curly brackets[2]:

```
op {_} : Configuration -> Config [ctor] .
```

This is to ensure compatibility with the VeStA tool. The semantics of the *tick* operation is the same as in the actor PMaude model [3], selecting the next object for execution in chronological order[3]:

```
op tickAux : Float ExecMark Configuration -> Config .

var CF : Configuration .
vars T1 : Float .
vars E E1 : ExecMark .

eq tick([T, E] CF) = tickAux(T, E, CF) .
eq tick(CF) = CF [owise] .

ceq tickAux(T, E, [T1, E1] CF) = tickAux(T1, E1, [T, E] CF) if T1 < T .
eq tickAux(T, E, CF T1) = E CF T [owise] .
```

To ensure compatibility with VeStA, we also add an initial state term, giving the state in which the PCreol model is initially found:

```
op initState : -> Config .
```

The `initState` term needs to be defined for each particular PCreol model.

The execution mark technique that we use can be described as follows. In order to resolve all nondeterminism, we adjust the implementation of the original Creol interpreter by adding execution marks in the left-hand side of each rewrite rule. For example, the rewrite rule giving the operational semantics for the *skip* statement is changed from

```
rl < O : C | Att: S, Pr: {L | skip; SL}, PrQ: W, Lcnt: F >
 => < O : C | Att: S, Pr: {L | SL}, PrQ: W, Lcnt: F > .
```

to

```
rl < O : C | Att: S, Pr: {L | skip; SL}, PrQ: W, Lcnt: F >  execute(O) T
 => < O : C | Att: S, Pr: {L | SL}, PrQ: W, Lcnt: F >
    [T + sampleExpWithRate(1.0), execute(O)] T .
```

[2] The `ctor` attribute is used to specify that the operator `{_}` is a constructor, i.e., an operator appearing in the normal forms of the term algebra.

[3] The `owise` attribute added to an equation specifies that the equation should be applied in all other cases in which the rest of the (possibly conditional) equations cannot apply.

by adding an execution mark and also the global time to the left-hand side of the rule, as well as adding a scheduled execution mark and the global time to its right-hand side. This makes the new subconfiguration (and the object `O`) active at a later time, after a random interval of time has passed, and represents the main idea of the execution mark technique.

In all updated rules the delay follows an exponential probability distribution with a fixed rate parameter of 1. The random time interval is generated using the `sampleExpWithRate` operation in Maude [9]. The other rewrite rules are adjusted in a similar manner. Note that in the current implementation of the PCreol interpreter the rates corresponding to the waiting times of all scheduled execution marks are equal to 1. However, we may provide an implementation in which the rates depend on the object with which they are associated, thus modeling the fact that different components have different processor speeds.

To fully integrate the implementation of the PCreol interpreter with the VeStA tool, we also add an operation giving the current time of the global configuration:

```
op getTime : Config -> Float .
eq getTime(T CF:Configuration) = T .
```

Furthermore, we define the predicates and valuations that map the current configuration of a PCreol model into a Boolean value and a floating point value, respectively. Thus, consider a set of atomic propositions $AP = \{ \mathrm{sat}_0, \mathrm{sat}_1, \ldots, \mathrm{sat}_n \}$ and a labeling function $L : S \to 2^{AP}$ mapping each state $s \in S$ of the PCreol model into the subset of atomic propositions $L(s) \subseteq AP$ that are true in s. This labeling function is used in the statistical model checking process of VeStA and is implemented by an operation

```
op sat : Nat Config -> Bool .
```

from which $L(s)$ is obtained by constructing the set $\{ \mathrm{sat}_i \mid \texttt{sat(i, s) == true} \}$. Also, let $V_1, V_2, \ldots, V_k : S \to \mathbb{R}$ be a set of $k \geq 1$ valuation functions. These are implemented in the PCreol interpreter by means of an operation

```
op val : Nat Config -> Float .
```

from which $V_i(s)$ is obtained as `val(i, s)`, for all $1 \leq i \leq k$.

Currently, after compiling a PCreol model into its corresponding Maude specification, the predicates, as well as the valuation functions need to be manually added to the compiled code and explicitly defined by the user to be able to use VeStA and determine whether the PCreol model satisfies the desired properties.

7 Conclusions and Future Work

The main contribution of this paper is to introduce PCreol, an extension of the Creol object-oriented modeling language which allows the specification of probabilistic open distributed systems, and to provide the Maude implementation of a prototype interpreter for PCreol models. We used the semantic framework of probabilistic rewrite theories to define the operational semantics of PCreol. Our implementation allowed us to integrate PCreol with the VeStA tool for statistical model checking and quantitative analysis, to be able to check properties of PCreol models, as well as to extract desired quantitative information from them.

We have integrated PCreol with VeStA by refining the implementation of the Creol interpreter using the stochastic time model introduced in [3] for actor PMaude modules. This technique can also be used to adapt interpreters of the recent ABS modeling language [14], which is in many ways similar to Creol. We may also attempt to integrate PCreol with the recent parallelization of the VeStA tool called PVeStA [4]. Also, as shown in our examples, the statistical tests that VeStA uses have large running times and require a large number of samples. This is because VeStA is designed to handle complex property checking, e.g., nested operators, whereas our properties are in general simpler. It is therefore worth investigating other, more suitable and more efficient approaches as in [16].

Another possible direction for future work is the exact (non-statistical) verification of probabilistic distributed systems modeled in PCreol. The integration of PCreol with exact probabilistic model checkers like PRISM [18] would be of great value. On a similar note, we may investigate the use of proof systems based on probabilistic extensions of Hoare logic (e.g., [10]) to prove different facts about PCreol models.

Acknowledgements

We thank Cristian Prisacariu for fruitful discussions throughout the writing of this paper. We are also indebted to Rudolf Schlatte for constant support during the development of the PCreol interpreter.

References

1. Ábrahám, E., Grabe, I., Grüner, A., Steffen, M.: Behavioral interface description of an object-oriented language with futures and promises. J. Log. Algebr. Program. 78(7), 491–518 (2009)
2. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge, MA, USA (1986)
3. Agha, G., Meseguer, J., Sen, K.: PMaude: Rewrite-based specification language for probabilistic object systems. Electronic Notes in Theoretical Computer Science 153(2) (2006)
4. Alturki, M., Meseguer, J.: PVeStA: A parallel statistical model checking and quantitative analysis tool. To appear in Proc. CALCO'11 (2011)
5. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Verifying continuous time Markov chains. In: Alur, R., Henzinger, T. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, vol. 1102, pp. 269–276. Springer Berlin / Heidelberg (1996)
6. Baier, C., Ciesinski, F., Grosser, M.: PROBMELA: a modeling language for communicating probabilistic processes. In: Proceedings of The Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. pp. 57 – 66 (2004)

7. Bentea, L., Owe, O.: The implementation of the prototype PCreol interpreter (2010), `http://www.ifi.uio.no/~lucianb/projects/pcreol/`
8. Bentea, L., Owe, O.: Towards an object-oriented modeling language for probabilistic open distributed systems (2010), `http://www.ifi.uio.no/~lucianb/publications/2010/pcreol.pdf`
9. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, LNCS, vol. 4350. Springer (2007)
10. den Hartog, J.: Probabilistic Extensions of Semantical Models. Ph.D. thesis, Vrije Univ., Amsterdam (2002)
11. Holzmann, G.: The Spin model checker: Primer and reference manual. Addison-Wesley Professional, first edn. (2003)
12. Johnsen, E., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and Systems Modeling 6(1), 39–58 (2007)
13. Johnsen, E.B., Blanchette, J.C., Kyas, M., Owe, O.: Intra-object versus inter-object: Concurrency and reasoning in Creol. In: Proc. 2nd Intl. Workshop on Harnessing Theories for Tool Support in Software (TTSS'08). Electronic Notes in Theoretical Computer Science, vol. 243, pp. 89–103. Elsevier (2009)
14. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B., de Boer, F.S., Bonsangue, M.M. (eds.) Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010). Lecture Notes in Computer Science, Springer-Verlag (2011), to appear
15. Johnsen, E.B., Owe, O., Arnestad, M.: Combining active and reactive behavior in concurrent objects. In: Langmyhr, D. (ed.) Proc. of the Norwegian Informatics Conference (NIK'03). pp. 193–204. Tapir Academic Publisher (2003)
16. Kim, M., Stehr, M.O., Talcott, C., Dutt, N., Venkatasubramanian, N.: A probabilistic formal analysis approach to cross layer optimization in distributed embedded systems. In: Proceedings of the 9th IFIP WG 6.1 international conference on Formal methods for open object-based distributed systems. pp. 285–300. FMOODS'07, Springer-Verlag, Berlin, Heidelberg (2007)
17. Kumar, N., Sen, K., Meseguer, J., Agha, G.: Probabilistic rewrite theories: Unifying models, logics and tools. Technical report UIUCDCS-R-2003-2347, Department of Computer Science, University of Illinois at Urbana-Champaign (2003)
18. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic model checking for performance and reliability analysis. ACM SIGMETRICS Performance Evaluation Review 36(4), 40–45 (2009)
19. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
20. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Presicce, F. (ed.) Recent Trends in Algebraic Development Techniques, Lecture Notes in Computer Science, vol. 1376, pp. 18–61. Springer Berlin / Heidelberg (1998)
21. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Commun. ACM 21, 993–999 (1978)
22. Sen, K., Viswanathan, M., Agha, G.A.: VESTA: A statistical model-checker and analyzer for probabilistic systems. In: Proceedings of The Second Internationl Conference on the Quantitative Evaluation of Systems (QEST'05). pp. 251–252. IEEE Computer Society (2005)

A Formal Model of User-Defined Resources in Resource-Restricted Deployment Scenarios [⋆]

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

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

Abstract. Software today is often developed for deployment on varying architectures. In order to model and analyze the consequences of such deployment choices at an early stage in software development, it seems desirable to capture aspects of low-level deployment concerns in high-level models. In this paper, we propose an integration of a generic cost model for resource consumption with deployment components in Timed ABS, an abstract behavioral specification language for executable object-oriented models. The actual cost model may be user-defined and specified by means of annotations in the executable Timed ABS model, and can be used to capture specific resource requirements such as processing capacity or memory usage. Architectural variations are specified by resource-restricted deployment scenarios with different capacities. For this purpose, the models have deployment components which are parametric in their assigned resources. The approach is demonstrated on an example of multimedia processing servers with a user-defined cost model for memory usage. We use our simulation tool to analyze deadline misses for given usage and deployment scenarios.

1 Introduction

Software systems often need to adapt to different deployment scenarios: *operating systems* adapt to different hardware, e.g., the number of processors; *virtualized applications* are deployed on varying (virtual) servers; and *services on the cloud* need to adapt dynamically to the underlying infrastructure. Such adaptability raises new challenges for the modeling and analysis of component-based systems.

In general, abstraction is seen as a means to reduce complexity in a model [21]. In formal methods, the ability to execute abstract models was initially considered counter-productive because the models would become less abstract [13,15], but recently abstract executable models have gained substantial attention and also been applied industrially in many different domains [30]. Specification languages range from design oriented languages like UML, which are concerned with structural models of architectural deployment, to programming language close specification languages such as JML [6], which are best suited to express

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

functional properties. *Abstract executable modeling languages* are found in between these two abstraction levels, and appear as the appropriate abstraction level to express deployment decisions because they abstract from concrete data structures in terms of abstract data types, yet allow the faithful specification of a system's control flow and data manipulation. For the kind of properties we are considering in this paper, it is paramount that the models are indeed executable in order to have a reasonable relationship to the final code.

The abstract behavioral specification language ABS [7, 17] is such an executable modeling language with a formally defined semantics and a simulator built on the Maude platform [8]. ABS is an object-oriented language in which concurrent objects communicate by asynchronous method calls and in which different activities in an object are cooperatively scheduled. In recent work, we have extended ABS with time and with a notion of *deployment component* in order to abstractly capture resource restrictions related to deployment decisions at the modeling level. This allows us to observe, by means of simulations, the performance of a system model ranging over the amount of resources assigned to the deployment components, with respect to execution capacity [19, 20] and with respect to memory [2]. This way, the modeler gains insight into the resource requirements of a component, in order to provide a minimum response time for given client behaviors. In the approach of these papers, the resource consumption of the model was fixed by the ABS simulator (or derived by the COSTA tool [1]), so the only parameter which could be controlled by the modeler was the capacity of the deployment components.

In this paper, we take a more high-level approach to the modeling of resource usage and consider a generic cost model $\mathcal{M}$. We propose a way for the modeler to explicitly associate resource costs to different activities in a model via optional annotations in the language. This allows simulations of performance at an early stage in the system design, by further abstraction from the control flow and data structures of the system under development. Resource usage according to $\mathcal{M}$ may be specified abstractly (e.g., for whole methods), or more fine-grained and following the control flow in parts of the model which are of particular interest; it can also be refined along with the model. Resource annotations are specified by means of user-defined expressions; e.g., depending on the input parameters to a method in the model. Given a model with explicit resource annotations, we show how our simulation tool for Timed ABS may be used for abstract performance analysis of formal object-oriented models, to analyze the performance of the model depending on the resources assigned to the deployment components. The proposed approach and associated tool are illustrated on an example of a multimedia processing service with a cost model for memory allocation.

Paper overview. Section 2 introduces the Timed ABS modeling language, Section 3 presents the semantics of Timed ABS with user-defined resource annotations. Section 4 presents an application and simulation results of a photo and video processing service in Timed ABS. Section 5 discusses related work and Section 6 concludes the paper.

2 Models of Deployed Concurrent Objects in Timed ABS

Timed ABS is an abstract behavioral specification language for distributed concurrent objects, a successor and extension of the Creol language used in some of our previous work. Characteristic features of Timed ABS are that: (1) it allows abstracting from implementation details while remaining executable; i.e., a *functional sub-language* over abstract data types is used to specify internal, sequential computations [17]; (2) it provides *flexible concurrency and synchronization mechanisms* by means of asynchronous method calls, release points in method definitions, and cooperative scheduling of method activations [9,18]; (3) it supports *user-provided deadlines* to method calls to express local QoS requirements [10]; and (4) it features *deployment components* with parametric resources to model deployment variability [2,19,20]. Compared to previous work on deployment components, we here extend the syntax and semantics of Timed ABS with *user-defined annotations* to express general cost-models for resource usage.

A Timed ABS *model* P defines interfaces, classes, data types, and functions, and has a main block $\{\overline{T}\ \overline{x}; s\}$ to configure the initial state. Objects are dynamically created instances of classes; their declared attributes are initialized to arbitrary type-correct values, but may be redefined in an optional method *init*. This paper assumes that models are well-typed, so method binding is guaranteed to succeed [17]. Intuitively, concurrent objects in Timed ABS have dedicated processors and live in a distributed environment with asynchronous and unordered communication. All communication is between named objects, typed by interfaces, by means of asynchronous method calls. (There is no remote field access.) Calls are asynchronous as the caller may decide at runtime when to synchronize with the reply from a call. Method calls may be seen as triggers of concurrent activity, spawning new activities (so-called *processes*) in the called object. 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* in a process pool. Process scheduling is non-deterministic, but controlled by *processor release points* in a cooperative way. *Deployment components* restrict the natural concurrency of objects in Timed ABS by introducing resource-restricted execution contexts to capture different deployment scenarios. Every object in Timed ABS is associated with one deployment component.

The functional level of Timed ABS defines user-defined parametric datatypes and functions, as shown in Fig. 1. The ground types T consist of basic types B (such as Bool, Int, and Resource), as well as names D for datatypes and I for interfaces. In general, a type A may also contain type variables N (i.e., uninterpreted type names [26]). In *datatype declarations* Dd, a datatype D has a set of constructors $Cons$, which have a name Co and a list of types $\overline{A}$ for their arguments. *Function declarations* F have a return type A, a function name fn, a list of parameters $\overline{x}$ of types $\overline{A}$, and a function body e. *Expressions* e include Boolean expressions b, variables x, values v, the self-identifier **this**, constructor expressions $Co(\overline{e})$, function expressions $fn(\overline{e})$, and case expressions

$$
\begin{array}{ll}
\textit{Syntactic categories.} & \textit{Definitions.} \\
T \text{ in Ground Type} & T ::= B \mid I \mid D \mid D\langle \overline{T} \rangle \\
A \text{ in Type} & A ::= N \mid T \mid N\langle \overline{A} \rangle \\
x \text{ in Variable} & Dd ::= \texttt{\textbf{data}}\ D[\langle \overline{A} \rangle] = [\overline{Cons}]; \\
e \text{ in Expression} & Cons ::= Co[(\overline{A})] \\
b \text{ in Bool Expression} & F ::= \texttt{\textbf{def}}\ A\ fn[\langle \overline{A} \rangle](\overline{A\ x}) = e; \\
v \text{ in Value} & e ::= b \mid x \mid v \mid Co[(\overline{e})] \mid fn(\overline{e}) \mid \texttt{\textbf{case}}\ e\ \{\overline{br}\} \\
br \text{ in Branch} & \qquad \mid \texttt{\textbf{this}} \mid \texttt{\textbf{thiscomp}} \mid \texttt{\textbf{deadline}} \mid \texttt{\textbf{now}} \\
p \text{ in Pattern} & v ::= Co[(\overline{v})] \mid \texttt{\textbf{null}} \\
 & br ::= p \Rightarrow e; \\
 & p ::= \_ \mid x \mid v \mid Co[(\overline{p})]
\end{array}
$$

Fig. 1. Syntax for the functional level of Timed ABS. Terms $\overline{e}$ and $\overline{x}$ denote possibly empty lists over the corresponding syntactic categories, and square brackets [ ] optional elements.

$$
\begin{array}{ll}
\textit{Syntactic categories.} & \textit{Definitions.} \\
C, I, m \text{ in Name} & P ::= \overline{Dd}\ \overline{F}\ \overline{IF}\ \overline{CL}\ \{\overline{T\ x};\ s\} \\
g \text{ in Guard} & IF ::= \texttt{\textbf{interface}}\ I\ \{\ [\overline{Sg}]\ \} \\
s \text{ in Stmt} & CL ::= [\texttt{[}an\texttt{]}]\ \texttt{\textbf{class}}\ C\ [(\overline{T\ x})]\ [\texttt{\textbf{implements}}\ \overline{I}]\ \{\ [\overline{T\ x};]\ \overline{M}\} \\
an \text{ in Annotation} & Sg ::= T\ m\ ([\overline{T\ x}]) \\
 & M ::= [\texttt{[}an\texttt{]}]\ Sg\ \{[\overline{T\ x};]\ s\ \} \\
 & an ::= \texttt{Deadline:}\ e \mid \texttt{Cost:}\ e \mid \texttt{Free:}\ e \\
 & g ::= b \mid x? \mid \texttt{\textbf{duration}}(e) \mid g \wedge g \\
 & s ::= s; s \mid [\texttt{[}an\texttt{]}]\ s \mid \texttt{\textbf{skip}} \mid \texttt{\textbf{if}}\ b\ \{\ s\ \}\ [\texttt{\textbf{else}}\ \{\ s\ \}] \mid \texttt{\textbf{while}}\ b\ \{\ s\ \} \\
 & \qquad \mid x = rhs \mid \texttt{\textbf{suspend}} \mid \texttt{\textbf{await}}\ g \mid \texttt{\textbf{duration}}(d) \mid \texttt{\textbf{return}}\ e \\
 & rhs ::= e \mid \texttt{\textbf{new}}\ C\ (\overline{e})\ [\texttt{\textbf{in}}\ e] \mid \texttt{\textbf{component}}\ (e) \mid e.\texttt{\textbf{get}} \mid o!m(\overline{e})
\end{array}
$$

Fig. 2. Syntax for the concurrent object level of Timed ABS.

`case` e $\{\overline{br}\}$. In Timed ABS, the expression **`deadline`** refers to the *remaining permitted execution time* of the current method activation, which is initially given by a deadline annotation at the method call site or by default. We assume that message transmission is instantaneous, so the deadline expresses the time until a reply is received; i.e., it corresponds to an *end-to-end* deadline. The expression **`now`** returns the current time (explained below) and **`thiscomp`** the deployment component to which the current object is associated. *Values* v are constructors applied to values $Co(\overline{v})$ or **`null`**. *Case expressions* have a list of branches $p \Rightarrow e$, where p is a pattern. Branches are evaluated in the listed order. Patterns include wild cards _, variables x, values v, and constructor patterns $Co(\overline{p})$. For simplicity, operator overloading is not considered.

The concurrent object level of Timed ABS is given in Figure 2. An interface IF has a name I and method signatures Sg. A class CL has a name C, interfaces $\overline{I}$ (specifying types for its instances), class parameters and state variables x of type T, and methods M. (The class *attributes* are both the parameters and state variables.) A signature Sg declares the return type T of a method with name m and formal parameters $\overline{x}$ of types $\overline{T}$. M defines a method with signature Sg, local variables $\overline{x}$ of types $\overline{T}$, and a statement s. Statements may access attributes of the current class, locally defined variables, and the method's formal parameters.

Right hand side expressions rhs include object and component creation, method calls $o!m(\overline{e})$ where o denotes the callee and $\overline{e}$ the actual paramters to the call, future dereferencing x.**get**, and (pure) expressions e. Method calls and future dereferencing are explained in detail below. An object may be created in the current deployment component, written **new** $C(\overline{e})$, or in a named component e, written **new** $C(\overline{e})$ **in** e. Deployment components are created by **component**(e), where e reflects the amount of resources assigned to the component.

Statements are standard for sequential composition $s_1; s_2$, and **skip**, **if**, **while**, and **return** constructs. The statement **suspend** unconditionally releases the processor, suspending the active process. In **await** g, the guard g controls processor release and consists of Boolean conditions b and return tests $x?$ (see below). Just like pure expressions e, guards g are side-effect free. If g evaluates to false, the processor is released and the process *suspended.* When the execution thread is idle, any process pr may be selected from the pool of suspended processes if pr is ready to execute.

Communication in Timed ABS is based on asynchronous method calls, denoted by assignments $x = o!m(\overline{e})$ to future variables x. (Local calls are written **this**$!m(\overline{e})$.) After making an asynchronous call $x = o!m(\overline{e})$, the caller may proceed with its execution without blocking on the method reply. Here o is an object expression, and $\overline{e}$ are (data value or object) expressions providing actual parameter values for the method invocation. A future variable refers to a return value which has yet to be computed. There are two operations on future variables, which control synchronization in Timed ABS. First, the guard **await** $x?$ suspends the active process unless a return to the call associated with x has arrived, allowing other processes in the object to execute. Second, the return value is retrieved by the expression x.**get**, which blocks all execution in the object until the return value is available. Standard usages of asynchronous method calls include the statement sequence $x = o!m(\overline{e})$; $v = x$.**get** which encodes a *blocking call,* abbreviated $v = o.m(\overline{e})$ (often referred to as a synchronous call), and the sequence $x = o!m(\overline{e})$; **await** $x?$; $v = x$.**get** which encodes a non-blocking, *preemptible call,* abbreviated **await** $v = o.m(\overline{e})$. As usual, if the return value of a call is of no interest, the call may be written as a statement $o!m(\overline{e})$. In Timed ABS, it is the decision of the caller whether to call a method synchronously or asynchronously, and when to synchronize on the return value of a call.

Time. In Timed ABS, we work with a discrete time model. The local passage of time is *explicitly expressed* using **duration** statements and **duration** guards. The statement **duration**(e) expresses the passage of e time units, and blocks the whole object. Similarly, the guard **await duration**(e) suspends the current process for e time units. Note that time can also pass during synchronization with a method invocation; this can block one process (via **await** $f?$) or the whole object (via $x = f$.**get**). All other statements (normal assignments, **skip** statements, etc.) do not cause time to pass.

Deployment Components can be understood as resource-restricted execution contexts which allow us to specify and compare different execution environments for Timed ABS models. Deployment components are parametric in the amount of resources they make available to their objects, which makes it easy to compare the behavior of a model under different resource assumptions. Deployment components were originally introduced for processing resources in [20]. In contrast, resources in this paper are general, and it is up to the modeller to define a cost model which fits with the specific targeted resource.

Deployment components are integrated in Timed ABS as follows. Resources are understood as a quantitative measure of cost and modeled by the basic data type Resource which extends the integers with an "unlimited resource" ω. We define addition and subtraction for resources as for integers and by $\omega + n = \omega$ and $\omega - n = \omega$ (for integers n). In Timed ABS, variables x of type Component refer to deployment components and allow deployment components to be dynamically created by the statement `x=`**`component`**`(`e`)`, which assigns a given quantity e of resources to the component referred to by `x`. All objects in a Timed ABS model belong to some deployment component. The number of objects residing on a component may grow dynamically through object creation. The ABS syntax for object creation is therefore extended with an optional clause to specify the targeted deployment component; in the expression **`new`** `C(`e`)` **`in`** `x`, the new `C` object will reside in the component `x`. Objects generated without an **`in`**-clause reside in the same component as their parent object. The behavior of a Timed ABS model which does not statically declare deployment components is captured by a root deployment component with ω resources.

The execution inside a deployment component is restricted by the number of available resources in the component; thus the execution in an object may need to wait for resources to become available. In general, the usage of resources for objects in a deployment component depends on a specific cost model $\mathcal{M}$, which expresses how resources are used during execution. In Timed ABS, cost models are expressed by the user by associating resource usage with the execution of different statements in the model. During the execution of a statement with cost n, the associated component consumes n resources. In a deployment component with ω resources, a transition can always be executed immediately. In a deployment component with less than n assigned resources, the object permanently blocks. Otherwise, the object needs to wait until sufficient resources become available. When time advances, resources which are no longer in use by an object may be returned to its deployment component, as determined by the cost model $\mathcal{M}$. For example, for processor resources, all resources may be returned to the deployment component when time advances. In contrast, for memory resources, the time advance corresponds to deallocating memory cells (e.g., by running a garbage collector), and will return the used stack memory and possibly some portion of the heap. (Remark that the memory deallocated after a transition may be larger than the memory allocated before the transition, which corresponds to transitions which reduce the size of the heap.)

Technically, for a cost model $\mathcal{M}$ and a transition from state t to t', the resources to be consumed by the transition are given by a function $\mathit{cost}_{\mathcal{M}}(t,t')$ and the resources to be returned by time advance are given by a function $\mathit{free}_{\mathcal{M}}(t,t')$. The time advance will return to the deployment component the accumulated number of returned resources, as determined by the sum of $\mathit{free}_{\mathcal{M}}(t,t')$ for all transitions in that component since the last time advance. For example, for processor resources, all resources are available in each time step so $\mathit{cost}_{\mathcal{M}}(t,t') = \mathit{free}_{\mathcal{M}}(t,t')$. For memory resources, $\mathit{free}_{\mathcal{M}}(t,t') = \mathit{cost}_{\mathcal{M}}(t,t') + \mathit{size}(t') - \mathit{size}(t)$ if we denote by $\mathit{size}(t)$ the size of the heap for state t. The cost model $\mathcal{M}$ is determined by the exact definitions of $\mathit{cost}_{\mathcal{M}}(t,t')$ and $\mathit{free}_{\mathcal{M}}(t,t')$ for the statements in the language. In order to give the modeler explicit control over the use of resources, the statements of Timed ABS have zero cost by default, which can be overridden by annotations defining the cost model.

Annotations. In this paper, we propose to extend the syntax of Timed ABS with the following resource and deadline annotations: `Cost:` e, `Free:` e, and `Deadline:` e. Annotations an are optional and may be associated with class and method declarations, as well as with statements. *Deadline annotations* interact with time to reflect soft end-to-end deadlines for method activations; i.e., deadlines may be violated in the model. The annotation `Deadline:` e specifies the relative time before which a method activation should complete its execution. Method calls without annotations get the infinite deadline by default.

Resource annotations interact with time and deployment components to express the resource consumption of a given cost model $\mathcal{M}$. Cost and free annotations have a default value of zero, which means that e.g., the execution of a statement without explicit resource annotations has no effect on the deployment component. For statements s, the annotations `Cost:` e and `Free:` e may be used to override the default resource consumption associated with the execution of s and the resources to be freed after finishing its execution. For class declarations the annotations `Cost:` e and `Free:` e may be used to specify the resource consumption associated with the creation of an object of that class and the resources to be freed after the object creation. For method declarations, the annotation `Cost:` e may be used to override a default resource consumption reflecting the method activation, and `Free:` e may be used to override the default amount of resources to be freed after method activation. The exact semantics can be seen in Fig. 6.

3 Semantics

This section presents the operational semantics of Timed ABS as a transition system in an SOS style [27]. 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 [22]). A run is a possibly nonterminating sequence of rule applications.

$$
\begin{array}{rcl@{\qquad}rcl}
cn & ::= & \epsilon \mid obj \mid comp \mid msg \mid fut \mid cn\ \ cn & s & ::= & \mathbf{\texttt{timer}}(v) \mid \ldots \\
obj & ::= & o(\sigma, pr, q) & v & ::= & o \mid f \mid dc \mid \ldots \\
comp & ::= & dc(av, fr, tot) & pr & ::= & \{\sigma|s\} \mid idle \\
msg & ::= & m(o, \overline{v}, f, d) & \sigma & ::= & x \mapsto v \mid \sigma \circ \sigma \\
fut & ::= & f \mid f(v) & q & ::= & \epsilon \mid pr \mid q \circ q \\
tcn & ::= & \{cn\ \ cl(t)\} & & &
\end{array}
$$

Fig. 3. Runtime syntax; here, o and f are object and future identifiers, d and c are the deadline and cost annotations.

The evaluation rules for the functional level of Timed ABS are standard and have been omitted for brevity.

3.1 Runtime Configurations

The runtime syntax is given in Figure 3. We add identifiers for objects, components, and futures to the values v and an auxiliary statement $\mathbf{\texttt{timer}}(v)$ to the statements s. *Configurations* cn are sets of objects, components, messages, and futures. A *timed configuration* tcn adds a global clock $cl(t)$ to a configuration (where t is a time value). Timed configurations live inside curly brackets; thus, in $\{cn\ cl(t)\}$, the variable cn captures the *entire* system configuration at time t. The associative and commutative union operator on (timed) configurations is denoted by whitespace and the empty configuration by ε. An *object obj* is a term $o(\sigma, pr, q)$ where o is the object's identifier, σ a substitution representing the object's fields, pr a process, and q a *pool of processes.* A *substitution* σ is a mapping from variables x to values v. For substitutions and process pools, concatenation is denoted by $\sigma_1 \circ \sigma_2$ and $q_1 \circ q_2$, respectively. A *process pr* is a structure $\{\sigma|s\}$, where s is a list of statements and σ a substitution representing the local variables, plus *destiny* (storing the future for the process's return value) and *deadline*, assuming no name conflicts. A process with an empty statement list is denoted by *idle*.

A *component comp* is a structure $dc(av, fr, tot)$ where dc is the component's identifier, av the unallocated resources, fr the deallocated resources, and tot the resources initially assigned to the component. An *invocation message msg* is a structure $m(o, \overline{v}, f, d)$, where m is the method name, o the callee, $\overline{v}$ the call's actual parameter values, f the future to which the call's result is returned, and d the provided deadline of the call. A *future fut* is either an identifier f, or a term $f(v)$ with an identifier f and a reply value v. For simplicity, classes are not represented explicitly in the semantics, but may be seen as static tables for object layout and method definitions.

3.2 A Transition System for Timed Configurations

General Expressions. Expressions are evaluated in the context of the attributes a of an object, the local variables l of a process, and a time t. For simplicity, we let *this* and *thiscomp* be appropriately bound attributes, and let *destiny* and

(Skip)
$$\begin{array}{c} o(a,\{l|\texttt{\textbf{skip}};s\},q)\\ \to o(a,\{l|s\},q)\end{array}$$

(Suspend)
$$\begin{array}{c} o(a,\{l|\texttt{\textbf{suspend}};s\},q)\\ \to o(a,\mathrm{idle},\{l|s\}\circ q)\end{array}$$

(Tick)
$$\frac{\mathrm{canAdv}(cn,t)}{\begin{array}{c}\{cn\ cl(t)\}\\ \to \{\mathrm{Adv}(cn)\ cl(t+1)\}\end{array}}$$

(Schedule)
$$\frac{\mathit{ready}(pr,a,cn,t)\quad pr\in q}{\begin{array}{c}\{o(a,\mathrm{idle},q)\ \ cl(t)\ \ cn\}\to\\ \{o(a,pr,(q\setminus pr))\ \ cl(t)\ \ cn\}\end{array}}$$

(Assign1)
$$\frac{v=\llbracket e\rrbracket^{t}_{a\circ l}\quad x\in\ dom(l)}{\begin{array}{c}o(a,\{l|x=e;s\},q)\ \ cl(t)\to\\ o(a,\{l[x\mapsto v]|s\},q)\ \ cl(t)\end{array}}$$

(Assign2)
$$\frac{v=\llbracket e\rrbracket^{t}_{a\circ l}\quad x\in\ dom(a)}{\begin{array}{c}o(a,\{l|x=e;s\},q)\ \ cl(t)\to\\ o(a[x\mapsto v],\{l|s\},q)\ \ cl(t)\end{array}}$$

(Await1)
$$\frac{\llbracket e\rrbracket^{t,cn}_{a\circ l}}{\begin{array}{c}\{o(a,\{l|\texttt{\textbf{await}}\ e;s\},q)\ \ cl(t)\ \ cn\}\\ \to\{o(a,\{l|s\},q)\ \ cl(t)\ \ cn\}\end{array}}$$

(Await2)
$$\frac{\neg\llbracket e\rrbracket^{t,cn}_{a\circ l}}{\begin{array}{c}\{o(a,\{l|\texttt{\textbf{await}}\ e;s\},q)\ \ cl(t)\ \ cn\}\to\\ \{o(a,\{l|\texttt{\textbf{suspend}};\texttt{\textbf{await}}\ e;s\},q)\ \ cl(t)\ \ cn\}\end{array}}$$

(Duration1)
$$\frac{v=\llbracket e\rrbracket^{t}_{a\circ l}}{\begin{array}{c}o(a,\{l|\texttt{\textbf{duration}}(e);s\},q)\ \ cl(t)\\ \to o(a,\{l|\texttt{\textbf{timer}}(v);s\},q)\ \ cl(t)\end{array}}$$

(Timer)
$$\frac{v\le 0}{\begin{array}{c}o(a,\{l|\texttt{\textbf{timer}}(v);s\},q)\\ \to o(a,\{l|s\},q)\end{array}}$$

(Read-Fut)
$$\frac{f=\llbracket e\rrbracket^{t}_{a\circ l}}{\begin{array}{c}o(a,\{l|x=e.\texttt{\textbf{get}};s\},q)\ \ f(v)\ \ cl(t)\\ \to o(a,\{l|x=v;s\},q)\ \ f(v)\ \ cl(t)\end{array}}$$

(Return)
$$\frac{v=\llbracket e\rrbracket^{t}_{a\circ l}\quad f=l(\mathrm{destiny})}{\begin{array}{c}o(a,\{l|\texttt{\textbf{return}}(e);s\},q)\ \ f\ \ cl(t)\\ \to o(a,\{l|s\},q)\ \ f(v)\ \ cl(t)\end{array}}$$

(Cond1)
$$\frac{\llbracket e\rrbracket^{t}_{a\circ l}}{\begin{array}{c}o(a,\{l|\texttt{\textbf{if}}\ e\ \{s_1\}\ \texttt{\textbf{else}}\ \{s_2\};s\},q)\ \ cl(t)\\ \to o(a,\{l|s_1;s\},q)\ \ cl(t)\end{array}}$$

(Cond2)
$$\frac{\neg\llbracket e\rrbracket^{t}_{a\circ l}}{\begin{array}{c}o(a,\{l|\texttt{\textbf{if}}\ e\ \{s_1\}\ \texttt{\textbf{else}}\ \{s_2\};s\},q)\ \ cl(t)\\ \to o(a,\{l|s_2;s\},q)\ \ cl(t)\end{array}}$$

Fig. 4. Non-annotated transitions in the semantics of ABS.

deadline be appropriately bound local variables, assuming no name conflicts. We denote by $\llbracket e\rrbracket^{t}_{\sigma}$ the result of evaluating e in the context $\sigma[\mathit{now}\mapsto t]$. (For simplicity, this evaluation is assumed to converge.)

Evaluating Guards. A guard g is evaluated to determine whether a given process starting with **await** g is ready to be scheduled. Given a substitution σ, a time t, and a configuration cn, we denote by $\llbracket g\rrbracket^{t,cn}_{\sigma}$ an evaluation function which reduces guards g to Boolean values, defined as follows: $\llbracket b\rrbracket^{t,cn}_{\sigma}=\llbracket b\rrbracket^{t}_{\sigma}$, $\llbracket g_1\wedge g_2\rrbracket^{t,cn}_{\sigma}=\llbracket g_1\rrbracket^{t,cn}_{\sigma}\wedge\llbracket g_2\rrbracket^{t,cn}_{\sigma}$, $\llbracket\texttt{\textbf{duration}}(e)\rrbracket^{t,cn}_{\sigma}=\mathrm{true\ iif}\ \llbracket e\rrbracket^{t}_{\sigma}\le 0$, $\llbracket x?\rrbracket^{t,cn}_{\sigma}=$ true if $\llbracket x\rrbracket^{t}_{\sigma}=f$ and $f(v)\in cn$ for some value v, otherwise $f\in cn$ and $\llbracket x?\rrbracket^{t,cn}_{\sigma}=$ false.

Transition rules transform (timed) configurations into new (timed) configurations, and are given in Figs. 4 and 6. Note that we need to pass the clock through all transition rules in which general expressions or guards are evaluated.

We denote by a the substitution which represents the attributes of an object and by l the substitution which represents the local variable bindings of a process. In the semantics, different assignment rules are defined for side effect free expressions (Assign1 and Assign2), object and component creation (New-Object1, New-Object2, and New-Component), method calls (Async-call), and future dereferencing (Read-Fut). Rule Skip consumes a **skip** in the active process. Here and in the sequel, the variable s will match any (possibly empty) statement list. Rules Assign1 and Assign2 assign the value of expression e to a variable x in the local variables l or in the fields a, respectively. Rules Cond1 and Cond2 cover the two cases of conditional statements in the same way. (We omit the rule for **while**, which unfolds into the conditional.)

Scheduling. Two operations manipulate a process pool q; $pr \circ q$ adds a process pr to q and $q \setminus pr$ removes pr from q. If pr is a process, σ a substitution, t a time value, and cn a configuration, we denote by $ready(pr, \sigma, cn, t)$ a predicate which checks if pr is ready to execute (in the sense that the processes will not directly suspend or block the object's processor [18]). Scheduling is captured by the rule Schedule, which applies when the active process is *idle* and schedules some new process pr for execution if it is ready. Note that in order to evaluate guards on futures, the configuration cn is passed to the *ready* function. This explains the use of brackets in the rules, which ensures that cn is bound to the rest of the global system configuration. The same approach is used to evaluate guards in the rules Await1 and Await2 below. Rule Suspend suspends the active process to the process pool, leaving the active process *idle*. Rule Await1 consumes the **await** g statement if g evaluates to true in the current state of the object, rule Await2 adds a suspend statement in order to suspend the process if the guard evaluates to false.

Durations. In rule Duration1, a statement **duration**(e) is reduced to the runtime statement **timer**(v), in which the expressios e have been reduced to a value. This statement blocks execution on the object until the best case execution time v has passed and Timer becomes applicable; i.e., until at least the duration v has passed. Time advance decrements the values of v (see below). Remark that time cannot advance beyond duration v before the statement has been executed.

Time advance. To simplify the logging of resource consumption, we consider a discrete time model in which time always advances by one unit. Rule Tick specifies how time advances in the system. In order to capture timed concurrent behavior with an interleaving semantics, we use a run-to-completion approach in which a transition must occur at the current time if possible. We follow the approach of Real-Time Maude [23, 24] and let time advance uniformly through the configuration cn. Auxiliary functions, defined in Fig. 5, specify the advancement of time and its effect: the predicate *canAdv* determines whether time may advance at the current state, reflecting the run-to-completion approach. The function *Adv*(cn) specifies how the advancement of time affects different parts of the configuration cn. Both *canAdv* and *Adv* have the whole configuration as input but mainly consider objects and deployment components since these exhibit time-dependent behavior. The function *Adv* updates the active and suspended

$$
\begin{array}{ll}
\mathrm{canAdv}(cn', t) = \mathit{true} & \textit{cn' contains no objects or messages}\\
\mathrm{canAdv}(msg\ cn, t) = \mathit{false} & \textit{messages are instantaneous}\\
\mathrm{canAdv}(o(a, \{l|[\mathtt{cost}\!: e, an]\ s\}, q)\ dc(av, fr, tot)\ cn, t) & \textit{not enough resources}\\
\quad = \mathrm{canAdv}(o(a, \{l|[an]\ s\}, q)\ dc(av, fr, tot)\ cn, t) & \textit{(other annotations ignored)}\\
\quad \wedge\, a(\mathrm{thiscomp}) = dc \wedge [\![e]\!]^{t}_{aol} = c \wedge c > av & \\
\mathrm{canAdv}(o(a, \{l|x = e.\mathbf{get}; s\}, q)\ f\ cn, t) & \textit{o is blocked and}\\
\quad = \mathrm{canAdv}(f\ cn, t) \wedge [\![e]\!]^{t}_{aol} = f & \textit{no value is available}\\
\mathrm{canAdv}(o(a, \{l|\mathbf{timer}(v); s\}, q)\ cn, t) & \textit{o must wait}\\
\quad = \mathrm{canAdv}(cn, t) \wedge v > 0 & \\
\mathrm{canAdv}(o(a, \mathrm{idle}, q)\ cn, t) & \textit{no ready processes}\\
\quad = \mathrm{canAdv}(cn, t) \wedge \neg\mathrm{ready}(pr, a, cn, t) \text{ for all } pr \in q & \\
\mathrm{canAdv}(obj\ cn, t) = \mathit{false} & \textit{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{Adv}(o(a, pr, q)\ cn) & = o(a, \mathrm{Adv}_1(pr), \mathrm{Adv}_2(q))\ \mathrm{Adv}(cn) & \\
\mathrm{Adv}(dc(av, fr, tot)\ cn) & = dc(av + fr, 0, tot)\ \mathrm{Adv}(cn) & \\
\mathrm{Adv}(cn) & = cn & \textit{otherwise}
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{Adv}_1(\mathrm{idle}) & = \mathrm{idle}\\
\mathrm{Adv}_1(\{l|[an]s\}) & = \{l[\mathrm{deadline} \mapsto l(\mathrm{deadline}) - 1]|\mathrm{Adv}_3(s)\}
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{Adv}_2(\emptyset) & = \emptyset\\
\mathrm{Adv}_2(\{l|[an]s\} \circ q) & = \{l[\mathrm{deadline} \mapsto l(\mathrm{deadline}) - 1]|[an]\mathrm{Adv}_4(s)\} \circ \mathrm{Adv}_2(q)
\end{array}
$$

$$
\mathrm{Adv}_3(s) = \begin{cases}
\mathbf{timer}(v-1) & \text{if } s = \mathbf{timer}(v)\\
\mathbf{await}\ \mathrm{Adv}_5(g) & \text{if } s = \mathbf{await}\ g\\
\mathrm{Adv}_3(s_1) & \text{if } s = s_1; s_2\\
s & \text{otherwise}
\end{cases}
$$

$$
\mathrm{Adv}_4(s) = \begin{cases}
\mathbf{await}\ \mathrm{Adv}_5(g) & \text{if } s = \mathbf{await}\ g\\
\mathrm{Adv}_4(s_1) & \text{if } s = s_1; s_2\\
s & \text{otherwise}
\end{cases}
$$

$$
\mathrm{Adv}_5(g) = \begin{cases}
\mathrm{Adv}_5(g_1) \wedge \mathrm{Adv}_5(g_2) & \text{if } g = g_1 \wedge g_2\\
\mathbf{timer}(v-1) & \text{if } g = \mathbf{timer}(v)\\
g & \text{otherwise}
\end{cases}
$$

Fig. 5. Functions controlling the advancement of time.

processes of all objects, decrementing all *deadline* values as well as **timer** statements and guards at the head of the statement list in processes.

Annotations and Resources. Figure 6 presents the transition rules related to the creation of deployment components and resource annotations. We consider three kinds of annotations in this paper: *deadline*, *cost*, and *free*. The deadline annotation is associated with method calls, and is handled by the rule Async-Call (explained below). The activation of a method and the creation of a class may affect the resources at both the sender and receiver side. Receiver side annotations are associated with the declaration of methods and classes, while sender side annotations may be associated with any statement in the model. A statement with an empty list of annotations is handled as a statement without annotations by rule No-Annotations. Cost annotations are controlled by rule Cost-Annotation

$$\textsc{(New-Component)}\quad \frac{\mathit{fresh}(dc) \quad v = \llbracket e \rrbracket^t_{aol}}{\begin{array}{c} o(a, \{l|x = \texttt{component}(e); s\}, q)\ cl(t) \\ \to o(a, \{l|x = dc; s\}, q)\ dc(v, 0, v)\ cl(t) \end{array}}$$

$$\textsc{(New-Object1)}\quad \frac{v = a(\text{thiscomp})}{\begin{array}{c} o(a, \{l|x = \texttt{new}\ C(\overline{e}); s\}, q) \to \\ o(a, \{l|x = \texttt{new}\ C(\overline{e})\ \texttt{in}\ v; s\}, q) \end{array}}$$

$$\textsc{(Async-Call)}\quad \frac{\begin{array}{c} \mathit{fresh}(f) \quad an = \texttt{deadline:}e' \\ o' = \llbracket e \rrbracket^t_{aol} \quad \overline{v} = \llbracket \overline{e} \rrbracket^t_{aol} \quad d = \llbracket e' \rrbracket^t_{aol} \end{array}}{\begin{array}{c} o(a, \{l|[an]\, x := e!m(\overline{e}); s\}, q)\ cl(t) \to \\ o(a, \{l|x := f); s\}, q)\ cl(t)\ m(o', \overline{v}, f, d)\ f \end{array}}$$

$$\textsc{(New-Object2)}\quad \frac{\begin{array}{c} \text{fresh}(o') \quad a' = \text{atts}(C, \llbracket \overline{e} \rrbracket^t_{aol}, o', \llbracket e \rrbracket^t_{aol}) \\ \{l'|s'\} = \text{init}(C) \quad \text{annotations}(\text{init}, C) = an \end{array}}{\begin{array}{c} o(a, \{l|x = \texttt{new}\ C(\overline{e})\ \texttt{in}\ e; s\}, q)\ cl(t) \to \\ o(a, \{l|x = o'; s\}, q)\ o'(p', a', \{l'|[an]\, s'\}, \emptyset)\ cl(t) \end{array}}$$

$$\textsc{(Activation)}\quad \frac{\begin{array}{c} \text{class}(o) = C \\ \text{annotations}(m, C) = an \\ \{l|s\} = \text{bind}(m, C, \overline{v}, f, d) \end{array}}{\begin{array}{c} o(a, pr, q)\ m(o, \overline{v}, f, d) \\ \to o(a, pr, \{l|[an]\, s\} \circ q) \end{array}}$$

$$\textsc{(Cost-Annotation)}\quad \frac{\begin{array}{c} c \leq av \quad \llbracket e \rrbracket^t_{aol} = c \quad a(\text{thiscomp}) = dc \\ \{o(a, \{l|[an]\, s\}, q)\ dc(av - c, fr, tot)\ cl(t)\ cn\} \\ \to \{o(a', pr', q')\ dc(av', fr', tot)\ cl(t)\ cn'\} \end{array}}{\begin{array}{c} \{o(a, \{l|[\texttt{cost:}e, an]\, s\}, q)\ dc(av, fr, tot)\ cl(t)\ cn\} \\ \to \{o(a', pr', q')\ dc(av', fr', tot)\ cl(t)\ cn'\} \end{array}}$$

$$\textsc{(No-Annotations)}\quad \frac{\begin{array}{c} \{o(a, \{l|s\}, q)\ cn\} \\ \to \{o(a', pr', q')\ cn'\} \end{array}}{\begin{array}{c} \{o(a, \{l|[\varepsilon]\, s\}, q)\ cn\} \\ \to \{o(a', pr', q')\ cn'\} \end{array}}$$

$$\textsc{(Free-Annotation)}\quad \frac{\begin{array}{cc} \llbracket e \rrbracket^t_{aol} = f & \{o(a, \{l|[an]\, s\}, q)\ dc(av, fr, tot)\ cl(t)\ cn\} \\ a(\text{thiscomp}) = dc & \to \{o(a', pr', q')\ dc(av', fr', tot)\ cl(t)\ cn'\} \end{array}}{\begin{array}{c} \{o(a, \{l|[\texttt{free:}e, an]\, s\}, q)\ dc(av, fr, tot)\ cl(t)\ cn\} \\ \to \{o(a', pr', q')\ dc(av', fr' + f, tot)\ cl(t)\ cn'\} \end{array}}$$

Fig. 6. Annotated transitions in the semantics of ABS.

and remove the specified amount e of resources from the available resources of the deployment component in which the execution occurs. Observe that execution can only occur if there are enough resources available, given by the constraint $c \leq av$. Free annotations are similarly controlled by rule Free-Annotation and add the specified amount e of resources to the resources which may be freed in the deployment component at the next time advance.

Auxiliary functions. Let $class(o)$ denote the class of an object with identifier o and let $bind(m, C, \overline{v}, f, d)$ return a process resulting from the activation of m in class C, with actual parameters $\overline{v}$, associated future f, and deadline d. If binding succeeds, the method's formal parameters are bound to $\overline{v}$, the reserved variables *destiny* and *deadline* are bound to f and d, respectively. Let $annotations(m, C)$ return the receiver side annotations associated with the definition of m in C. If m is *init*, the annotations of the class definition are returned. If there are no associated annotations, it returns ε. Let $atts(C, \overline{v}, o, dc)$ return the initial state of an instance of class C, in which the formal class parameters are bound to $\overline{v}$ and the reserved variables *this* and *thiscomp* are bound to o and dc. Let $init(C)$ return an activation of the *init* method of C, if defined, and call the *run* method, if defined. Otherwise it returns *idle*.

Method Calls. Rule Async-Call sends an invocation message of method m to $\llbracket e \rrbracket^t_{aol}$ with actual parameters $\llbracket \overline{e} \rrbracket^t_{aol}$, the unique identity f of a new future (since $\mathit{fresh}(f)$), and a deadline obtained from the (possibly default) annotation. The identifier of the new future is placed in the configuration, and is bound to a return value in Return. In rule Activation the function $\mathit{bind}(m, C, \bar{v}, f, d)$ binds a method call to object o in the class of o and $\mathit{annotations}(m, C)$ returns the annotations associated with the method declaration. This results in a new process $\{l|[an]s\}$ which is placed in the queue, where $l(\text{destiny}) = f$, $l(\text{deadline}) = d$, and where the formal parameters of m are bound to $\overline{v}$. Rule Return places the evaluated return expression in the future associated with the *destiny* variable of the process. Rule Read-Fut dereferences the future $f(v)$. Note that if the future lacks a return value, the reduction in this object is *blocked*.

Creation of Objects and Deployment Components. Rule New-Object1 reduces to an object creation in the current deployment component (**thiscomp**). Rule New-Object2 creates a new object with a unique identifier o' in a deployment component dc. The object's fields are given default values by $\mathrm{atts}(C, \overline{v}, o', dc)$, extended with the actual values $\overline{v}$ for the class parameters, o' for *this*, and dc for *thiscomp*. In order to instantiate the remaining attributes, the process $\mathrm{init}(C)$ is scheduled, with the appropriate class annotations. Rule New-Component creates a new deployment component with a specified number of resources e, which are initially available.

4 Example

The semantics of Section 3 have been implemented in rewriting logic to be executable in the Maude tool [8]. This section briefly presents a working example and simulation results. Consider the model of a service for processing a job consisting of a number of work packets, e.g., batch-converting a number of photos or video frames (Figure 7). A client calls the `request` method of a `Server` object; the number of frames, and a deadline for processing are given as parameters to the `request` method, as are (specifications of) the time and memory processing cost for each work unit. These last two parameters can be seen as abstractions of the size of the image. The class `ClientImp` can be used to simulate specific workloads, dispatching jobs of a given size with a certain frequency to the server.

4.1 Simulating Resource-Restricted Behaviors

A very simple scenario shows the principle of simulating resource restricted execution in Timed ABS. Starting a client as follows:

```
new ClientImp (s, 1, 1, 3, 3, 3, Duration(8));
```

(1 job with 3 frames, $t = 3$ and cost 3 for each frame, deadline 8), with the server `s` running in a deployment component with 3 memory units results in one missed deadline. Since the available memory allows only one process to run at a time, the three frames are processed in sequence and the last one cannot meet

```
interface Server {
   Bool request(Int nFrames, Int bc, Int wc, Duration dl);
   Bool processframe(Int bc, Int wc);
}
class ServerImp implements Server {
   Bool request(Int nFrames, Int bc, Int wc, Duration dl) {
      List<Fut<Bool>> results = Nil;
      Bool result = True;
      while (nFrames > 0) {
         [Deadline: dl] Fut<Bool> r = this!processframe(bc, wc);
         results = Cons(r, results);
         nFrames = nFrames - 1;
         }
      while (~(results == Nil)){
         await (head(results))?;
         Bool fr = head(results).get; result = result && fr;
         results = tail(results);
      }
      return result;
   }
   Bool processframe(Int bc, Int wc) {
      [MCost: wc] await duration(bc) ;
      [MFree: wc] return deadline() > 0;
   }
}

interface Client { }
class ClientImp (Server s, Int frequency, Int nJobs, Int nFrames,
             Int bc, Int wc, Duration dl)
implements Client {
   Unit run() {
     if (nJobs > 0) {
      Fut<Bool> res = s!request(nFrames, bc, wc, dl);
      await duration(frequency);
      nJobs = nJobs - 1;
      this!run(); await res?;
      }
   }
}
```

Fig. 7. Photo and video processing example.

the deadline. Increasing available memory to 6 allows the job to run within the specified deadline.

Figure 8 shows the results of a more involved scenario: two clients running a job with 10 frames every 6 time units, for a total of 10 jobs, which would be done in time $t = 60$ for an unconstrained server. The amount of missed deadlines varies with the amount of memory available for processing, up to 75 memory units which are enough to process the given workload scenario without deadline

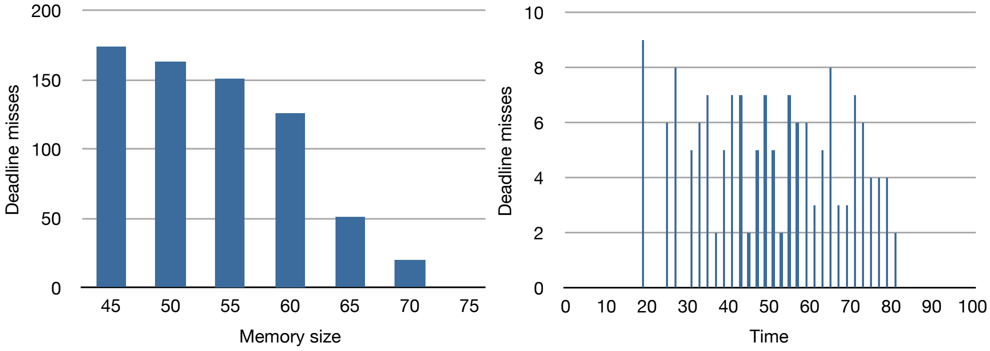


Fig. 8. Deadline misses as a function of memory size (left); deadline misses over time for memory size=55 (right)

misses. The behavior of the server over time can also be visualized; Figure 8 also shows deadline misses over time on memory size 55 for the same scenario.

5 Related Work

The concurrency model provided by concurrent objects and Actor-based computation, in which software units with encapsulated processors communicate asynchronously, is increasingly getting attention due to its intuitive and compositional nature [3, 14, 29]. This compositionality allows concurrent objects to be naturally distributed on different locations, because only the local state of a concurrent object is needed to execute its methods. In previous work [2, 19, 20], the authors have introduced *deployment components* as a modeling concept which captures restricted resources shared between a group of concurrent objects, and shown how components with parametric resources may be used to capture a model's behavior for different assumptions about the available resources.

Techniques for prediction or analysis of non-functional properties are based on either *measurement* or *modeling*. Measurement-based approaches apply to existing implementations, using dedicated profiling or tracing tools like JMeter or LoadRunner. Model-based approaches allow abstraction from specific system intricacies, but depend on parameters provided by domain experts [11]. A survey of model-based performance analysis techniques is given in [4]. Formal systems using process algebra, Petri Nets, game theory, and timed automata have been used in the embedded software domain (e.g., [5,12]), but also to the schedulability of processes in concurrent objects [16]. The latter work complements ours as it does not consider restrictions on shared deployment resources, but associates deadlines with method calls with abstract duration statements.

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 [25] 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 [4]. Closer to our work is M. Verhoef's extension of VDM++ for embedded real-time systems [28], in which architectures are explicitly modeled using CPUs and buses. The approach uses fixed resources targeting the embedded domain, namely processor cycles bound to the CPUs, while we consider more general resources for arbitrary software.

6 Discussion and Future Work

We have described a generic way of adding resource constraints to a executable behavioral model. This expands on previous work, where the cost of execution was bound to a specific resource and directly fixed in the language semantics. In contrast, this paper generalized the approach by enabling the specification of resource costs as part of the software development process, supported by explicit user-defined cost statements expressed in terms of the local state and the input parameters to methods. This way, the cost of execution in the model can be adapted by the modeler to a specific cost scenario. Our extension to ABS allows us to abstractly model the effect of deploying concurrent objects on deployment components with different amounts of assigned resources at an early stage in the software development process, before modeling the detailed control flow of the targeted system.

This approach gives the modeler all freedoms, at the cost of manual resource tracking. For specific kinds of resources, more specialized notations can make expressing resource constraints easier – for example, for CPU (processing) resources, cost and free are always the same amount, while for power consumption resources are never freed, except when modeling an operator recharging a battery. Alternatively, our approach can be supported by a cost analysis tool such as COSTA [1]). In collaboration with Albert et al., this was implemented for memory analysis of ABS models [2]. However, the generalization of that work for general, user-defined resources and its integration into the software development process remains future work.

Another extension of this work is to strengthen the available analysis methods via a number of language extensions: User-defined schedulers for ABS, probabilistic scheduling and load balancing of resources and objects will provide more precise simulation results given knowledge of the deployment platform and show the range of possible behaviors wrt performance of the system. Prototype implementations of these features exist, and our current work is dedicated to integrating them in the base language and tools.

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-form upper bounds in static cost analysis. *Journal of Automated Reasoning*, 46:161–203, 2011.

2. E. Albert, S. Genaim, M. Gómez-Zamalloa, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Simulating concurrent behaviors with worst-case cost bounds. In *FM 2011*, volume 6664 of *LNCS*, pages 353–368. Springer, June 2011.
3. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
4. S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.
5. A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Resource interfaces. In *Proc. EMSOFT'03*, volume 2855 of *LNCS*, pages 117–133. Springer, 2003.
6. P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Proc. FMCO'05*, volume 4111 of *LNCS*, pages 342–363. Springer, 2005.
7. D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, and P. Y. H. Wong. Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In *Proc. SFM 2011*, volume 6659 of *LNCS*, pages 417–457. Springer, 2011.
8. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.
9. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *Proc. ESOP'07*, volume 4421 of *LNCS*, pages 316–330. Springer, Mar. 2007.
10. F. S. de Boer, M. M. Jaghoori, and E. B. Johnsen. Dating concurrent objects: Real-time modeling and schedulability analysis. In *Proc. CONCUR'10*, volume 6269 of *LNCS*, pages 1–18. Springer, Sept. 2010.
11. I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by runtime parameter adaptation. In *Proc. ICSE'09*, pages 111–121. IEEE, 2009.
12. E. Fersman, P. Krcál, P. Pettersson, and W. Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 2007.
13. N. E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, pages 323–334, September 1992.
14. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.
15. I. Hayes and C. Jones. Specifications are not (Necessarily) Executable. *Software Engineering Journal*, pages 330–338, November 1989.
16. M. M. Jaghoori, F. S. de Boer, T. Chothia, and M. Sirjani. Schedulability of asynchronous real-time concurrent objects. *Journal of Logic and Algebraic Programming*, 78(5):402–416, 2009.
17. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Proc. FMCO'10*, LNCS. Springer, 2011. To appear.
18. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
19. E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Dynamic resource reallocation between deployment components. In *Proc. ICFEM'10*, volume 6447 of *LNCS*, pages 646–661. Springer, Nov. 2010.

20. E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Validating timed models of deployment components with parametric concurrency. In *Proc. FoVeOOS'10*, volume 6528 of *LNCS*, pages 46–60. Springer, 2011.
21. J. Kramer. Is Abstraction the Key to Computing? *Communications of the ACM*, 50(4):37–42, 2007.
22. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
23. P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285(2):359–405, 2002.
24. P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1–2):161–196, 2007.
25. D. B. Petriu and C. M. Woodside. An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Software and System Modeling*, 6(2):163–184, 2007.
26. B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
27. G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.
28. M. Verhoef, P. G. Larsen, and J. Hooman. Modeling and validating distributed embedded real-time systems with VDM++. In *Proc. FM'06*, volume 4085 of *LNCS*, pages 147–162. Springer, 2006.
29. A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proc. OOPSLA'05*, pages 439–453, New York, NY, USA, 2005. ACM Press.
30. J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):1–36, October 2009.

Automated Detection of Non-Termination and `NullPointerExceptions` for **Java Bytecode**[⋆]

Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl

LuFG Informatik 2, RWTH Aachen University, Germany

Abstract. Recently, we developed an approach for automated termination proofs of Java Bytecode (JBC), which is based on constructing and analyzing *termination graphs*. These graphs represent all possible program executions in a finite way. In this paper, we show that this approach can also be used to detect *non-termination* or `NullPointerExceptions`. Our approach automatically generates *witnesses*, i.e., calling the program with these witness arguments indeed leads to non-termination resp. to a `NullPointerException`. Thus, we never obtain "false positives". We implemented our results in the termination prover AProVE and provide experimental evidence for the power of our approach.

1 Introduction

To use program verification in the software development process, one is not only interested in proving the validity of desired program properties, but also in providing a witness (i.e., a counterexample) if the property is violated.

Our approach is based on our earlier work to prove termination of JBC [4, 6, 17]. Here, a JBC program is first automatically transformed to a *termination graph* by symbolic evaluation. Afterwards, a term rewrite system is generated from the termination graph and existing techniques from term rewriting are used to prove termination of the rewrite system. As shown in the annual *International Termination Competition*,[1] our corresponding tool AProVE is currently among the most powerful ones for automated termination proofs of Java programs.

Termination graphs finitely represent all runs through a program for a certain set of input values. Similar graphs were used for many kinds of program analysis (e.g., to improve the efficiency of software verification [7], or to ensure termination of program optimization [22]). In this paper, we show that termination graphs can also be used to detect non-termination and `NullPointerExceptions`.

In Sect. 2, we recapitulate termination graphs. In contrast to [4, 6, 17], we also handle programs with *arrays* and we present an algorithm to *merge* abstract states in a termination graph which is needed in order to keep termination graphs finite. In Sect. 3 we show how to automatically generate *witness states* (i.e., suitable inputs to the program) which result in errors like `NullPointerExceptions`. Sect. 4 presents our approach to detect non-termination. Here, we use an SMT

[⋆] Supported by the DFG grant GI 274/5-3, the G.I.F. grant 966-116.6, and the DFG Research Training Group 1298 (*AlgoSyn*).

[1] See `http://www.termination-portal.org/wiki/Termination_Competition`

solver to find different forms of non-terminating loops and the technique of Sect. 3 is applied to generate appropriate witness states.

Concerning the detection of NullPointerExceptions, most existing techniques try to prove *absence* of such exceptions (e.g., [15, 23]), whereas our approach tries to prove *existence* of NullPointerExceptions and provides counterexamples which indeed lead to such exceptions. So in contrast to bug finding techniques like [2, 9], our approach does not yield “false positives”.

Methods to detect *non-termination* automatically have for example been studied for term rewriting (e.g., [11, 19]) and logic programming (e.g., [18]). We are only aware of two existing tools for automated non-termination analysis of Java: The tool Julia transforms JBC programs into constraint logic programs, which are then analyzed for non-termination [20]. The tool Invel [24] investigates non-termination of Java programs based on a combination of theorem proving and invariant generation using the KeY [3] system. In contrast to Julia and to our approach, Invel only has limited support for programs operating on the heap. Moreover, neither Julia nor Invel return witnesses for non-termination. In Sect. 5 we compare the implementation of our approach in the tool AProVE with Julia and Invel and show that our approach indeed leads to the most powerful automated non-termination analyzer for Java so far.

Moreover, [14] presents a method for non-termination proofs of C programs. In contrast to our approach, [14] can deal with non-terminating recursion and integer overflows. On the other hand, [14] cannot detect *non-periodic* non-termination (where there is no fixed sequence of program positions that is repeated infinitely many times), whereas this is no problem for our approach, cf. Sect. 4.2.

There also exist tools for testing C programs in a randomized way, which can detect candidates for potential non-termination bugs (e.g., [13, 21]). However, they do not provide a proof for non-termination and may return “false positives”.

2 Termination Graphs

```
public class Loop {
 public static void main(String[] a){
  int i = 0;
  int j = a.length;
  while (i < j) {
   i += a[i].length(); }}}
```

Fig. 1. Java Program

```
    main(String[] a):
00: iconst_0     #load 0 to stack
01: istore_1     #store to i
02: aload_0      #load a to stack
03: arraylength  #get array length
04: istore_2     #store to j
05: iload_1      #load i to stack
06: iload_2      #load j to stack
07: if_icmpge 22 #jump to end if i >= j
10: iload_1      #load i to stack
11: aload_0      #load a to stack
12: iload_1      #load i to stack
13: aaload       #load a[i]
14: invokevirtual length #call length()
17: iadd         #add length and i
18: istore_1     #store to i
19: goto 05
22: return
   length():
00: aload_0        #load this to stack
01: getfield count #load count field
04: ireturn        #return it
```

Fig. 2. JBC Program

We illustrate our approach by the main method of the Java program in Fig. 1. The main method is the entry point when starting a program. Its only argument is an array of String objects corresponding to the arguments specified on the command line. To avoid dealing with all syntactic constructs of Java, we analyze JBC instead. JBC [16] is an assembly-like object-oriented language designed as intermediate format for the execution of Java. The corresponding JBC for our example, obtained automatically by the standard `javac` compiler, is shown in Fig. 2 and will be explained in Sect. 2.2.

The method `main` increments `i` by the length of the `i`-th input string until `i` exceeds the number `j` of input arguments. It combines two typical problems:

(a) The accesses to `a.length` and `a[i].length()` are not guarded by appropriate checks to ensure memory safety. Thus, if `a` or `a[i]` are `null`, the method ends with a `NullPointerException`. While this cannot happen when the method is used as an entry point for the program, another method could for instance contain `String[] b = {null}; Loop.main(b)`.

(b) The method may not terminate, as the input arguments could contain the empty string. If `a[i] = ""`, then the counter `i` is not increased, leading to *looping* non-termination, as the same program state is visited again and again. For instance, the call `java Loop ""` does not terminate.

We show how to automatically detect such problems and to synthesize appropriate witnesses in Sect. 3 and 4. Our approach is based on *termination graphs* that over-approximate all program executions. After introducing our notion of states in Sect. 2.1, we describe the construction of termination graphs in Sect. 2.2. Sect. 2.3 shows how to create "merged" states representing two given states.

2.1 Abstract States

Our approach is related to *abstract interpretation* [8], since the states in termination graphs are *abstract*, i.e., they represent a (possibly infinite) set of concrete system configurations of the program. We define the set of all states as $\textsc{States} = (\textsc{PPos} \times \textsc{LocVar} \times \textsc{OpStack})^* \times (\{\bot\} \cup \textsc{Refs}) \times \textsc{Heap} \times \textsc{Annotations}$.

Consider the program from Fig. 1. The initial state A in Fig. 3 represents all system configurations entering the `main` method with arbitrary tree-shaped (and thus, acyclic) non-`null` arguments. A state consists of four parts: the call stack, exception information, the heap, and annotations for possible sharing effects.

$\texttt{00} \,|\, \texttt{a}\!:\!a_1 \,|\, \varepsilon$
$a_1\!:\!\texttt{String[]}\ i_1 \quad i_1\!:\![\geq 0]$

Fig. 3. State A

The call stack consists of stack frames, where *several* frames may occur due to method calls. For readability, we exclude recursive programs, but our results easily extend to the approach of [6] for recursion. We also disregard multi-threading, reflection, static fields, static initialization of classes, and floats.

Each stack frame has three components. We write the frames of the call stack below each other and separate their components by "|". The first component of a frame is the program position, indicated by the number of the next instruction (`00` in Fig. 3). The second component represents the local variables by a list of references to the heap, i.e., $\textsc{LocVar} = \textsc{Refs}^*$. To avoid a special treatment of primitive values, we also represent them by references. In examples, we write the names of variables instead of their indices. Thus, "$\texttt{a}\!:\!a_1$" means that the value of the 0-th local variable `a` is the reference a_1 (i.e., a_1 is the address of an array object). Of course, different local variables can point to the same address. The third component is the operand stack that JBC instructions work on, where $\textsc{OpStack} = \textsc{Refs}^*$. The empty stack is "$\varepsilon$" and "$i_6, i_4$" is a stack with i_6 on top.

Information about thrown exceptions is represented in the second part of our states. If no exception is currently thrown, this part is $\bot$ (which we do not display in example states). Otherwise it is a reference to the exception object.

Below the call stack, information about the heap is given by a partial function from $\textsc{Heap} = \textsc{Refs} \to (\textsc{Integers} \cup \textsc{Unknown} \cup \textsc{Instances} \cup \textsc{Arrays} \cup \{\texttt{null}\})$ and by a set of annotations which specify possible sharing effects.

Our representation of integers abstracts from the different bounded types of integers in Java and considers arbitrary integer numbers instead (i.e., we do not handle overflows). To represent unknown integer values, we use possibly unbounded intervals, i.e., $\textsc{Integers} = \{\{x \in \mathbb{Z} \mid a \leq x \leq b\} \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{\infty\}, a \leq b\}$. We abbreviate $(-\infty, \infty)$ by $\mathbb{Z}$ and intervals like $[0, \infty)$ by $[\geq 0]$. So "i_1: $[\geq 0]$" means that any non-negative integer can be at the address i_1.

$\textsc{Classnames}$ contains the names of all classes and interfaces in the program. $\textsc{Types} = \textsc{Classnames} \cup \{t[\,] \mid t \in \textsc{Types}\}$ contains $\textsc{Classnames}$ and all resulting array types. So a type $t[\,]$ can be generated from any type t to describe arrays with entries of type t.[2] We call t' a *subtype* of t iff $t' = t$; or t' `extends`[3] or `implements` a subtype of t; or $t' = \hat{t}'[\,]$, $t = \hat{t}[\,]$, and $\hat{t}'$ is a subtype of $\hat{t}$.

The values in $\textsc{Unknown} = \textsc{Types} \times \{?\}$ represent tree-shaped (and thus acyclic) objects and arrays where we have no information except the type. For example, for a class `List` with the field `next` of type `List`, "o_1 : `List`(?)" means that the object at address o_1 is `null` or of a subtype of `List`.

$\textsc{Instances}$ represent objects of some class. They are described by the values of their fields, i.e., $\textsc{Instances} = \textsc{Classnames} \times (\textsc{FieldIDs} \to \textsc{Refs})$. For cases where field names are overloaded, the $\textsc{FieldIDs}$ also contain the respective class name to avoid ambiguities, but we usually do not display it in our examples. So "o_1 : `List`(`next` $= o_2$)" means that at the address o_1, there is a `List` object and the value of its field `next` is o_2. For all $(cl, f) \in \textsc{Instances}$, the function f is defined for all fields of the class cl and all of its superclasses.

In contrast to our earlier papers [4, 6, 17], in this paper we also show how to handle *arrays*. An array can be represented by an element from $\textsc{Types} \times \textsc{Refs}$ denoting the array's type and length (specified by a reference to an integer value). For instance, "a_1: `String`[ ] i_1" means that at the address a_1, there is a `String` array of length i_1. Alternatively, the array representation can also contain an additional list of references for the array entries. So "a_2: `String`[ ] i_1 $\{o_1, o_2\}$" denotes that at the address a_2, we have a `String` array of length i_1, and its entries are o_1 and o_2. Thus, $\textsc{Arrays} = (\textsc{Types} \times \textsc{Refs}) \cup (\textsc{Types} \times \textsc{Refs} \times \textsc{Refs}^*)$.

In our representation, no sharing can occur unless explicitly stated. So an abstract state containing the references o_1, o_2 and not mentioning that they could be sharing, only represents concrete states where o_1 and the references reachable from o_1 are disjoint[4] from o_2 and the references reachable from o_2. Moreover, then the objects at o_1 and o_2 must be tree-shaped (and thus acyclic).

[2] We do not consider arrays of primitives in this paper, but our approach can easily be extended to handle them, as we did in our implementation.

[3] For example, any type (implicitly) `extends` the type `java.lang.Object`.

[4] Disjointness is not required for references pointing to $\textsc{Integers}$ or to `null`.

Certain sharing effects are represented directly (e.g., "o_1:`List`(`next`$=o_1$)" is a cyclic singleton list). Other sharing effects are represented by three kinds of *annotations*, which are only built for references o where $h(o) \notin \text{Integers} \cup \{\texttt{null}\}$ for the heap h. The first kind of annotation is called *equality annotation* and has the form "$o_1 =^? o_2$". Its meaning is that the addresses o_1 and o_2 could be equal. We only use such annotations if the value of at least one of o_1 and o_2 is Unknown. *Joinability annotations* are the second kind of annotation. They express that two objects "may join" ($o_1 \searrow\!\!\swarrow o_2$). We say that a non-integer and non-`null` reference o' is a direct successor of o in a state s (denoted $o \to_s o'$) iff the object at address o has a field whose value is o' or if the array at address o has o' as one of its entries. The meaning of "$o_1 \searrow\!\!\swarrow o_2$" is that there could be an o with $o_1 \to_s^* o \leftarrow_s^+ o_2$ or $o_1 \to_s^+ o \leftarrow_s^* o_2$, i.e., o is a common successor of the two references. However, $o_1 \searrow\!\!\swarrow o_2$ does not imply $o_1 =^? o_2$. Finally, as the third type of annotations, we use *cyclicity annotations* "$o!$" to denote that the object at address o is not necessarily tree-shaped (so in particular, it could be cyclic).

2.2 Constructing Termination Graphs

Starting from the initial state A, the termination graph in Fig. 4 is constructed by symbolic evaluation. In the first step, we have to evaluate `iconst_0`, i.e., we load the integer 0 on top of the operand stack. The second instruction `istore_1` stores the value 0 on top of the operand stack in the first local variable `i`.[5]

After that, the value of the 0-th local variable `a` (the array in the input argument) is loaded on the operand stack and the instruction `arraylength` retrieves its (unknown) length i_1. That value is then stored in the second local variable `j` using the instruction `istore_2`. This results in the state B in Fig. 4. We connect A and B by a dotted arrow, indicating several evaluation steps (i.e., we omitted the states between A and B for space reasons in Fig. 4).

From B on, we load the values of `i` and `j` on the operand stack and reach C.[6] The instruction `if_icmpge` branches depending on the relation of the two elements on top of the stack. However, based on the knowledge in C, we cannot determine whether `i >= j` holds. Thus, we perform a case analysis (called *integer refinement* [4, Def. 1]), obtaining two new states D and E. We label the *refinement edges* from C to D and E (represented by dashed arrows) by the reference i_1 that was refined. In D, we assume that `i >= j` holds. Hence, i_1 (corresponding to `j`) is ≤ 0 and from $i_1 : [\geq 0]$ in state C we conclude that i_1 is 0. We thus reach instruction 22 (`return`), where the program ends (denoted by $\Box$).

In E, we consider the other case and replace i_1 by i_2, which only represents positive integers. We mark what relation holds in this case by labeling the evaluation edge from E to its successor with $0 < i_2$. In general, we always use a fresh reference name like i_2 when generating new values by a case analysis, to

[5] If we have a reference whose value is from a singleton interval like $[0, 0]$ or `null`, we replace all its occurrences in states by 0 resp. by `null`. So in state B, we simply write "`i`:0". Such abbreviations will also be used in the labels of edges.

[6] The box around C and the following states is dashed to indicate that these states will be removed from the termination graph later on.

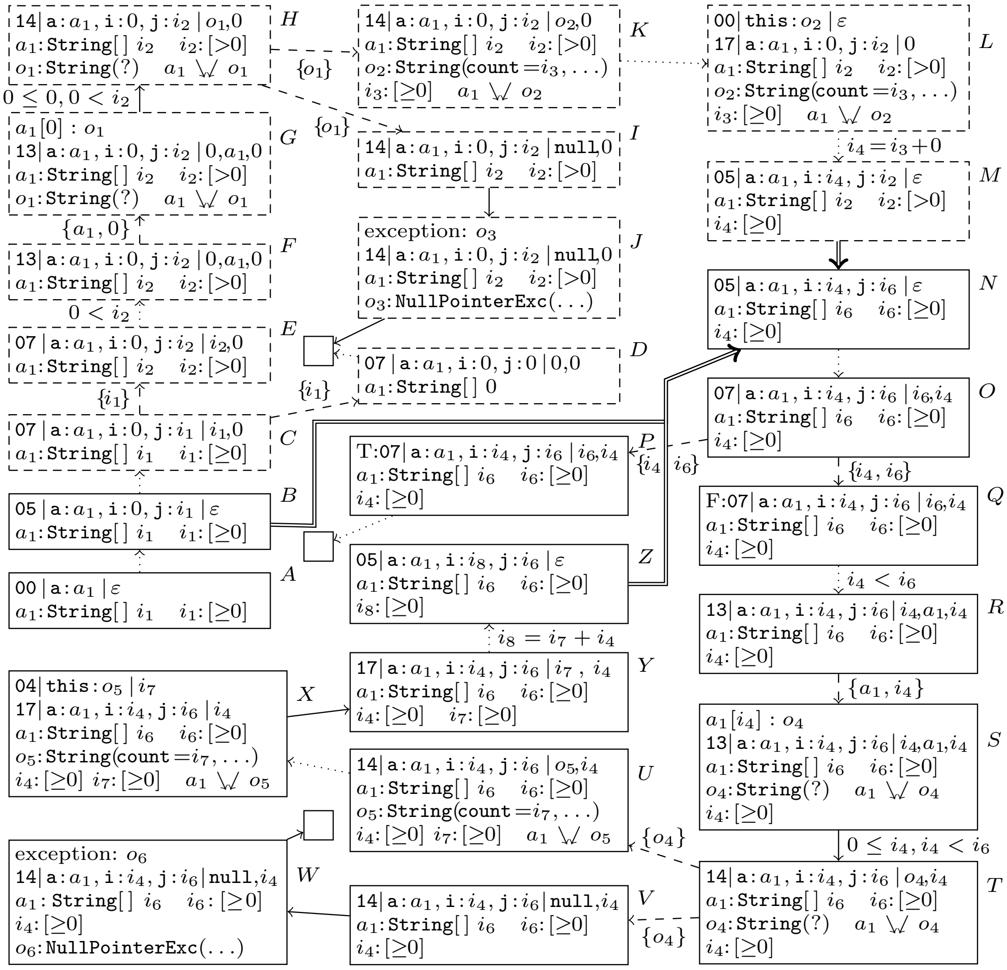

H: 14| a : a_1, i : 0, j : i_2 | o_1,0 / a_1: String[] i_2 i_2: [>0] / o_1: String(?) $a_1 \searrow\!\swarrow o_1$
$0 \le 0, 0 < i_2$
G: $a_1[0]$: o_1 / 13| a : a_1, i : 0, j : i_2 | 0,a_1,0 / a_1: String[] i_2 i_2: [>0] / o_1: String(?) $a_1 \searrow\!\swarrow o_1$
$\{a_1, 0\}$
F: 13| a : a_1, i : 0, j : i_2 | 0,a_1,0 / a_1: String[] i_2 i_2: [>0]
$0 < i_2$
E: 07 | a : a_1, i : 0, j : i_2 | i_2,0 / a_1: String[] i_2 i_2: [>0]
$\{i_1\}$
C: 07 | a : a_1, i : 0, j : i_1 | i_1,0 / a_1: String[] i_1 i_1: [≥0]
B: 05 | a : a_1, i : 0, j : i_1 | ε / a_1: String[] i_1 i_1: [≥0]
A: 00 | a : a_1 | ε / a_1: String[] i_1 i_1: [≥0]
$\{o_1\}$
K: 14| a : a_1, i : 0, j : i_2 | o_2,0 / a_1: String[] i_2 i_2: [>0] / o_2: String(count = i_3, . . .) / i_3: [≥0] $a_1 \searrow\!\swarrow o_2$
$\{o_1\}$
I: 14| a : a_1, i : 0, j : i_2 | null,0 / a_1: String[] i_2 i_2: [>0]
J: exception: o_3 / 14| a : a_1, i : 0, j : i_2 | null,0 / a_1: String[] i_2 i_2: [>0] / o_3: NullPointerExc(. . .)
D: 07 | a : a_1, i : 0, j : 0 | 0,0 / a_1: String[] 0
$\{i_1\}$
P: T:07| a : a_1, i : i_4, j : i_6 | i_6,i_4 / a_1: String[] i_6 i_6: [≥0] / i_4: [≥0]
$\{i_4$ $i_6\}$
Z: 05| a : a_1, i : i_8, j : i_6 | ε / a_1: String[] i_6 i_6: [≥0] / i_8: [≥0]
$i_8 = i_7 + i_4$
Y: 17| a : a_1, i : i_4, j : i_6 | i_7 , i_4 / a_1: String[] i_6 i_6: [≥0] / i_4: [≥0] i_7: [≥0]
X: 04| this : o_5 | i_7 / 17| a : a_1, i : i_4, j : i_6 | i_4 / a_1: String[] i_6 i_6: [≥0] / o_5: String(count = i_7, . . .) / i_4: [≥0] i_7: [≥0] $a_1 \searrow\!\swarrow o_5$
U: 14| a : a_1, i : i_4, j : i_6 | o_5,i_4 / a_1: String[] i_6 i_6: [≥0] / o_5: String(count = i_7, . . .) / i_4: [≥0] i_7: [≥0] $a_1 \searrow\!\swarrow o_5$
$\{o_4\}$
W: exception: o_6 / 14| a : a_1, i : i_4, j : i_6 | null,i_4 / a_1: String[] i_6 i_6: [≥0] / i_4: [≥0] / o_6: NullPointerExc(. . .)
V: 14| a : a_1, i : i_4, j : i_6 | null,i_4 / a_1: String[] i_6 i_6: [≥0] / i_4: [≥0]
$\{o_4\}$
L: 00| this : o_2 | ε / 17| a : a_1, i : 0, j : i_2 | 0 / a_1: String[] i_2 i_2: [>0] / o_2: String(count = i_3, . . .) / i_3: [≥0] $a_1 \searrow\!\swarrow o_2$
$i_4 = i_3 + 0$
M: 05| a : a_1, i : i_4, j : i_2 | ε / a_1: String[] i_2 i_2: [>0] / i_4: [≥0]
N: 05| a : a_1, i : i_4, j : i_6 | ε / a_1: String[] i_6 i_6: [≥0] / i_4: [≥0]
O: 07| a : a_1, i : i_4, j : i_6 | i_6,i_4 / a_1: String[] i_6 i_6: [≥0] / i_4: [≥0]
$\{i_4, i_6\}$
Q: F:07| a : a_1, i : i_4, j : i_6 | i_6,i_4 / a_1: String[] i_6 i_6: [≥0] / i_4: [≥0]
$i_4 < i_6$
R: 13| a : a_1, i : i_4, j : i_6 | i_4,a_1,i_4 / a_1: String[] i_6 i_6: [≥0] / i_4: [≥0]
$\{a_1, i_4\}$
S: $a_1[i_4]$: o_4 / 13| a : a_1, i : i_4, j : i_6 | i_4,a_1,i_4 / a_1: String[] i_6 i_6: [≥0] / o_4: String(?) $a_1 \searrow\!\swarrow o_4$ / i_4: [≥0]
$0 \le i_4, i_4 < i_6$
T: 14| a : a_1, i : i_4, j : i_6 | o_4,i_4 / a_1: String[] i_6 i_6: [≥0] / o_4: String(?) $a_1 \searrow\!\swarrow o_4$ / i_4: [≥0]

Fig. 4. Termination Graph

ensure single static assignments, which will be useful in the analysis later on. We continue with instruction `10` and load the values of `i`, `a`, and `i` on the operand stack, obtaining state F. To evaluate `aaload` (i.e., to load the 0-th element from the array a_1 on the operand stack), we add more information about a_1 at the index 0 and label the refinement edge from F to G accordingly. In G, we created some object o_1 for the 0-th entry of the array a_1 and marked that o_1 is reachable from a_1 by adding the *joinability annotation* $a_1 \searrow\!\swarrow o_1$.[7]

Now evaluation of `aaload` moves o_1 to the operand stack in state H. Whenever an array access succeeds, we label the corresponding edge by the condition that the used index is ≥ 0 and smaller than the length of the array.

In H, we need to `invoke` the method `length()` on the object o_1. However, we do not know whether o_1 is `null` (which would lead to a `NullPointerException`).

[7] If we had already retrieved another value o' from the array a_1, it would also have been annotated with $a_1 \searrow\!\swarrow o'$ and we would consequently add $o_1 \searrow\!\swarrow o'$ and $o_1 =^? o'$ when retrieving o_1, indicating that the two values may share or even be equal.

Hence, we perform an *instance refinement* [4, Def. 5] and label the edges from H to the new states I and K by the reference o_1 that is refined. In I, o_1 has the value `null`. In K, we replace the reference o_1 by o_2, pointing to a concrete `String` object with unknown field values. In Fig. 4, we only display the field `count`, containing the integer reference i_3. In this instance refinement, one uses the special semantics of the pre-defined `String` class to conclude that i_3 can only point to a non-negative integer, as `count` corresponds to the *length* of the string. In I, further evaluation results in a `NullPointerException`. A corresponding exception object o_3 is generated and the exception is represented in J. As no exception handler is defined, evaluation ends and the program terminates.

In K, calling `length()` succeeds. In L, a new stack frame is put on top of the call stack, where the implicit argument `this` is set to o_2. In the called method `length()`, we load o_2 on the operand stack and get the value i_3 of its field `count`. We then return from `length()`, add the returned value i_3 to 0, and store the result in the variable `i`. Afterwards, we jump back to instruction `05`. This is shown in state M and the computation $i_4 = i_3 + 0$ is noted on the evaluation edge.

But now M is at the same program position as B. Continuing our symbolic evaluation would lead to an infinite tree, as we would always have to consider the case where the loop condition `i < j` is still true. Instead, our goal is to obtain a finite termination graph. The solution is to automatically generate a new state N which represents all concrete states that are represented by B or M (i.e., N results from *merging* B and M). Then we can insert *instance edges* from B and M to N (displayed by double arrows) and continue the graph construction with N.

2.3 Instantiating and Merging States

To find differences between states and to merge states, we introduce *state positions.* Such a position describes a “path” through a state, starting with some local variable, operand stack entry, or the exception object and then continuing through fields of objects or entries of arrays. For the latter, we use the set $\text{ArrayIdxs} = \{[j] \mid j \geq 0\}$ to describe the set of all possible array indices.

Definition 1 (State Positions SPos). *Let* $s = (\langle fr_0, \ldots, fr_n \rangle, e, h, a)$ *be a state where each stack frame* fr_i *has the form* (pp_i, lv_i, os_i). *Then* $\text{SPos}(s)$ *is the smallest set containing all the following sequences* π:

- $\pi = \text{LV}_{i,j}$ *where* $0 \leq i \leq n$, $lv_i = \langle o_{i,0}, \ldots, o_{i,m_i} \rangle$, $0 \leq j \leq m_i$. *Then* $s|_\pi$ *is* $o_{i,j}$.
- $\pi = \text{OS}_{i,j}$ *where* $0 \leq i \leq n$, $os_i = \langle o'_{i,0}, \ldots, o'_{i,k_i} \rangle$, $0 \leq j \leq k_i$. *Then* $s|_\pi$ *is* $o'_{i,j}$.
- $\pi = \text{EXC}$ *if* $e \neq \bot$. *Then* $s|_\pi$ *is* e.
- $\pi = \pi' \, v$ *for some* $v \in \text{FieldIDs}$ *and some* $\pi' \in \text{SPos}(s)$ *where* $h(s|_{\pi'}) = (cl, f) \in \text{Instances}$ *and where* $f(v)$ *is defined. Then* $s|_\pi$ *is* $f(v)$.
- $\pi = \pi' \, \text{LEN}$ *for some* $\pi' \in \text{SPos}(s)$ *where* $h(s|_{\pi'}) = (t, i) \in \text{Arrays}$ *or* $h(s|_{\pi'}) = (t, i, d) \in \text{Arrays}$. *Then* $s|_\pi$ *is* i.
- $\pi = \pi' \, [j]$ *for some* $[j] \in \text{ArrayIdxs}$ *and some* $\pi' \in \text{SPos}(s)$, *where* $h(s|_{\pi'}) = (t, i, \langle r_0, \ldots, r_q \rangle) \in \text{Arrays}$ *and* $0 \leq j \leq q$. *Then* $s|_\pi$ *is* r_j.

For any position π, *let* $\overline{\pi}_s$ *denote the maximal prefix of* π *such that* $\overline{\pi}_s \in \text{SPos}(s)$. *We write* $\overline{\pi}$ *if* s *is clear from the context.*

For example, in state K, the position $\pi = \text{OS}_{0,0}$ `count` refers to the reference i_3, i.e., we have $K|_\pi = i_3$ and for the position $\tau = \text{LV}_{0,0}$ LEN, we have $K|_\tau = i_2$. As the field `count` was introduced between H and K by an instance refinement, we have $\pi \notin \text{SPos}(H)$ and $\overline{\pi}_H = \text{OS}_{0,0}$, where $H|_{\overline{\pi}} = o_1$. We can now see that B and M only differ in the positions $\text{LV}_{0,0}$ LEN, $\text{LV}_{0,1}$, and $\text{LV}_{0,2}$.

A state s' is an *instance* of another state s (denoted $s' \sqsubseteq s$) if both are at the same program position and if whenever there is a reference $s'|_\pi$, then either the values represented by $s'|_\pi$ in the heap of s' are a subset of the values represented by $s|_\pi$ in the heap of s or else, $\pi \notin \text{SPos}(s)$. Moreover, shared parts of the heap in s' must also be shared in s. As we only consider verified JBC programs, the fact that s and s' are at the same program position implies that they have the same number of local variables and their operand stacks have the same size. For a formal definition of "instance", we refer to [5, 4].[8]

For example, B is not an instance of M since $h_B(B|_{\text{LV}_{0,2}}) = [0,\infty) \not\subseteq [1,\infty) = h_M(M|_{\text{LV}_{0,2}})$ for the heaps h_B and h_M of B and M. Similarly, $M \not\sqsubseteq B$ because $h_M(M|_{\text{LV}_{0,1}}) = [0,\infty) \not\subseteq \{0\} = h_B(B|_{\text{LV}_{0,1}})$. However, we can automatically synthesize a "merged" (or "widened") state N with $B \sqsubseteq N$ and $M \sqsubseteq N$ by choosing the values for common positions π in B and M to be the union of the values in B and M, i.e., $h_N(N|_\pi) = h_B(B|_\pi) \cup h_M(M|_\pi)$. Thus, we have $h_N(N|_{\text{LV}_{0,2}}) = [0,\infty) \cup [1,\infty) = [0,\infty)$ and $h_N(N|_{\text{LV}_{0,1}}) = \{0\} \cup [0,\infty) = [0,\infty)$.

```
Algorithm mergeStates($s$, $s'$):
$\hat{s}$ = new State($s$)
for $\pi \in \text{SPos}(s) \cap \text{SPos}(s')$:
    $ref$ = mergeRef($s|_\pi$, $s'|_\pi$)
    $\hat{h}(ref)$ = mergeVal($h(s|_\pi)$, $h'(s'|_\pi)$)
    $\hat{s}|_\pi = ref$
for $\pi \neq \pi' \in \text{SPos}(s)$:
    if $(s|_\pi = s|_{\pi'} \vee s|_\pi =^? s|_{\pi'})$
        $\wedge\ h(s|_\pi) \notin \text{Integers} \cup \{\texttt{null}\}$:
        if $\pi, \pi' \in \text{SPos}(\hat{s})$:
            if $\hat{s}|_\pi \neq \hat{s}|_{\pi'}$: Set $\hat{s}|_\pi =^? \hat{s}|_{\pi'}$
        else:
            Set $\hat{s}|_{\overline{\pi}} \searrow\!\!\swarrow \hat{s}|_{\overline{\pi'}}$
    if $s|_\pi \searrow\!\!\swarrow s|_{\pi'}$: Set $\hat{s}|_{\overline{\pi}} \searrow\!\!\swarrow \hat{s}|_{\overline{\pi'}}$
for $\pi \in \text{SPos}(s)$:
    if $s|_\pi!$: Set $\hat{s}|_{\overline{\pi}}!$
    if $\exists \rho \neq \rho' : \pi\rho, \pi\rho' \in \text{SPos}(s) \wedge s|_{\pi\rho} = s|_{\pi\rho'}$
        $\wedge\ \rho, \rho'$ have no common prefix $\neq \varepsilon$
        $\wedge\ h(s|_{\pi\rho}) \notin \text{Integers} \cup \{\texttt{null}\}$:
        if $\pi\rho, \pi\rho' \in \text{SPos}(\hat{s}) \wedge \hat{s}|_{\pi\rho} \neq \hat{s}|_{\pi\rho'}$:
            Set $\hat{s}|_\pi!$
        if $\{\pi\rho, \pi\rho'\} \not\subseteq \text{SPos}(\hat{s})$: Set $\hat{s}|_{\overline{\pi}}!$
... same for $\text{SPos}(s')$ ...
return $\hat{s}$
```

Fig. 5. Merging Algorithm

This merging algorithm is illustrated in Fig. 5. Here, $h, h', \hat{h}$ refer to the heaps of the states $s, s', \hat{s}$, respectively. With `new State(`s`)`, we create a fresh state at the same program position as s. The auxiliary function `mergeRef` is an injective mapping from a pair of references to a fresh reference name. The function `mergeVal` maps two heap values to the most precise value from our abstract domains that represents both input values. For example, `mergeVal`$([0,1],[10,15])$ is $[0,15]$, covering both input values, but also adding $[2,9]$ to the set of represented values. For values of the same type, e.g., `String(count`$=i_1,\ldots$`)` and `String(count`$=i_2,\ldots$`)`, `mergeVal` returns a new object of same type with field values obtained by `mergeRef`, e.g., `String(count`$=i_3,\ldots$`)` where i_3 = `mergeRef(`i_1, i_2`)`. When merging values of differing types or `null`, a

[8] The "instance" definition from [4, Def. 3] can easily be extended to arrays, cf. [5].

value from $\textsc{Unknown}$ with the most precise common supertype is returned.

To handle sharing effects, in a second step, we check if there are "sharing" references at some positions π and π' in s or s' that do not share anymore in the merged state $\hat{s}$. Then we add the corresponding annotations to the maximal prefixes $\hat{s}|_{\overline{\pi}}$ and $\hat{s}|_{\overline{\pi'}}$. Furthermore, we check if there are non-tree shaped objects at some position π in s or s', i.e., if one can reach the same successor using different paths starting in position π. Then we add the annotation $\hat{s}|_{\overline{\pi}}!$.

Theorem 2. *Let $s, s' \in \textsc{States}$ and $\hat{s} =$ `mergeStates(`s`,`s'`)`. Then $s \sqsubseteq \hat{s} \sqsupseteq s'$.*[9]

In our example, we used the algorithm `mergeStates` to create the state N and draw instance edges from B and M to N. Since the computation in N also represents the states C to M (marked by dashed borders), we now remove them.

We continue symbolic evaluation in N, reaching state O, which is like C. In C, we refined our information to decide whether the condition `i >= j` of `if_icmpge` holds. However, now this case analysis cannot be expressed by simply refining the intervals from $\textsc{Integers}$ that correspond to the references i_6 and i_4 (i.e., a relation like $i_4 \geq i_6$ is not expressible in our states). Instead, we again generate successors for both possible values of the condition `i >= j`, but do not change the actual information about our values. In the resulting states P and Q, we mark the truth value of the condition `i >= j` by "T" and "F". The refinement edges from O to P and Q are marked by the references i_4 and i_6 that are refined. P leads to a program end, while we continue the symbolic evaluation in Q. As before, we label the refinement edge from Q to R by $i_4 < i_6$.

R and S are like F and G. The refinement edge from R to S is labeled by a_1 and i_4 which were refined in order to evaluate `aaload` (note that since we only reach R if $i_4 < i_6$, the array access succeeds). As in H, we then perform an instance refinement to decide whether calling `length()` on the object o_4 succeeds, leading to U and V. From V, we again reach a program end after a `NullPointerException` was thrown in W. From U, we reach X by evaluating the call to `length()`. Between X to Y, we return from `length()`. After that, we add the two non-negative integers i_7 and i_4, creating a non-negative integer i_8. The edge from Y to Z is labeled by the computation $i_8 = i_7 + i_4$.

Z is again an instance of N. We can also use the algorithm `mergeStates` to determine whether one state is an instance of another: When merging s, s' to obtain a new state $\hat{s}$, one adapts `mergeStates(`s`,`s'`)` such that the algorithm terminates with failure whenever we widen a value of s or add an annotation to $\hat{s}$ that did not exist in s (e.g., when we add $\hat{s}|_{\pi} =^? \hat{s}|_{\pi'}$ and there is no $s|_{\pi} =^? s|_{\pi'}$). Then the algorithm terminates successfully iff $s' \sqsubseteq s$ holds. After drawing the instance edge from Z to N (yielding a *cycle* in our graph), all leaves of the graph are program ends and thus the graph construction is finished.

We now define termination graphs formally. We extend our earlier definition from [4] slightly by labeling edges with information about the performed refinements and about the relations of integers. Let $\mathcal{R}el\mathcal{O}p = \{i \circ i' \mid i, i' \in \textsc{Refs}, \circ \in \{<, \leq, =, \neq, \geq, >\}\}$ denote the set of relations between two integer references such as $i_4 < i_6$ and $\mathcal{A}rith\mathcal{O}p = \{i = i' \bowtie i'' \mid i, i', i'' \in \textsc{Refs}, \bowtie \in \{+, -, *, /, \%\}\}$

[9] For all proofs, we refer to [5].

denote the set of arithmetic computations such as $i_8 = i_7 + i_4$.

Termination graphs are constructed by repeatedly expanding those leaves that do not correspond to program ends. Whenever possible, we use *symbolic evaluation* $\overset{SyEv}{\longrightarrow}$. Here, $\overset{SyEv}{\longrightarrow}$ extends the usual evaluation relation for JBC such that it can also be applied to *abstract* states representing several concrete states. For a formal definition of $\overset{SyEv}{\longrightarrow}$, we refer to [4, Def. 6]. In the termination graph, the corresponding *evaluation edges* can be labeled by a set $C \subseteq \mathcal{A}rith\mathcal{O}p \cup \mathcal{R}el\mathcal{O}p$ which corresponds to the arithmetic operations and (implicitly) checked relations in the evaluation. For example, when accessing the index `i` of an array `a` succeeds, we have implicitly ensured $0 \leq$ `i` and `i` $<$ `a.length` and this is noted in C.

If symbolic evaluation is not possible, we refine the information for some references R by case analysis and label the resulting *refinement edges* with R.

To obtain a *finite* graph, we create a more general state by *merging* whenever a program position is visited a second time in our symbolic evaluation and add appropriate *instance edges* to the graph. However, we require all cycles of the termination graph to contain at least one evaluation edge. By using an appropriate strategy for merging resp. widening states, we can automatically generate a finite termination graph for any program.

Definition 3 (Termination Graph). *A graph* (V, E) *with* $V \subseteq \textsc{States}$*,* $E \subseteq V \times \big((\{\textsc{Eval}\} \times 2^{\mathcal{A}rith\mathcal{O}p \cup \mathcal{R}el\mathcal{O}p}) \cup (\{\textsc{Refine}\} \times 2^{\textsc{Refs}}) \cup \{\textsc{Ins}\}\big) \times V$ *is a* termination graph *if every cycle contains at least one edge labeled with some* $\textsc{Eval}_C$ *and one of the following holds for each* $s \in V$*:*

- *s has just one outgoing edge $(s, \textsc{Eval}_C, s')$, $s \overset{SyEv}{\longrightarrow} s'$, and C is the set of integer relations that are checked (resp. generated) in this step*
- *the outgoing edges of s are $(s, \textsc{Refine}_R, s_1), \dots, (s, \textsc{Refine}_R, s_n)$ and $\{s_1, \dots, s_n\}$ is a refinement of s on the references $R \subseteq \textsc{Refs}$*
- *s has just one outgoing edge $(s, \textsc{Ins}, s')$ and $s \sqsubseteq s'$*
- *s has no outgoing edge and $s = (\varepsilon, e, h, a)$ (i.e., s is a program end)*

We refer to [6, 17] for methods to use termination graphs for termination proofs and to [4] for soundness proofs which show that if c is a concrete state with $c \sqsubseteq s$ for some state s in the termination graph, then the JBC evaluation of c is represented in the termination graph. In Sect. 3 and 4 we show how to use termination graphs to detect `NullPointerExceptions` and non-termination.

3 Generating Witnesses for `NullPointerExceptions`

In our example, an uncaught `NullPointerException` is thrown in the "error state" W, leading to a program end. Such violations of memory safety can be immediately detected from the termination graph. In particular, if the graph does not contain any such exceptions, then memory safety is proved.[10]

To report a possible violation of memory safety to the user, we now show

[10] In languages like C, *memory safety* (or *pointer safety*) means absence of (i) accesses to `null`, (ii) dangling pointers, and (iii) memory leaks [25]. In Java, (ii) and (iii) are ensured by the JVM and only `NullPointerExceptions` can destroy memory safety.

how to automatically generate a *witness* (i.e., an assignment to the arguments of the program) that leads to the exception. Our termination graph allows us to generate such witnesses automatically. This technique for witness generation will also be used to construct witnesses for non-termination in Sect. 4.

So our goal is to find a *witness state* A' for the initial state A of the method `main` *w.r.t. the "error state"* W. This state A' describes a subset of arguments, all of which lead to an instance of W, i.e., to a `NullPointerException`.

Definition 4 (Witness State). *Let* $s, s', w \in \textsc{States}$*. The state* s' *is a* witness state for s w.r.t. w *iff* $s' \sqsubseteq s$ *and* $s' \overset{SyEv}{\longrightarrow}{}^{*} w'$ *for some state* $w' \sqsubseteq w$.

To obtain a witness state A' for A automatically, we start with the error state W and traverse the edges of the termination graph backwards until we reach A. In general, let $s_0, s_1, \ldots, s_n = w$ be a path in the termination graph from the initial state s_0 to the error state s_n. Assuming that we already have a witness state s'_i for s_i w.r.t. w, we show how to generate a witness state s'_{i-1} for s_{i-1} w.r.t. w. To this end, we revert the changes done to the state s_{i-1} when creating the state s_i during the construction of the termination graph (i.e., we apply the rules for termination graph construction "backwards"). Of course, this generation of witness states can fail (in particular, this happens for error states that are not reachable from any concrete instantiation of the initial state s_0). So in this way, our technique for witness generation is also used as a check whether certain errors can really result from initial method calls.

In our example, the error state is W. Trivially, W itself is a witness state for W w.r.t. W. The only edge leading to W is from V. Thus, we now generate a witness state V' for V w.r.t. W. The edge from V to W represents the evaluation of the instruction `invokevirtual` that triggered the exception. Reversing this instruction is straightforward, as we only have to remove the exception object from W again. Thus, V is a witness state for V w.r.t. W.

The only edge leading to V is a refinement edge from T. As a refinement corresponds to a case analysis, the information in the target state is more precise. Hence, we can reuse the witness state for V, since V is an instance of T. So V is also a witness state for T w.r.t. W.

13| a : a_2, i : 0, j : 1 | 0, a_2, 0
a_2: String[] 1 {null}

Fig. 6. State R'

To reverse the edge between T and S, we have to undo the instruction `aaload`. This is easy since S contains the information that the entry at index i_4 in the array a_1 is o_4. Thus the witness state S' for S w.r.t. W is like S, but here o_4's value is not an unknown object, but `null`. Reversing the refinement between S and R is more complex. Note that not every state represented by R leads to a `NullPointerException`. In S we had noted the relation between the newly created reference o_4 and the original array a_1. In other words, in S we know that $a_1[i_4]$ is o_4, where o_4 has the value `null` in the witness state S' for S. But in R, o_4 is missing. To solve this problem, in the witness state R' for R, we instantiate the abstract array a_1 by a concrete one that contains the entry `null` at the index i_4. We use a simple heuristic[11] to choose a suitable

[11] Such heuristics cannot affect soundness, but just the power of our approach (choosing unsuitable values may prevent us from finding a witness for the initial state).

length i_6 for this concrete array, which tries to find "minimal" values. Here, our heuristic chooses a_1 to be an array of length one (i.e., i_6 is chosen to be 1), which only contains the entry `null` (at the index 0, i.e., i_4 is chosen to be 0). The resulting witness state R' for R w.r.t. W is displayed in Fig. 6.

Reversing the evaluation steps between R and Q yields a witness state Q' for Q w.r.t. W. From O to Q, we have a refinement edge and thus, Q' is also a witness for O.

0| `a`: a_2 | ε
a_2: `String[]` 1 `{null}`

Fig. 7. State A'

The steps from N to O can also be reversed easily. In N, we use a heuristic to decide whether to follow the incoming edge from Z or from B. Our heuristic chooses B as it is more concrete than Z. From there, we continue our reversed evaluation until we reach a witness state A' for the initial state A of the method w.r.t. W, cf. Fig. 7. So any instance of A' evaluates to an instance of W, i.e., it leads to a `NullPointerException`. If the `main` method is called directly (as the entry point of the program), then the JVM ensures that the input array does not contain `null` references. But if the `main` method is called from another method, then this violation of memory safety can indeed occur, cf. problem (a) in Sect. 2.

The following theorem summarizes our procedure to generate witness states. If there is an edge from a state s_1 to a state s_2 in the termination graph and we already have a witness state s_2' for s_2 w.r.t. w, then Thm. 5 shows how to obtain a witness state s_1' for s_1 w.r.t. w. Hence, by repeated application of this construction, we finally obtain a witness state for the initial state of the method w.r.t. w. If there is an *evaluation edge* from s_1 to s_2, then we first apply the reversed rules for symbolic evaluation on s_2'. Afterwards, we instantiate the freshly appearing references (for example, those overwritten by the forward symbolic evaluation) such that s_1' is indeed an instance of s_1. If there is a *refinement edge* from s_1 to s_2, then the witness state s_1' is like s_2', but when reading from abstract arrays (such as between R and S), we instantiate the array to a concrete one in s_1'. If there is an *instance edge* from s_1 to s_2, then we *intersect* the states s_1 and s_2' to obtain a representation of those states that are instances of both s_1 and s_2'.

Theorem 5 (Generating Witnesses). *Let (s_1, l, s_2) be an edge in the termination graph and let s_2' be a witness state for s_2 w.r.t. w. Let $s_1' \in \textsc{States}$ with:*

- *if $l = \textsc{Eval}_C$, then s_1' is obtained from s_2' by applying the symbolic evaluation used between s_1 and s_2 backwards. In s_1', we instantiate freshly appearing variables such that $s_1' \sqsubseteq s_1$ and $s_1' \overset{SyEv}{\longrightarrow} s_2'$ holds.*
- *if $l = \textsc{Refine}_R$, then $s_1' \sqsubseteq s_2'$.*
- *if $l = \textsc{Ins}$, then $s_1' = s_1 \sqcap s_2'$ (for the definition of $\sqcap$, see [6, Def. 2]).*

Then s_1' is a witness state for s_1 w.r.t. w.

4 Proving Non-Termination

Now we show how to prove non-termination automatically. Sect. 4.1 introduces a method to detect *looping* non-termination, i.e., infinite evaluations where the *interesting references* (that determine the termination behavior) are unchanged. Sect. 4.2 presents a method which can also detect *non-looping* non-termination.

4.1 Looping Non-Termination

For each state, we define its *interesting* references that determine the control flow and hence, the termination behavior. Which references are interesting can be deduced from the termination graph, because whenever the (changing) value of a variable may influence the control flow, we perform a refinement. Hence, the references in the labels of refinement edges are "interesting" in the corresponding states. For example, the references i_4 and i_6 are interesting in the state O.

We propagate the information on interesting references backwards. For evaluation edges, those references that are interesting in the target state are also interesting in the source state. Thus, i_4 and i_6 are also interesting in N.

When drawing refinement or instance edges, references may be renamed. But if a reference at position π is interesting in the target state of such an edge, the reference at π is also interesting in the source state. So $i_8 = Z|_{\text{LV}_{0,1}}$ and $i_6 = Z|_{\text{LV}_{0,2}}$ are interesting in Z, as $i_4 = N|_{\text{LV}_{0,1}}$ and $i_6 = N|_{\text{LV}_{0,2}}$ are interesting in N.

Furthermore, if an interesting reference i of the target state was the result of some computation (i.e., the evaluation edge is labeled with $i = i' \bowtie i''$), we mark i' and i'' as interesting in the source state. The edge from Y to Z has the label $i_8 = i_7 + i_4$. As i_8 is interesting in Z, i_7 and i_4 are interesting in Y.

Definition 6 (Interesting References). *Let $G = (V, E)$ be a termination graph, and let $s, s' \in V$ be some states. Then $I(s) \subseteq \{s|_\pi \mid \pi \in \text{SPos}(s)\}$ is the set of* interesting references *of s, defined as the minimal set of references with*

- *if $(s, \text{Refine}_R, s') \in E$, then $R \subseteq I(s)$.*
- *if $(s, l, s') \in E$ with $l \in \{\text{Refine}_R, \text{Ins}\}$, then we have $\{s|_\pi \mid \pi \in \text{SPos}(s) \cap \text{SPos}(s'), s'|_\pi \in I(s')\} \subseteq I(s)$.*
- *if $(s, \text{Eval}_C, s') \in E$, then $I(s') \cap \{s|_\pi \mid \pi \in \text{SPos}(s)\} \subseteq I(s)$.*
- *if $(s, \text{Eval}_C, s') \in E$, $i = i' \bowtie i'' \in C$ and $i \in I(s')$, then $\{i', i''\} \subseteq I(s)$.*

Note that if there is an evaluation where the same program position is visited repeatedly, but the values of the interesting references do not change, then this evaluation will continue infinitely. We refer to this as *looping* non-termination.

To detect such non-terminating loops, we look at cycles $s = s_0, s_1, \ldots, s_{n-1}, s_n = s$ in the termination graph. Our goal is to find a state $v \sqsubseteq s$ such that when executing the loop, the values of the interesting references in v do not change. More precisely, when executing the loop in v, one should reach a state v' with $v' \sqsubseteq_\Pi v$. Here, Π are the positions of interesting references in s and $\sqsubseteq_\Pi$ is the "instance" relation restricted to positions with prefixes from Π, whereas the values at other positions are ignored. The following theorem proves that if one finds such a state v, then indeed the loop will be executed infinitely many times when starting the evaluation in a concrete instance of v.

Theorem 7 (Looping Non-Termination). *Let s occur in a cycle of the termination graph. Let $\Pi = \{\pi \in \text{SPos}(s) \mid s|_\pi \in I(s)\}$ be the positions of interesting references in s. If there is a $v \sqsubseteq s$ where $v \xrightarrow{SyEv}{}^{+} v'$ for some $v' \sqsubseteq_\Pi v$, then any concrete state that is an instance of v starts an infinite JBC evaluation.*

We now automate Thm. 7 by a technique consisting of four steps (the first three steps find suitable states v automatically and the fourth step checks whether v can be reached from the initial state of the method). Let $s = s_0, s_1, \ldots, s_{n-1}, s_n = s$ be a cycle in the termination graph such that there is an instance edge from s_{n-1} to s_n. In Fig. 4, $N, \ldots Z, N$ is such a cycle (i.e., here s is N).

1. Find suitable values for interesting integer references. In the first step, we find out how to instantiate the interesting references of *integer* type in v. To this end, we convert the cycle $s = s_0, \ldots, s_n = s$ edge by edge to a formula φ over the integers. Then every model of φ indicates values for the interesting integer references that are not modified when executing the loop.

Essentially, φ is a conjunction of all constraints that the edges are labeled with. More precisely, to compute φ, we process each edge (s_i, l, s_{i+1}). If l is $\textsc{Refine}_R$, then we connect the variable names in s_i and s_{i+1} by adding the equations $s_i|_\pi = s_{i+1}|_\pi$ to φ for all those positions π where $s_i|_\pi$ is in R and points to an integer. Thus, for the edge from O to Q, we add the trivial equations $i_4 = i_4 \wedge i_6 = i_6$, as the references were not renamed in this refinement step.

If $l = \textsc{Eval}_C$, we add the constraints and computations from C to the formula φ.[12] Thus, for the edge from Q to R we add the constraint $i_4 < i_6$, for the edge from S to T we add $0 \leq i_4 \wedge i_4 < i_6$, and the edge from Y to Z yields $i_8 = i_7 + i_4$. If l is $\textsc{Ins}$, we again connect the reference names in s_i and s_{i+1} by adding the equations $s_i|_\pi = s_{i+1}|_\pi$ for all $\pi \in \mathrm{S}\textsc{Pos}(s_{i+1})$ that point to integers. Thus, for the edge from Z to N, we get $i_6 = i_6 \wedge i_8 = i_4$. So for the cycle $N, \ldots, Z, N$, φ is $i_4 < i_6 \wedge 0 \leq i_4 \wedge i_8 = i_7 + i_4 \wedge i_8 = i_4$ (where tautologies have been removed).

To find values for the integer references that are not modified in the loop, we now try to synthesize a model of φ. In our example, a standard SMT solver easily proves satisfiability and returns a model like $i_4 = 0$, $i_6 = 1$, $i_7 = 0$, $i_8 = 0$.

2. Guess suitable values for interesting non-integer references. We want to find a state $v \sqsubseteq s$ such that executing the loop does not change the values of interesting references in v. We have determined the values of the interesting *integer* references in v (i.e., i_4 is 0 and i_6 is 1 in our example). It remains to determine suitable values for the other interesting references (i.e., for a_1 in our example)

$$\boxed{\begin{array}{l}\texttt{05}|\,\texttt{a}:a_3,\texttt{i}:0,\texttt{j}:1\,|\,\varepsilon\\ a_3:\texttt{String}[\,]\;1\end{array}}$$

Fig. 8. State Z'

To this end, we use the following heuristic. We instantiate the integer references in s_{n-1} according to the model found for φ, yielding a state $s'_{n-1} \sqsubseteq s_{n-1}$. So in our example (where $s_n = s$ is N and s_{n-1} is Z), we instantiate i_6 and i_8 in Z by 1 resp. 0, resulting in the state Z' in Fig. 8 (i.e., here s'_{n-1} is Z').

$$\boxed{\begin{array}{l}\texttt{05}|\,\texttt{a}:a_3,\texttt{i}:0,\texttt{j}:1\,|\,\varepsilon\\ a_3:\texttt{String}[\,]\;1\;\{o_6\}\\ o_6:\texttt{String}(\texttt{count}=\texttt{0},\ldots)\end{array}}$$

Fig. 9. State N'

Afterwards, we traverse the path from s_{n-1} backwards to s_0 and use the technique of witness generation from Sect. 3 to generate a witness v for s_0 w.r.t. s'_{n-1} (i.e., $v \sqsubseteq s_0$ such that $v \overset{SyEv}{\longrightarrow}{}^{+} v'$ for some $v' \sqsubseteq s'_{n-1}$). In our example,[13] the

[12] Remember that we use a single static assignment technique. Thus, we do not have to perform renamings to avoid name clashes.

[13] During the witness generation, one again uses the model of φ for intermediate integer references. So when reversing the `iadd` evaluation between Y and Z, we choose 0 as value for the newly appearing reference i_7.

witness generation results in the state N' in Fig. 9. Note that the witness generation technique automatically "guessed" a suitable instantiation for the array (i.e., it was instantiated by a 1-element array containing just the empty string). Indeed, $N' \sqsubseteq N$ and $N' \overset{SyEv}{\longrightarrow}{}^{+} v'$ for an instance v' of Z' (i.e., in our example $s_0 = s$ is N and v is N'). Here, v' is like Z', but instead of "a_3:`String`[ ] 1", we have "a_3:`String`[ ] 1 $\{o_6\}$" and "o_6:`String`(`count` = `0`, . . .)". Thus, $v' = N'$.

3. Check whether the guessed values for non-integer references do not change in the loop. While our construction ensures that the interesting *integer* references remain unchanged when executing the loop, this is not ensured for the interesting non-integer references. Hence, in the third step, we now have to check whether $v' \sqsubseteq_{\Pi} v$ holds, where Π are the positions of interesting references in s.

To this end, we adapt our algorithm `mergeStates(`v`,`v'`)` such that it terminates with failure whenever we widen a value of v or add an annotation that did not exist in v at a position with a prefix from Π. Then the algorithm terminates successfully iff $v' \sqsubseteq_{\Pi} v$. In our example where $v = v' = N$, we clearly have $v' \sqsubseteq_{\Pi} v$. Hence by Thm. 7, any instance of v (i.e., of N') starts an infinite execution.

4. Check whether the non-terminating loop can be reached from the initial state. In the fourth step, we finally check whether N' can be reached from the initial state of the method. Hence, we again use the witness generation technique from Sect. 3 to create a witness state for A w.r.t. N'. This witness state has the stack frame "`00` | `a` : a_3 | ε" where a_3 is a 1-element array containing just the empty string. In other words, we automatically synthesized the counterexample to termination indicated in problem (b) of Sect. 2.

4.2 Non-Looping Non-Termination

```
static void nonLoop(
  int x, int y) {
   if (y >= 0) {
     while(x >= y) {
       int z = x - y;
       if (z > 0) {
         x--;
       } else {
         x = 2*x + 1;
         y++; }}}}
```

Fig. 10. `nonLoop(x,y)`

A loop can also be non-terminating if the values of interesting references are modified in its body. We now present a method to find such *non-looping* forms of non-termination. In contrast to the technique of Sect. 4.1, this method is restricted to loops that have no sub-loops and whose termination behavior only depends on integer arithmetic (i.e., the interesting references in all states of the loop may only refer to integers). Then we can construct a formula that represents the loop condition and the computation on each path through the loop. If we can prove that no variable assignment that satisfies the loop condition violates it in the next loop iteration, then we can conclude non-termination under the condition that the loop condition is satisfiable.

The method `nonLoop` in Fig. 11 does not terminate if $\texttt{x} \geq \texttt{y} \geq 0$. For example, if $\texttt{x} = 2$, $\texttt{y} = 1$ at the beginning of the loop, then after one iteration we have $\texttt{x} = 1$, $\texttt{y} = 1$. In the next iterations, we obtain $\texttt{x} = 3$, $\texttt{y} = 2$; $\texttt{x} = 2$, $\texttt{y} = 2$; and $\texttt{x} = 5$, $\texttt{y} = 3$, etc. So this non-termination is non-looping and even *non-periodic* (since there is no fixed sequence of program positions that is repeated infinitely many times). Thus, non-termination cannot be proved by techniques like [14].

Consider the termination graph, which is shown in a simplified version in

Fig. 11. A node in a cycle with a predecessor outside of the cycle is called a *loop head node.* In Fig. 11, A is such a node. We consider all paths $p_1, \dots, p_n$ from the loop head node back to itself (without traversing the loop head node in between), i.e., $p_1 = A, \dots, B, A$ and $p_2 = A, \dots, C, A$. Here, p_1 corresponds to the case where $\mathtt{x} \geq \mathtt{y}$ and $\mathtt{z} = \mathtt{x} - \mathtt{y} > 0$, whereas p_2 handles the case where $\mathtt{x} \geq \mathtt{y}$ and $\mathtt{z} = \mathtt{x} - \mathtt{y} \leq 0$. For each path p_j, we generate a *loop condition formula* φ_j (expressing the condition for entering this path) and a *loop body formula* ψ_j (expressing how the values of the interesting references are changed in this path).

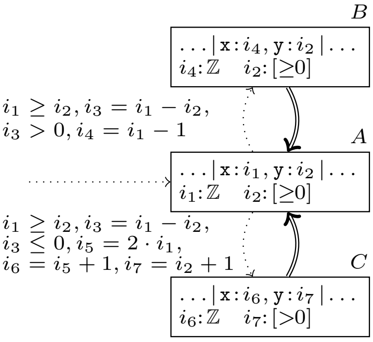

B
$\dots | \mathtt{x}: i_4, \mathtt{y}: i_2 | \dots$
$i_4{:}\,\mathbb{Z} \quad i_2{:}\,[\geq 0]$
$i_1 \geq i_2, i_3 = i_1 - i_2,$
$i_3 > 0, i_4 = i_1 - 1$
A
$\dots | \mathtt{x}: i_1, \mathtt{y}: i_2 | \dots$
$i_1{:}\,\mathbb{Z} \quad i_2{:}\,[\geq 0]$
$i_1 \geq i_2, i_3 = i_1 - i_2,$
$i_3 \leq 0, i_5 = 2 \cdot i_1,$
$i_6 = i_5 + 1, i_7 = i_2 + 1$
C
$\dots | \mathtt{x}: i_6, \mathtt{y}: i_7 | \dots$
$i_6{:}\,\mathbb{Z} \quad i_7{:}\,[> 0]$

Fig. 11. Graph for `nonLoop`

The formulas φ_j and ψ_j are generated as in Step 1 of Sect. 4.1, where we add relations from $\mathcal{R}el\mathcal{O}p$ to φ_j and constraints from $\mathcal{A}rith\mathcal{O}p$ to ψ_j. In our example, φ_1 is $i_1 \geq i_2 \land i_3 > 0$ and φ_2 is $i_1 \geq i_2 \land i_3 \leq 0$. Moreover, ψ_1 is $i_3 = i_1 - i_2 \land i_4 = i_1 - 1$ and ψ_2 is $i_3 = i_1 - i_2 \land i_5 = 2 \cdot i_1 \land i_6 = i_5 + 1 \land i_7 = i_2 + 1$. To connect these formulas, we use a labeling function ℓ^k where for any formula ξ, $\ell^k(\xi)$ results from ξ by labeling all variables with ^{k}. We use the labels $^1, \dots, ^n$ for the paths through the loop and the label ^{r} for the **r**esulting variables (in the second run, leaving the loop). We construct the formula

$$\rho(p_1, \dots, p_n) = \underbrace{\mu}_{\text{invariants}} \land \underbrace{(\bigvee\nolimits_{j=1}^{n} (\ell^j(\varphi_j) \land \ell^j(\psi_j) \land \iota_j))}_{\text{first run through the loop}} \land \underbrace{(\bigwedge\nolimits_{j=1}^{n} (\neg \ell^r(\varphi_j) \land \ell^r(\psi_j)))}_{\text{second run, leaving the loop}}$$

Here, μ is a set of invariants that are known in the loop head node. So as we know "$i_2{:}\,[\geq 0]$" in state A, μ is $i_2 \geq 0$ for our example. The formula ι_j connects the variables labeled with ^{j} to the unlabeled variables in μ and to the variables labeled with ^{r} in the formulas for the second iteration. So for every integer reference i in the loop head node, ι_j contains $i = i^j$. Moreover, if i is an integer reference at position π in the loop head node s and i' is at position π in the predecessor s' of s (where there is an instance edge from s' to s), then ι_j contains $i'^j = i^r$. For our example, ι_1 is $i_1 = i_1^1 \land i_2 = i_2^1 \land i_4^1 = i_1^r \land i_2^1 = i_2^r$.

Intuitively, satisfiability of the first two parts of $\rho(p_1, \dots, p_n)$ corresponds to one successful run through the loop. The third part encodes that none of the loop conditions holds in the next run. Here, we do not only consider the negated conditions $\neg \ell^r(\varphi_j)$, but we also need $\ell^r(\psi_j)$, as φ_j can contain variables computed in the loop body. For example in `nonLoop`, $\ell^r(\varphi_1)$ contains $i_3^r > 0$. But to determine how i_3^r results from the "input arguments" i_1^r, i_2^r, one needs $\ell^r(\psi_1)$ which contains $i_3^r = i_1^r - i_2^r$. If an SMT solver proves unsatisfiability of $\rho(p_1, \dots, p_n)$, we know that whenever a variable assignment satisfies a loop condition, then after one execution of the loop body, a loop condition is satisfied again (i.e., the loop runs forever). Note that we generalized the notion of "loop conditions", as we discover the conditions by symbolic evaluation of the loop. Consequently, we can also handle loop control constructs like `break` or `continue`.

So unsatisfiability of $\rho(p_1, \dots, p_n)$ implies that the loop is non-terminating, provided that the loop condition can be satisfied at all. To check this, we use an SMT solver to find a model for $\sigma(p_1, \dots, p_n) = \mu \land (\bigvee_{j=1}^{n} (\ell^j(\varphi_j) \land \ell^j(\psi_j) \land \iota_j))$.

Theorem 8 (Non-Looping Non-Termination). *Let s be a loop head node in a termination graph where $I(s)$ only point to integer values and let $p_1, \ldots, p_n$ be all paths from s back to s. Let $\rho(p_1, \ldots, p_n)$ be unsatisfiable and let $\sigma(p_1, \ldots, p_n)$ be satisfiable by some model M (i.e., M is an assignment of integer references to concrete integers). Let $c \sqsubseteq s$ be a concrete state where every integer reference in c has been assigned the value given in M. Then c starts an infinite JBC evaluation.*

From the model M of $\sigma(p_1, \ldots, p_n)$, we obtain an instance v of the loop head node where we replace unknown integers by the values in M. Then the technique from Sect. 3 can generate a witness for the initial state of the method w.r.t. v. For our example, $i_1 = i_1^1 = i_3^1 = 1, i_2 = i_2^1 = i_2^r = i_4^1 = i_1^r = 0$ satisfies $\sigma(p_1, \ldots, p_n)$. From this, we obtain a witness for the initial state with $\texttt{x} = 1$ and $\texttt{y} = 0$, i.e., we automatically generate a non-terminating counterexample.

5 Evaluation and Conclusion

Based on termination graphs for Java Bytecode, we presented a technique to generate witnesses w.r.t. arbitrary error states. We then showed how to use this technique to prove the reachability of `NullPointerException`s or of non-terminating loops, which we detect by a novel SMT-based technique.

We implemented our new approach in the termination tool AProVE [12], using the SMT solver Z3 [10] and evaluated it on a collection of 325 examples. They consist of all 268 JBC programs from the *Termination Problem Data Base* that is used in the annual *International Termination Competition*,[14] all 55 examples from [24] used to evaluate the Invel tool, and the two examples from this paper. For our evaluation, we compared the old version of AProVE (without support for non-termination), the new version AProVE-No containing the results of the present paper, and Julia [20]. We were not able to obtain a running version of Invel, and thus we only compared to the results of Invel reported in [24].

We used a time-out of 60 seconds for each example. "**Y**es" and "**N**o" indicate how often termination (resp. non-termination) could be proved, "**F**ail" states how often the tool failed in less than 1 minute, "**T**" indicates how many examples led to a **T**ime-out, and "**R**" gives the average **R**untime in seconds for each example. The experiments clearly show the power of our contributions, since AProVE-No is the most powerful tool for automated non-termination proofs of Java resp. JBC programs. Moreover, the comparison between AProVE-No and AProVE indicates that the runtime for termination proofs did not increase due to the added non-termination techniques. To experiment with our implementation via a web interface and for details on the experiments, we refer to [1].

	Invel Ex.					Other Ex.				
	Y	**N**	**F**	**T**	**R**	**Y**	**N**	**F**	**T**	**R**
AProVE-No	1	51	0	3	5	204	30	12	24	11
AProVE	1	0	5	49	54	204	0	27	39	15
Julia	1	0	54	0	2	166	22	82	0	4
Invel	0	42	13	0	?					

References

1. http://aprove.informatik.rwth-aachen.de/eval/JBC-Nonterm/.

[14] We removed a controversial example whose termination depends on integer overflows.

2. N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.
3. B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. 2007.
4. M. Brockschmidt, C. Otto, C. von Essen, and J. Giesl. Termination graphs for Java Bytecode. In *Verification, Induction, Termination Analysis*, LNCS 6463, pages 17–37, 2010. Extended version (with proofs) available at [1].
5. M. Brockschmidt, T. Ströder, C. Otto, and J. Giesl. Automated detection of non-termination and `NullPointerException`s for Java Bytecode. Report AIB 2011-19, RWTH Aachen, 2011. Available at [1] and at `aib.informatik.rwth-aachen.de`.
6. M. Brockschmidt, C. Otto, and J. Giesl. Modular termination proofs of recursive Java Bytecode programs by term rewriting. In *Proc. RTA '11*, LIPIcs 10, pages 155–170, 2011. Extended version (with proofs) available at [1].
7. R. Bubel, R. Hähnle, and R. Ji. Interleaving symbolic execution and partial evaluation. In *Proc. FMCO '09*, LNCS 6286, pages 247–277, 2010.
8. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL '77*, pages 238–252. ACM Press, 1977.
9. C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17:8:1–8:37, 2008.
10. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS '08*, LNCS 4963, pages 337–340, 2008.
11. J. Giesl, R. Thiemann, P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proc. FroCoS '05*, LNAI 3717, pages 216–231, 2005.
12. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proc. IJCAR '06*, LNAI 4130, pages 281–286, 2006.
13. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. PLDI '05*, pages 213–223. ACM Press, 2005.
14. A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R. Xu. Proving non-termination. In *Proc. POPL '08*, pages 147–158. ACM Press, 2008.
15. L. Hubert, T. Jensen, and D. Pichardie. Semantic foundations and inference of non-`null` annotations. In *Proc. FMOODS '08*, LNCS 5051, pages 132–149, 2008.
16. T. Lindholm and F. Yellin. *Java Virtual Machine Specification.* Prentice Hall, 1999.
17. C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *Proc. RTA '10*, LIPIcs 6, pages 259–276, 2010. Extended version (with proofs) available at [1].
18. É. Payet and F. Mesnard. Nontermination inference of logic programs. *ACM Trans. Prog. Lang. Syst.*, 28:256–289, 2006.
19. É. Payet. Loop detection in term rewriting using the eliminating unfoldings. *Theoretical Computer Science*, 403:307–327, 2008.
20. É. Payet and F. Spoto. Experiments with non-termination analysis for Java Bytecode. In *Proc. BYTECODE '09*, ENTCS 5, pages 83–96, 2009.
21. K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. FSE '10*, pages 263–272. ACM Press, 2005.
22. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In *Proc. ILPS '95*, pages 465–479. MIT Press, 1995.
23. F. Spoto. Precise `null`-pointer analysis. *Softw. Syst. Model.*, 10:219–252, 2011.
24. H. Velroyen and P. Rümmer. Non-termination checking for imperative programs. In *Proc. TAP '08*, LNCS 5051, pages 154–170, 2008.
25. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn. Scalable shape analysis for systems code. *Proc. CAV '08*, LNCS 5123, p. 385–398, 2008.

Trust the clones

Sophia Drossopoulou[1] and James Noble[2]

[1] Department of Computing,
Imperial College, London
scd@doc.ic.ac.uk
[2] School of Mathematics, Statistics, and Computer Science,
Victoria University of Wellington,
Wellington, New Zealand
kjx@mcs.vuw.ac.nz

Abstract. Most object-oriented programming languages provide some way to clone objects — to produce a new object that is a copy of an existing object. Typically this is either a shallow clone that clones only one object, or a deep clone that clones objects recursively. In practice, programmers have to write custom cloning methods for each of their classes, and this boilerplate code is tedious to write and requires care to write correctly. Inspired by ownership types, we propose a system that annotates classes to express how their instances should be cloned. Furthermore, we show how these annotations can be used to generate cloning methods for these classes.

1 Introduction

Background Ownership types were first suggested in 1998; their aim was to characterise aliases, and thus to control the topology of objects on the heap [19]. Since then, they have attracted tremendous interest; they have been developed in several brands and variations, and they have been put to several different uses, e.g. memory management, encapsulation, effect systems, avoidance of race conditions, locations, parallel programming, program verification etc.

One example given in the seminal ECOOP paper [19], was cloning, whereby all the objects "inside" a certain other object would automatically be copied when the enclosing object was copied.

The aim of supporting cloning, although appealing, has — to our knowledge — not been further pursued in the context of static ownership types. Interestingly, this aim was recently pursued in the static analysis world [14], whereby copy policies are expressed by annotating code so as to specify the maximally allowed sharing between an object and its clone, and a type and effects system checks whether the copy policy is adhered to.

In this paper we propose the opposite approach: the copy code is *generated* out of *clone annotations* given with the types of fields, and thus, adherence to the copy policy is automatically guaranteed.

Overview of our approach The key problem we aim to address is determining which objects to copy. Generally, the set of objects which have to be cloned when cloning an object o does not solely depend on o — it also depends on the object which started the the cloning process — the "originator" of the cloning.

Consider the object structure shown in Fig. 1, where the small boxes represent objects of a class as written in the box, and the labelled arrows represent fields. For example, we have an object of class `Union`, with a field called `students` which points to an object of class `StudentList`.

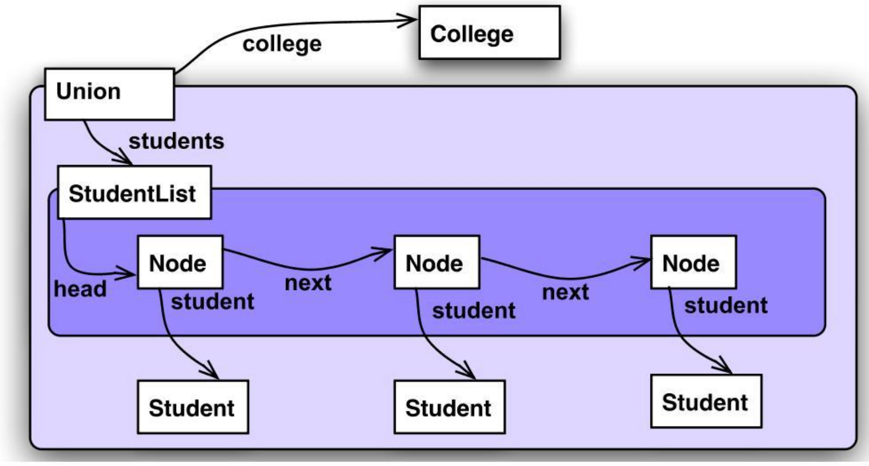


Fig. 1. An example of cloning domains

Cloning a `Node` object when the originator is itslef requires only the current object (the `Node`) to be cloned. Cloning a `Node` as a result of cloning a `StudentList` requires cloning all the successors of that node too, so that all the `Nodes` in the list are cloned. On the other hand, cloning the `StudentList` should not clone the `Students` pointed to by the `Nodes` in the list — however, if the `StudentList` is cloned as part of cloning the `Union` object, then the `Students` should also be cloned. Therefore, we say that the `Node` objects belong to the "cloning domain" of the `StudentList` object, as well as that of the `Union` object, while the `StudentList` and the `Student` objects belong to the cloning domain of the `Union` object.

The two cloning domains are depicted graphically in Fig. 1, through the two shaded rounded boxes. The cloning domains introduce a hierarchy among objects: when e.g. o belongs to the cloning domain of o' and o' belongs to the cloning domain of o'', then o belongs to the cloning domain of o''.

The hierarchy shown in Fig. 1 is very similar to that shown in various works on ownership types. In both cases, objects are put into boxes, and the boxes belong to objects. The difference is in the meaning of these boxes. In some cases, the boxes guarantee encapsulation[24], while in others they guarantee domination[8], or restrict who may modify the objects[16], or guarantee common de-allocation[1]. In our current work, the boxes guarantee that cloning the object which owns the box will also clone all objects in that particular box.

Note that the diagram in Fig. 1 presents a "ghost view" of the objects; the boxes, *i.e.,* the cloning domain information is *not* available at runtime. All the information we have is that `StudentList` will have a pointer field called `head` that points to its nodes; those `Nodes` have two pointer fields, one called `next` that points to the next `Node` and another called `student` that points to its `Student`.

To clone an object, we have to answer this question: for each object we reach, considering each of its fields in turn, *without* the runtime topological ownership information shown in Fig. 1: should we clone the object to which the field refers?

Organisation of our paper In Sect. 2 we describe our solution — clone annotated types — and give an example. In Sect. 3 we describe how clone annotated types can be used to decide whether or not a particular pointer should be cloned, and then in Sect. 4 we show how to generate the appropriate cloning methods. In Sect. 5 we describe the guarantees provided by our approach. In Sect. 6 we discuss related work and conclude.

2 Cloning Annotations

We propose to answer the question by introducing *cloning annotations* on field types. These annotations follow the tradition of ownership types [6, 8]. Every class is defined so as to take one or more formal *cloning parameters*, *c*. Thus, cloning annotated types have the form `C<c1,...cn>`, where `C` is a class, and `c1`, ... `cn` are the cloning parameters. The type `C<c1,..,cn>` expresses that the object has class `C`, and that it will be cloned whenever the object standing for `c1` is cloned; the remaining copy annotations, `c2`,..,`cn` may be used to annotate fields from that object and so describe when they are to be cloned.

The class definition determines the scope of these parameters — similarly to generic parameters. As in traditional ownership types, these parameters stand for objects. Again as in traditional ownership type systems, types are formed by classes followed by *actual cloning parameters*, *ca*, which may be any cloning parameter which is in scope, or `this`. The syntax is shown in Fig. 2.

$$
\begin{array}{lll}
\mathit{ClassDecl} ::= \texttt{class C}\langle\overline{c}\rangle & & \\
\qquad\qquad\quad \{\ \overline{\mathit{FieldDecl}}\ \overline{\mathit{MethDecl}}\ \} & & \\
\mathit{FieldDecl} ::= \mathit{FieldType}\ \texttt{f} & & \\
\mathit{FieldType} ::= \texttt{C}\langle\overline{ca}\rangle & & \\
c \qquad\quad ::= \mathit{Identifier} & & \text{clone parameters} \\
ca \qquad\ \ ::= c \mid \texttt{this} & & \text{clone arguments}
\end{array}
$$

Fig. 2. Syntax extracts

Note that the syntax of clone arguments does not allow any annotations which indicate that the cloning domain is unknown; thus, we do not support annotations such as norep from [10], or any from [9] or existentially bound arguments as in [2].

We use the term *cloning domain* of an object *o* to describe all the objects which have to be cloned when *o* is cloned. The first cloning parameter, $\texttt{c}_1$, in the annotation of a field f determines that the object pointed at by f belongs to the cloning domain of $\texttt{c}_1$: thus f must be cloned whenever $\texttt{c}_1$ is cloned. The remaining cloning parameters are used to determine cloning for the fields of f.

```
class Node<c1, c2>{
   Node<c1,c2> next;
   Student<c2> student;
}
class StudentList<c>{
   Node<this,c> head;
}
class Union<c>{
   StudentList<this> students;
   College<c> college;
}
```

Fig. 3. Lists example

Fig. 3 shows cloning annotated class declarations that match the structure from Fig. 1. When a Node is cloned, then a new object of class Node needs to be created, and its fields need to be initialised according to those of the old object. When a StudentList is cloned, then a new object of class StudentList needs to be created, and *all* accessible Nodes will have to be cloned — this is denoted by the head field's first cloning parameter being this. Finally, when a Union is cloned, then the StudentList will be cloned, all accessible Nodes and all accessible Students will have to be cloned.

3 Cloning Domains and Cloning Paths

Cloning domains have no runtime representation. In order to clone objects, we can only access their subordinate objects via *cloning paths* traced through objects' fields.

Inspecting Fig. 1 and Fig. 3 we can see how cloning paths lead to objects in different cloning domains. Consider paths beginning from a StudentList object. The objects at the paths this.head, or this.head.next, or this.head.next.next belong to the cloning domain of that StudentList object, but the object reachable through this.head.next.student does not. For paths starting at a Union object, objects reachable at this.students, or at this.students.head, or through `this.students.head.next`, and finally also at `this.students.head.next.next.student` belong to the cloning domain of the `Union` object.

Note that cloning domains are nested, and thus object may belong to more than one cloning domain. For example, for a `Union` object, `this.students`, `this.students.head` and `this.students.head.next` all belong to the `Union`'s cloning domain. On the other hand, the objects at `this.students.head` and `this.students.head.next` also belong to the cloning domain of the `StudentList` at the `this.students`, as well as to the `Union`'s cloning domain. Furthermore, it is possible for an object to be reachable from only one other object, but not belong to its cloning domain. For example, `this.head.next` is only reachable via `this.head`, but it does not belong to `this.head`'s cloning domain.

We now formalise the notion of cloning paths and cloning domains. We first define paths and path types as follows:

$p \quad ::= \texttt{this} \mid p.\texttt{f}$ paths
$PT ::= \texttt{C}\langle\overline{cap}\rangle$ path types
$cap ::= ca \mid p$ actual path cloning parameters

Path types express types which are relative to the current object `this`, and where paths can be used as actual cloning parameters. For example $\texttt{Node}\langle\texttt{this.students},\texttt{this}\rangle$ describes an object of class `Node`, which belongs to the cloning domain of the object at `this.students`, and whose `student` field belongs to the cloning domain of `this`.

In Fig. 4 we define a judgment of the form $\texttt{C} \vdash p : \texttt{D}\langle cap_1, ...cap_n\rangle$ which says that from an object of class `C`, the path p leads to an object in the cloning domain of cap_1. Furthermore, the actual path cloning parameters cap_1, ... cap_n may be used to characterize the cloning domain of $p.f$, where f is a field of p.

$$\frac{\texttt{class C}\langle\texttt{c}_1, ...\texttt{c}_\texttt{n}\rangle\{\ \ ...\ \ \}}{\texttt{C} \vdash \texttt{this} : \texttt{C}\langle\texttt{c}_1, ...\texttt{c}_\texttt{n}\rangle}$$

$$\frac{\begin{array}{c}\texttt{class D}\langle\texttt{c}_1, ...\texttt{c}_\texttt{m}\rangle\{\ \ ...\ \texttt{E}\langle\texttt{ap}_1, ...\texttt{ap}_\texttt{n}\ \rangle\ \texttt{f}\quad ...\quad \}\\ \texttt{C} \vdash \texttt{this}.\overline{\texttt{f}} : \texttt{D}\langle\texttt{cap}_1, ...\texttt{cap}_\texttt{m}\rangle\end{array}}{\texttt{C} \vdash \texttt{this}.\overline{\texttt{f}}.\texttt{f} : \texttt{E}\langle\ \texttt{ap}_1, ...\texttt{ap}_\texttt{n}[\texttt{cap}_1, ...\texttt{cap}_\texttt{m}/\texttt{c}_1, ...\texttt{c}_\texttt{m}][\texttt{this}.\overline{\texttt{f}}/\texttt{this}]\ \rangle}$$

Fig. 4. Path types for paths

The first rule says that in class `C` the path `this` has the type given by the cloning parameters of the class `C`. This gives, for example, that $\texttt{Union} \vdash \texttt{this} : \texttt{Union}\langle\texttt{c}\rangle$.

The second rule says that from class `C` the path $\texttt{this}.\overline{\texttt{f}}.\texttt{f}$ has the type of `f` as given in `f`'s declaration, but where $\texttt{c}_1, ...\texttt{c}_m$, the cloning parameters of the class declaring field `f`, are replaced by $\texttt{cap}_1, ...\texttt{cap}_m$, the actual cloning parameters for $\texttt{this}.\overline{\texttt{f}}$. Moreover, any occurrence of `this` in the type of `f` is replaced by $\texttt{this}.\overline{\texttt{f}}$. The first replacement is standard in the literature, and the second replacement is novel. This rule gives $\texttt{Union} \vdash \texttt{this.students} : \texttt{StudentList}\langle\texttt{this}\rangle$ and $\texttt{Union} \vdash \texttt{this.students.head} : \texttt{Node}\langle\texttt{this.students},\ \texttt{this}\rangle$.

Here are some more examples of the path type judgment:

$$
\begin{array}{c}
\texttt{Node} \vdash \texttt{this.next.next} : \texttt{Node}\langle\texttt{c1, c2}\rangle \\
\texttt{StudentList} \vdash \texttt{this.head.next.next} : \texttt{Node}\langle\texttt{this}, \texttt{c}\rangle \\
\texttt{StudentList} \vdash \texttt{this.head.student} : \texttt{Student}\langle\texttt{c}\rangle \\
\texttt{Union} \vdash \texttt{this.students.head} : \texttt{Node}\langle\texttt{this.students}, \texttt{this}\rangle \\
\texttt{Union} \vdash \texttt{this.students.head.student} : \texttt{Student}\langle\texttt{this}\rangle \\
\texttt{Union} \vdash \texttt{this.college} : \texttt{College}\langle\texttt{c}\rangle
\end{array}
$$

Note that the judgments only talk about paths, and makes no requirement about consistent views of the heap. So far, nothing precludes two paths $\texttt{p}_1$ and $\texttt{p}_2$, which are aliases at runtime, but which have different path types, eg $\texttt{C} \vdash \texttt{p}_1 : \texttt{D}\langle\texttt{cap}_1, ...\texttt{cap}_\texttt{n}\rangle$ and $\texttt{C} \vdash \texttt{p}_2 : \texttt{D}\langle\texttt{cap}'_1, ...\texttt{cap}'_\texttt{n}\rangle$ and $\texttt{cap}_1 \neq \texttt{cap}'_1$. However, as we see in section 5.3, we require consistent views on the heap in order to obtain completeness, *i.e.,* that all objects which should be cloned, are, indeed, cloned.

We can now characterise for a given class $\texttt{C}$ the set of paths to the objects in their cloning domain, $\mathit{ClnDom}(\texttt{C})$. This set consists of `this`, and all these paths $\texttt{p.f}$ whose path-type has as first argument another path $\texttt{p}'$ (remember that all paths start at `this`), *i.e.,* $\texttt{C} \vdash \texttt{p.f} : \texttt{D}\langle\texttt{p}', ...\rangle$, and which lie exclusively within the cloning domain of $\texttt{C}$, *i.e.,* $\texttt{p} \in \mathit{ClnDom}(\texttt{C})$.

$$
\mathit{ClnDom}(\texttt{C}) \;=\; \{\, \texttt{this} \,\} \cup \{\, \texttt{p.f} \mid \texttt{C} \vdash \texttt{p.f} : \texttt{D}\langle\texttt{p}', ...\rangle \;\; \text{for a class } \texttt{D}\text{, and a path } \texttt{p}', \text{ and where } \texttt{p} \in \mathit{ClnDom}(\texttt{C}) \,\}
$$

Thus, we obtain that

$$
\begin{array}{c}
\texttt{this.next.next} \notin \mathit{ClnDom}(\texttt{Node}) \\
\texttt{this.head.next.next} \in \mathit{ClnDom}(\texttt{StudentList}) \\
\texttt{this.college} \notin \mathit{ClnDom}(\texttt{Union}) \\
\texttt{this.students.head.next.next.student} \in \mathit{ClnDom}(\texttt{Union})
\end{array}
$$

The requirement that the complete path should lie within the cloning domain (i.e that $\texttt{p} \in \mathit{ClnDom}(\texttt{C})$) is relevant for the case where the cloning-owners are not dominators. For example, if the `College` object had a field `f` to some object which lied under the `StudentList` object, then `this.college.f` would not belong to $\mathit{ClnDom}(\texttt{Union})$.

4 Generating cloning methods

We have seen that the cloning behaviour of an object depends on the originator of the cloning action. For example, cloning a `Node` behaves differently when called as a result of cloning a `Union` or cloning a `StudentList`. This means that a single `clone()` method defined on each object cannot clone that object correctly in all contexts.

One approach would be to write a cloning method for every class that not only clones the object itself, but also clones every object within its cloning domain. Such a clone method would have to navigate through the appropriate paths and duplicate the appropriate objects. This approach is highly non-modular, since it exposes the internals of potentially every class.

Instead, we propose an overloaded parametric `clone(...)` method for each class, with additional Boolean actual parameters to describe whether or not its additional cloning domains should be cloned. For a class $\mathtt{C}\langle \mathtt{c}_1, ..\mathtt{c}_n\rangle$, we generate one overloaded parametric method, $\mathtt{clone}(\mathtt{Boolean}\ \mathtt{s}_1, ...\mathtt{Boolean}\ \mathtt{s}_\mathtt{n}, \mathtt{Map}\ \mathtt{m})$. The value of $\mathtt{s}_\mathtt{i}$ determines whether objects from the cloning domain $\mathtt{c}_i$ are to be cloned too. The original `clone` method can then be reimplemented to call the parametric cloning method, requesting only the object itself is cloned, by passing **false** to the additional cloning parameters.

The `Map m` formal parameter to the cloning method is to ensure that we clone the current object only if it has not already been cloned. We use the conventional solution of maintaining a table mapping original to cloned objects — something like a Smalltalk `IdentityDictionary` or Java 1.4's `IdentityHashMap` is ideal. If the object is not yet in the map, then the method will call the corresponding `clone` methods for all fields which need to be cloned, and will make aliases to all other fields — otherwise we simply use the clone stored in the map.

We show some examples of cloning methods in subsection 4.1 and then define the general case in subsection in 4.2.

4.1 A `clone` method for our example

For class Node, the basic `clone` method is called when the originator is the node itself, and simply delegates to the parametric cloning method. This method will be part of the interface of the cloning library.

```
Node clone( ){
    this.clone(false, false, new IdentityHashMap())
}
```

The parametric cloning method distinguishes whether the cloning domains of the clone parameters are to be cloned too, and accepts a `Map` to ensure each object is only cloned once. This method is internal to our system, and will not be visible outside the cloning library.

```
Node clone(Boolean s1, Boolean s2, Map m){
    Object n = m.get(this);
    if ( n != null) then {
       return (Node)n;
    } else {
       Node clone=new Node();
       m.put(this,clone);
       clone.next=  s1 ? this.next.clone(s1,s2,m) : this.next;
       clone.student= s2 ? this.student.clone(s2,m) : this.student;
       return clone;
     }
}
```

4.2 Code generation for clone

In Fig. 5 we show the generation of the basic cloning method for a class C. It calls the parametric cloning method on itself, where all the boolean parameters are set to false. This indicates that except for the cloning domain of the currnt object, no other cloning domains are "active".

In Fig. 6 we show the generation of the parametric cloning method for C.

As a first step, in the first four lines of the method, we need to determine whether the object is to be cloned. If the object is the outcome of a previous cloning action, *i.e.* is in the lookup table m, then nothing happens. Otherwise, a new object of that class is created, and entered into the table.

```
    C clone(){
        return this.clone($\mathtt{false}_1$, ...$\mathtt{false}_q$, newIdentitityHashMap());
    }
```

where q is the number of clone parameters of class C

Fig. 5. The function clone() for class C

```
    C clone(Boolean $\mathtt{s}_1$...Boolean $\mathtt{s}_q$, Map m){
        Object o = m.get(this)
        if o $\neq$ null then
            return (C)o;
        else{
            C o$'$ = new C();
            m.put(this, o$'$);
            o$'$.$\mathtt{f}_1$ = $\mathtt{s}_{1,1}$ ? this.$\mathtt{f}_1$.clone($\mathtt{s}_{1,1}$...$\mathtt{s}_{1,k1}$, m) : this.$\mathtt{f}_1$;
            ... = ...
            o$'$.$\mathtt{f}_n$ = $\mathtt{s}_{n,1}$ ? this.$\mathtt{f}_n$.clone($\mathtt{s}_{n,1}$...$\mathtt{s}_{n,kn}$, m) : this.$\mathtt{f}_n$;
            return o$'$;
        }
    }
```

where

$\{\mathtt{f}_1, ...\mathtt{f}_n\}$ are the fields defined in class C

and where, for all $i \in 1..n$:

$(fType(\mathtt{C}\langle \mathtt{s_1...s_n}\rangle, \mathtt{f_i}))[\mathtt{true/this}]{=}\mathtt{C_i}\langle \mathtt{s_{i,1}...s_{i,ki}}\rangle$

for some classes $\mathtt{C}_1$,.. $\mathtt{C}_n$.

Fig. 6. The function clone($\mathtt{s}_1 \ldots \mathtt{s}_n$, m) for class C

As a second step, we need to initialize the fields of the newly crated object, and determine which further objects have to be cloned. We assume that $\{\mathtt{f}_1, ...\mathtt{f}_n\}$ are all the fields defined in class C. Consider any field $\mathtt{f}_i$. We calculate the type of $\mathtt{f}_i$, and because anything that belongs to the cloning domain of the current object also belongs to the cloning domain of the owner

of the current object, we replace any occurrence of `this` by `true`. This gives that $\mathtt{C_i}\langle \mathtt{s_{i,1}}...\mathtt{s_{i,ki}}\rangle = \mathit{fType}(\mathtt{C}\langle \mathtt{s}_1...\mathtt{s}_n\rangle, \mathtt{f}_i)[\mathtt{true/this}]$ for some $\mathtt{C}_i$. We find the cloning parameter corresponding to $\mathtt{s}_{i,1}$: If $\mathtt{s}_{i,1}$ is `true`, then the object at $\mathtt{f}_i$ has to be cloned and the result has to be assigned to the field $\mathtt{f}_i$. If $\mathtt{s}_{i,1}$ is `false`, then the object at $\mathtt{f_i}$ should not be cloned, but the field $\mathtt{f_i}$ of the new object needs to alias its value. This is why we emit the the conditional expression:
$\mathtt{o}'.\mathtt{f}_i = \mathtt{s}_{i,1}\ ?\ \mathtt{this}.\mathtt{f}_i.\mathtt{clone}(\mathtt{s}_{i,1}...\mathtt{s}_{i,ki}, \mathtt{m})\ :\ \mathtt{this}.\mathtt{f_i}$.

5 Guarantees of cloning

We now discuss the properties of our cloning operation, namely

- termination,
- soundness, *i.e.,* the objects which are cloned are those which are reachable through paths in the cloning domain,
- completeness, *i.e.,* if the system also guarantees a consistent view of the heap then all objects from the cloning domain will be cloned.

We do not present an operational semantics, because our system works for any imperative object oriented language with the standard meaning for field read and write, conditional expressions and method calls. Because the system works on the basis of statically known paths and their types, the runtime representation of objects does not need to be enhanced with cloning domain information - we do not even need any ghost information for the purposes of the formal argument.

Nevertheless, we expect an operational semantics of form $H, \phi, \mathtt{expr} \rightsquigarrow H', v$, where H and H' are heaps, ϕ is a stack frame, `expr` is an expression, v stands for values, that values include `null` the booleans `true` and `false`, and addresses; and that addresses are represented as ι, ι' etc. Furthermore, we expect that $H(\iota, f)$ returns the value of field f of the object at ι, that H stores the class for each object, and that the stack frames ϕ map identifiers to values. Finally, we expect that expressions include field read, filed write, conditional expressions, and that these have the standard meaning.

The guarantees we describe in this section are applicable to richer languages, such that *e.g.,* support exceptions, or re-entrant method calls, or in fact, any further sequential control features. This is so, because our work makes guarantees about the effect of the code produced by our approach. This code only uses the language features listed above, and does not call any user-defined functions. The guarantees about the effects of the code remain valid regardless about the features used in the context that may be calling the generated `clone` methods, provided that this context is in the sequential setting.

In the following, we distinguish between the original call of the `clone()` method — without any arguments — and the subsequent recursive calls of the `clone(...)` methods which have at least one argument.

5.1 Termination

Lemma 1. *For all heaps H and variables $x \in dom(\phi)$ there exists an address ι and a heap H', such that $H, \phi, \mathtt{x.clone}() \rightsquigarrow H', \iota$.*

Proof Sketch We can show that during execution of `clone()` the values of fields in the objects in the original heap H do not change, and that all receivers of any of the recursive calls of the `clone(..)` methods were accessible from the originator object, ι, through a path p in the original heap H.

Therefore, the transitive calls of the `clone` method do not clone objects from the original heap H more than once, and do not clone any of the newly created objects. This gives a finite upper bound to the possible number of calls to the recursively called `clone(...)` methods. Since these methods do not contain any iteration, the fact that the number of possible recursive calls is finite, guarantees termination.

5.2 Soundness

With respect to the new heap, the cloning operation $H, \phi, \texttt{x.clone}() \rightsquigarrow H', \iota$, makes the following four guarantees:

1. It creates a new object for the object `x`,
2. H', the new heap, is homomorphic to H, the old heap,
3. In H' there is a self-contained part that corresponds to the old heap,
4. In H' there is a part that corresponds to the part in the old heap, which had to be copied according to the *ClnDom* function.

In order to express the last point formally, we define the set of objects that should be copied, as follows:
$ToCopy \ : \ Addr \times Heap \rightarrow Power(Addr)$
$ToCopy(\iota)_H = \{\ H(\iota.\overline{\texttt{f}}) = \iota' \mid \texttt{this}.\overline{\texttt{f}} \in ClnDom(\texttt{C}), \text{and } \texttt{C} \text{ the class of } \iota \text{ in } H\ \}$

We can now express soundness of the `clone` operation as follows:

Lemma 2. *If $H, \phi, \texttt{x.clone}() \rightsquigarrow H', \iota$, then there exists a mapping $\alpha : dom(H') \rightarrow dom(H)$ such that*

1. *ι is new in H, and $\alpha(\iota) = \phi(\texttt{x})$,*
2. *$\alpha(H'(\iota', \texttt{f})) = H(\alpha(\iota'), \texttt{f}))$ for all $\iota' \in dom(H')$ and fields `f`.*
3. *$\alpha|_{dom(H)}$ is the identity function, and $\alpha|_{dom(H')\backslash dom(H)}$ is injective.* [3]
4. *$\alpha(dom(H')\backslash dom(H)) \subseteq ToCopy(\phi(x))_H$.* [4]

Proof Sketch: We can show that during execution of `clone` for the newly created `Map` object `m`, that the range of `m` is the original heap H, and that any object being mapped comes from the new heap H', *i.e.,* for all objects ι', ι'', if $\texttt{m.get}(\iota'') = \iota'$ then $\iota'' \notin dom(H)$, and $\iota' \in dom(H)$. Furthermore, we can show that the lookup m.get(...) is injective, *i.e.,* for all objects ι', ι'', if $\texttt{m.get}(\iota') = \texttt{m.get}(\iota'')$ then $\iota'{=}\iota''$.

[3] **Notation:** For an f which is a mapping from A to B, and for $A' \subseteq A$, we use the term $f|_{A'})$ to the function f as restricted to the domain A'.

[4] **Notation:** For an f which is a mapping from A to B, and for $A' \subseteq A$, we use the term $f(A')$ to describe the image of f from A'.

Because of injectivity, we can construct the mapping $\alpha(\iota')$ as the inverse of `m` in the cases where ι' is the image of an original object in H (ie $\texttt{m.get}(\iota') \neq \texttt{null}$), and the identity otherwise.

Parts 1 and 3 follow from the construction of α and from the properties of the `Map` object mentioned above.

Part 2 follows from the body of the methods `clone(...)`.

For Part 4, we show that for any recursive application of any of the `clone(...)` methods, all the objects cloned so far are reachable from `x` through a path which lies completely within $\mathit{ClnDom}(\texttt{C})$, where `C` is the class of the object at `x`.

The latter follows from the following two facts

- All receivers of one of the recursive calls of the `clone(...)` methods are reachable from $\phi(\texttt{x})$ through a path that lies within $\mathit{ClnDom}(\texttt{C})$. This fact can be shown by induction on the recursive calls as follows:
  - The base case is easy.
  - For the inductive step, consider an object ι'' which executes a method `clone(...)` as part of the n-th recursive call, and which calls a method `clone(...)` on an object ι'''.
    Then, by construction of the `clone(...)` method, we know that the former call has the shape $\texttt{clone(true}, \texttt{b}_2, ...\texttt{b}_\texttt{q})$ and $q \geq 2$, that the latter call has the shape $\texttt{clone(b}'_1, ...\texttt{b}'_\texttt{r})$ and $r \geq 2$, and that $\texttt{b}'_1 = \texttt{true}$ such that $H(\iota'', f){=}\iota'''$, and where $\mathit{fType}(\texttt{C}' < \texttt{true}, \texttt{b}_2, ...\texttt{b}_\texttt{q} >, f) = \texttt{D} < \texttt{b}''_1, .., \texttt{b}''_\texttt{r} >$ for some $\texttt{b}''_1, .., \texttt{b}''_\texttt{r}$, s.t. $\texttt{b}''_1, .., \texttt{b}''_\texttt{r}[\texttt{true/this}] = \texttt{b}'_1, .., \texttt{b}'_\texttt{r}$, and where $\texttt{C}'$ is the class of the object ι''.
    From the inductive hypothesis, we obtain that there exists a path from $\phi(x)$ to ι'', which lies within $\mathit{ClnDom}(\texttt{C})$, *i.e.,* that $\texttt{C} \vdash p : \texttt{C}' < \texttt{p}', ... >$ for some path p'. Furthermore, the fact that $\texttt{b}'_1 = \texttt{true} = \texttt{b}''\texttt{1}[\texttt{true/this}]$ gives that $\mathit{fType}(\texttt{C}' < \texttt{c}_1, \texttt{c}_2, ...\texttt{c}_\texttt{q} >, f) = \texttt{D} < \texttt{ca}_1, .., \texttt{ca}_\texttt{r} >$ such that $\texttt{ca}_1 = \texttt{this}$, or that there exists an m such that $\texttt{ca}_1 = \texttt{c}_\texttt{m}$ and $\texttt{b}_\texttt{m} = \texttt{true}$. In the first case, by definition of the judgment, we obtain that $\texttt{C} \vdash p.f : \texttt{D} < \texttt{p}', ... >$, while in the second case we obtain $\texttt{C} \vdash p.f : \texttt{D} < \texttt{p}, ... >$. In either case, the path $p.f$ lies within $\mathit{ClnDom}(\texttt{C})$.
- Only the objects that received one of the recursive calls of `clone(...)` may be have been cloned.

5.3 Completeness

Note, that Lemma 2 only guarantees that all cloned objects come from $\mathit{ToCopy}(\iota)_H$, and does not guarantee that *all* objects from $\mathit{ToCopy}(\iota)_H$ have been cloned. However, we can obtain this stronger property, if the system guarantees consistent views of the heap.

We first define consistent views of the heap to say that different paths leading to the same object must have the same path type.

Definition 1. *We say that a type system gives* consistent views of the heap*, if for all classes* $\mathtt{C}$*, paths* $\mathtt{p}_1$ *and* $\mathtt{p}_2$*, and for all heaps* H *and stack frames* ϕ *which may arise through execution of well-typed expressions, if* $\mathtt{p}_1$ *and* $\mathtt{p}_2$ *are aliases in* H *and* ϕ*, and* $\phi(\mathtt{this})$ *is an object of class* $\mathtt{C}$*, then*

$$\Gamma \vdash \mathtt{p}_1 : T_1 \textit{ and } \Gamma \vdash \mathtt{p}_2 : T_2 \textit{ imply } T_1 = T_2$$

For example, if we had a system with consistent views of the heap, and if in our Fig. 1 the object of class `StudentList` contained a field called `last` which pointed to the same object as `this.head.next.next`, then, for typing environments $\varGamma_1$ and $\varGamma_2$, such that $\varGamma_1(\mathtt{this}) = \mathtt{StudentList}$ and $\varGamma_2(\mathtt{this}) = \mathtt{Union}$, we would also have that $\varGamma_1 \vdash \mathtt{this.last} : \mathtt{Node} < \mathtt{this}, \mathtt{c} >$ — this follows because of $\varGamma_1 \vdash \mathtt{this.head.next.next} : \mathtt{Node} < \mathtt{this}, \mathtt{c} >$. Furthermore, we would also have that $\varGamma_2 \vdash \mathtt{this.students.last.student} : \mathtt{Student} < \mathtt{this} >$ — this follows because of $\varGamma_2 \vdash \mathtt{this.students.head.next.next.student} : \mathtt{Student} < \mathtt{this} >$.

The requirement of consistent views guarantees that the underlying type system restricts field assignment so that paths pointing to separate cloning domains cannot be aliases. Such requirements are satisfied by any sound ownership type systems, *e.g.,* as early as the system in [8].

Consistency in the views of the heap implies consistency in the path types:

Lemma 3. *If a type system gives* consistent views of the heap*, then, for all classes* $\mathtt{C}$*, paths* $\mathtt{p}_1$ *and* $\mathtt{p}_2$*, and for all heaps* H *and stack frames* ϕ *which may arise through execution of well-typed expressions, if* $\mathtt{p}_1$ *and* $\mathtt{p}_2$ *are aliases in* H *and* ϕ*, and* $\phi(\mathtt{this})$ *is an object of class* $\mathtt{C}$*, then*

$$\mathtt{C} \vdash \mathtt{p}_1 : \mathtt{PT}_1 \textit{ and } \mathtt{C} \vdash \mathtt{p}_2 : \mathtt{PT}_2 \textit{ imply } \mathtt{PT}_1 = \mathtt{PT}_2$$

Continuing our example from above, cosistent views of the heap would give us that $\mathtt{Union} \vdash \mathtt{this.students.last} : \mathtt{Node} < \mathtt{this.students}, \mathtt{this} >$, because $\mathtt{Union} \vdash \mathtt{this.students.head.next.next} : \mathtt{Node} < \mathtt{this.students}, \mathtt{this} >$.

Lemma 4. *Assume that the type system gives consistent views of the heap. If* $H, \phi, \mathtt{x.clone}() \rightsquigarrow H', \iota$*, then the mapping* $\alpha : dom(H') \to dom(H)$ *described in lemma 2 has the property that* $\alpha(dom(H') \backslash dom(H)) = \mathit{ToCopy}(\phi(x))_H$

Proof Sketch: We show by induction on n, that when considering the breadth first view of recursive applications of the `clone(...)` methods at depth n, then all objects which are reachable from `x` through a path of length at most n which lies within $\mathit{ClnDom}(\mathtt{C}')$, have been cloned – with $\mathtt{C}'$ the class of the object at `x`.

In order to prove the above, we strengthen the assertion to say that for all n:

1. For all paths $\mathtt{p}$ and $\mathtt{p}'$, where $\mathtt{p} \in \mathit{ClmDom}(\mathtt{C}')$, and $\mathtt{p}$ has length at most n such that $\mathtt{C}' \vdash \mathtt{p} : \mathtt{D} < \mathtt{p}_1, \mathtt{cap}_2 ... \mathtt{cap}_\mathtt{m} >$, if $H(\phi(\mathtt{x}), \mathtt{p}) = \iota$, then within the breadth first view of recursive applications of the `clone(...)` methods at depth n there is a call of $\mathtt{clone}(\mathtt{true}, \mathtt{b}_2, ..\mathtt{b}_\mathtt{m})$ on ι, where $\mathtt{b}_\mathtt{i}$=`true` if $\mathtt{cap}_\mathtt{i}$ has the form $\mathtt{this}.\overline{f}$, and `false` otherwise.

2. For any calls of $\texttt{clone}(\texttt{true}, \texttt{b}_2, ..\texttt{b}_\texttt{m})$ on an object ι within the breadth first view of recursive applications of the `clone(...)` methods at depth n, there is a path `p` from $\phi(\texttt{x})$ to ι, such that $\texttt{C}' \vdash \texttt{p} : \texttt{D} < \texttt{p}_1, \texttt{cap}_2 ... \texttt{cap}_\texttt{m} >$, and $\texttt{b}_\texttt{i}$=`true` if $\texttt{cap}_\texttt{i}$ has the form $\texttt{this}.\overline{f}$, and `false` otherwise.

We now sketch the proof of the proposition from above:

- The base case is trivial.
- For the inductive step, we look at the two assertions separately:
  1. Consider any object ι' which is reachable from $\phi(\texttt{x})$ through a path `p.f` of length $n+1$ such that $\texttt{p.f} \in \mathit{ClmDom}(\texttt{C}')$, and $\texttt{C}' \vdash \texttt{p.f} : \texttt{D} < \texttt{p}', \texttt{cap}_2 ... \texttt{cap}_\texttt{m} >$ for some further path $\texttt{p}'$. Therefore, there exists a further object ι'', reachable from $\phi(\texttt{x})$ through `p`, such that $\texttt{C}' \vdash \texttt{p} : \texttt{E} < \texttt{cap}'_1, \texttt{cap}'_2 ... \texttt{cap}'_\texttt{r} >$, and that $H(\iota'', f) = \iota'$, and $\texttt{cap}'_1$ has the form of a path.
     By application of the inductive hypothesis part 1., we know that when considering the breadth first view of the recursive applications of the `clone(...)` methods at depth n, the object ι'' will have received the recursive method call $\texttt{clone}(\texttt{b}_1, \texttt{b}_2, ...\texttt{b}_\texttt{r})$ such that $\texttt{b}_\texttt{i}$ is `true`, if $\texttt{cap}'_\texttt{i}$ has itself the form of a path, and otherwise $\texttt{b}_\texttt{i}$ is `false`.
     We now inspect the first lines of the code of the `clone(...)` method. At the point of execution of the method call $\texttt{clone}(\texttt{b}_1, \texttt{b}_2, ...\texttt{b}_\texttt{r})$ on ι' it is possible that the object ι' already had been entered in the map table `m`. By inspection of the `clone(...)` methods, this implies that ι' had already received the method call $\texttt{clone}(\texttt{b}'_1, \texttt{b}'_2, ...\texttt{b}'_\texttt{r})$ for some $\texttt{b}'_1, \texttt{b}'_2, ...\texttt{b}'_\texttt{r}$. By application of the inductive hypothesis part 2, we know that there exists a path `p''` from $\phi(\texttt{x})$ to ι'', such that $\underline{\texttt{C}}' \vdash \texttt{p}'' : \texttt{E} < \texttt{p}''_1, \texttt{cap}''_2 ... \texttt{cap}''_\texttt{m} >$, and $\texttt{b}'_\texttt{i}$=`true` if $\texttt{cap}''_\texttt{i}$ has the form $\texttt{this}.\overline{f}$, and `false` otherwise.
     By application of the consistent view property and lemma 3, we obtain that $\texttt{b}'_\texttt{i} = \texttt{b}_\texttt{i}$. We now inspect the remaining lines of the code of the `clone(...)` method. These will give that the object ι' will receive the method call $\texttt{clone}(\texttt{true}, \texttt{b}''_2, ...\texttt{b}''_\texttt{m})$ where $\texttt{b}''_\texttt{i}$ is `true` if if $\texttt{cap}_\texttt{i}$ has itself the form of a path, and otherwise $\texttt{b}''_\texttt{i}$ is `false`.
  2. This assertion is proven by induction on the depth of method calls of `clone(...)`, and analysis of the body of the method.

6 Related Work and Conclusions

In this paper we have sketched a way of *specifying* cloning policies by annotating the fields of objects, and then *deriving* code for cloning which satisfies the cloning policy. Our lemma 2 guarantees that the derived cloning code produces a faithful copy of the cloned object, and that cloned objects are within the domain expressed by the types, while lemma 4 guarantees that if the types give consistent views of the heap, then *all* objects within the domain expressed by the types will be cloned. We have not yet dealt with subclasses, but we do not expect this to be challenging, because the cloning method will be bound dynamically to the class of the receiver. We plan to develop a detailed formal system and its proofs in further work.

Our work was inspired by Jensen et al.'s secure cloning policies [14]. Unlike the secure cloning policies, our work *generates* cloning methods, rather than simply checking them, and is based on ownership types, rather than a series of ad-hoc annotations. Compared with most ownership systems [5, 10, 9] our system is *descriptive* [3] in that it captures object relationships, rather than *prescriptively* ensuring invariants over the heap. We do not require or enforce heap properties (*e.g.,* clone/owners as dominators or owners as modifiers), nor even that the paths support a consistent view of the heap. In fact, owners as dominators could be too strong a requirement for cloning purposes, as it would unnecessarily forbid many programs, e.g. cloning a list with iterators. A consistent view of the heap may be enforced by the type system in the usual way [5]. We want to investigate in how far such a consistent view is the usual idiom, or whether there are situations where consistency is too restrictive from the practical perspective [15].

The relationship between ownership and cloning was first identified in a *dynamic* ownership setting, where every object knows its owner at runtime [17]. These ownership-based clones are called ”sheep clones”, as they are intermediate between shallow (single object) clones and deep (fully recursive) clones. Cloning, and programming models and languages based on cloning, have of course been studied more generally [4, 18, 20, 23]. Ownership type techniques have also been applied to other problems closely related to cloning, in particular object initialisation [11, 22], as cloning creates and initialises a new object — and ownership transfer [7], as transfer can be modelled by cloning an object then deleting the original. Cloning objects is also similar to comparing objects [12, 13] and generating hash codes [21].

As well as completing the formal model, in further work we plan to investigate whether it is possible to derive more efficient versions of the `clone` function, so as to avoid the use of the `Map` object when possible. We also want to support generics, and existential (unknown) clone parameters.

Acknowledgments

We thank the FTfJP'11 reviewers for their constructive criticism on an earlier version of this paper, the FoVEOOS'11 reviewers for their feedback, and Paley Li for general discussions.

References

1. C. Andreae, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao. Scoped Types and Aspects for Real-Time Java. In European Conference on Object-Oriented Programming (ECOOP), 2006.
2. N. Cameron. *Existential Types for Variance — Java Wildcards and Ownership Types.* PhD thesis, Imperial College London, 2009.

3. N. Cameron, S. Drossopoulou, J. Noble, and M. Smith. Multiple Ownership. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 441–460. ACM, 2007.
4. K. Campbell, J. McWhir, W. Ritchie, and I. Wilmut. Sheep cloned by nuclear transfer from a cultured cell line. *Nature*, 380:64–66, Mar. 1996.
5. D. Clarke. *Object Ownership and Containment.* PhD thesis, University of New South Wales, 2001.
6. D. Clarke and S. Drossopoulou. Ownership, Encapsulation and the Disjointness of Types and Effects. In *OOPSLA*, pages 292–310, Seattle, Washington, USA, November 2002. ACM Press.
7. D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *ECOOP*, pages 176–200, 2003.
8. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, volume 33(10), pages 48–64. ACM Press, 1998.
9. W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In *ECOOP*, volume 4609 of *LNCS*, pages 28–53. Springer, 2007.
10. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *JOT*, 4(8):5–32, October 2005.
11. M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *OOPSLA*, pages 337–350. ACM Press, 2007.
12. P. Grogono and P. Chalin. Copying, sharing, and aliasing. In *Proceedings of the Colloquium on Object Orientation in Data bases and Software Engineering (COODBSE'94)*, Montreal, Quebec, May 1994.
13. P. Grogono and M. Sakkinen. Copying and comparing: Problems and solutions. In *ECOOP*, pages 226–250, 2000.
14. T. Jensen, F. Kirchner, and D. Pichardie. Secure the Clones – Static Enforcement of Policies for Secure Object Copying. In *ESOP*, 2011.
15. P. Li. Sheep cloning for ownership. In *OOPSLA Doctoral Consortium*, 2011. To Appear.
16. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
17. J. Noble, D. Clarke, and J. Potter. Object ownership for dynamic alias protection. In *TOOLS Pacific*, 1999.
18. J. Noble and B. Foote. Attack of the clones. In *Proceedings of the 2002 Australasian Conference on Pattern Languages of Programs*, volume 13 of *CRPIT*, pages 99–144, 2002.
19. J. Noble, J. Potter, and J. Vitek. Flexible alias protection. In *ECOOP*, Brussels, Belgium, July 1998.
20. J. Noble, A. Taivalsaari, and I. Moore, editors. *Prototype-Based Programming: Conecepts, Languages, Applications.* Springer-Verlag, 1997.
21. D. Rayside, Z. Benjamin, R. Singh, J. P. Near, A. Milicevic, and D. Jackson. Equality and hashing for (almost) free: Generating implementations from abstraction functions. In *ICSE*, pages 342–352, 2009.
22. A. J. Summers and P. Müller. Freedom before commitment : A lightweight type system for object initialisation. In *OOPSLA*, 2011. To Appear.
23. D. Ungar and R. B. Smith. SELF: the Power of Simplicity. *LISP and Symbolic Computation*, 4(3), June 1991.
24. J. Vitek and J. Bokowski. Confined types for java. *Software Partice and Experience*, 2001.

Towards the verification of efficient BDD algorithms

Mathieu Giorgino and Martin Strecker

IRIT, Université de Toulouse

Abstract. This paper is an extended case study using a high-level approach to the verification of graph transformation algorithms: To represent sharing, graphs are considered as trees with additional pointers, and algorithms manipulating them are essentially primitive recursive traversals written in a monadic style. With this, we achieve almost trivial termination arguments and can use inductive reasoning principles for showing the correctness of the algorithms. We illustrate the approach with the verification of a BDD package which is modular in that it can be instantiated with different implementations of association tables for node lookup. We have also implemented a garbage collector for freeing association tables from unused entries. Even without low-level optimizations, the resulting implementation is reasonably efficient.

Keywords: Verification of imperative algorithms, Pointer algorithms, Modular Program Development, Binary Decision Diagram

1 Introduction

There is now a large range of verification tools for imperative and object-oriented (OO) languages. Most of them have in common that they operate on source code of a particular programming language like C or Java, annotated with pre- / postconditions and invariants. This combination of code and properties is then fed to a verification condition generator which extracts proof obligations that can be discharged by provers offering various degrees of automation (see below for a more detailed discussion).

This approach has an undeniable success when it comes to showing that a program is well-behaved (no null-pointer accesses, index ranges within bounds, deadlock-freedom of concurrent programs etc.). Program verification in this sense essentially amounts to showing the absence of undesirable situations with the aid of a property language that is considerably more expressive than a traditional type system, but nevertheless has a restricted set of syntagmas that cannot be user-extended.

These limitations turn out to be a hindrance when one has to build up a larger “background theory” capable of expressing deeper semantic properties of the data structures manipulated by the program (such as the notions of interpretation and validity of a formula used in this paper). Even worse, high-level mathematical notions (such as “sets” and “trees”) are often not directly available in the specification language. Even if they are, recovering an algebraic data type from a pointer structure in the heap is not straightforward: one has to ensure, for example, that a structure encoding a list is indeed an instance of a data type and not cyclic.

In this paper, we explore the opposite direction: we start from high-level data structures based on inductive data types, which allows for an easy definition of algorithms with the aid of primitive recursion and for reasoning principles based on structural induction. References are added explicitly to these data structures, which makes it possible to express sharing of subtrees as well as a simple notion of reference equality. The notion of state is manipulated with a state monad (see Section 2), thus allowing for a restricted form of object manipulation (in particular object creation and disposal).

We illustrate our approach with the development of a Binary Decision Diagram (BDD) package. After recalling the basic notions and the semantics of BDDs in Section 3, we describe a first, non-optimized version of the essential algorithms in Section 4 and the implementation of lookup tables in Section 6.

As compared to our preliminary report [15] on this subject, the current work is based on a different memory model and includes a garbage collector and memoization (Section 5), which lead to a substantial speed-up.

As formal framework, we use the Isabelle proof assistant [23] and its extension Imperative_HOL [8], together with its Isabelle-to-Scala code extractor. Our algorithms are therefore executable in Scala and, as witnessed by the performance evaluation of Section 7, within the realm of state-of-the-art BDD packages.

A further gain in efficiency might be achieved by mapping our still rather coarse-grained memory model to a fine-grained memory model, which would allow us to to introduce bit-level optimizations. Such a multi-level refinement has been reported for the verification of a microkernel [10] which achieves a similar performance as non-verified hand-crafted implementations. Even though such a refinement is compatible with our approach, we have refrained from it here. It would lead to a considerable increase in complexity, because it requires a simulation proof between the two levels of abstraction.

We conclude in Section 8 with a discussion of the general limitations and current weaknesses of our approach and a wish-list for future enhancements.

The formal development is available on the authors' home pages [14].

Related work – program verification: There are roughly two broad classes of program verifiers - those aiming at a mostly automatic verification, as Spec# [1], VCC [11], Frama-C [13] or Why3 [3], or at mostly interactive proofs, such as the ones based on Dynamic Logic [2,19] or codings of programming languages and their associated Hoare logics in proof assistants [12,28]. The borderline is not clear-cut, since some of the "automatic" tools can also be interfaced with interactive proof assistants such as Coq and Isabelle, as in [4].

The work that comes closest to ours is the extension of Isabelle with OO features, in an encoding [5,6]. It is at the same time more complete and considerably more complex, since it has the ambition to simulate genuine OO capabilities such as late binding, which requires, among others, the management of dynamic type tags of objects. Our approach remains confined to what can be done within a conventional polymorphic functional type system. Our aim is not to be able to verify arbitrary programs in languages such as Java or Scala, but to export programs written and verified in a functional style with stateful features to a language such as Scala. We thus hope to reduce the proof burden, while still obtaining relatively efficient target code.

Related work – verification of BDDs: Binary Decision Diagrams (BDDs) [7] are a compact format for representing Boolean formulas, making extensive use of sharing of subtrees and thus achieving a canonical representation of formulas. A verified BDD package might become useful for the formal verification of decision procedures [9,26,27]. In this spirit, a recent SAT solver verification effort [22] comes to the conclusion that mutable references would lead to huge performance improvements.

Even without such an application in mind, BDDs have become a favorite case study for the verification of pointer programs. As mentioned above, all the approaches we are aware of use a low-level representation of BDDs as linked pointer structures. [20] introduces the idea of representing the state space in monadic style, but ensuring the termination of the functions poses a problem because termination and well-formedness of the state space are closely intertwined.

For a verification framework embedding a Hoare logic for imperative programs in the Isabelle proof assistant, the paper [25] describes the verification of BDD algorithms written in a C-like language. As in our case, it is possible to take semantic properties of BDDs into account, but the proof of correctness has a considerable complexity.

By a tricky encoding of lookup tables by injective pairing functions, the PVS formalization in [30] can avoid the use of the notion of "state" altogether. On the downside, the encoding creates huge integers even for a small number of BDD nodes, so that the approach might not scale to larger examples.

The most comprehensive verification [29] (apart from ours) describes a verification in the Coq proof assistant, including some optimizations and a garbage collector. The state space is explicitly represented and manipulated by a functional program, and also the OCaml code extracted from Coq is functional. This seems to account for the lower performance (slower execution and faster exhaustion of memory) as compared to genuine imperative code.

2 Memory and Object Models

We first present a shallow embedding of an OO management of references in Isabelle. As a basis we use the Imperative_HOL theory belonging to the Isabelle library. This theory defines a state-exception monad with syntax facilities like `do`-notation. Details can be found in [8].

2.1 Imperative_HOL

Imperative_HOL first defines a polymorphic heap (of type *heap*) on which a state-exception monad (of type *'a Heap*) is defined. The monadic constructors *return* and *raise* allow to encapsulate a value or an exception in a monad and the monadic combinator *bind* (written $m \trianglerighteq n$) allows to combine operations on monads.

Then a reference type *'a ref* is defined and several primitives are provided to manipulate directly the heap through references:

- *Ref.present* r: Tests allocation of reference r.
- *Ref.get* h r: Obtains the value of reference r in a heap h
- *Ref.set* h r a: Replaces the value referenced by r in heap h by v
- *Ref.alloc* a: Returns a newly allocated reference to a value a value
- *Ref.noteq* r r' ($r =!= r'$): Tests equality between references r and r' (possibly with distinct types)

These are intended for use in logical statements and proofs and no code equations (used for extraction to programming languages like Caml or Scala) are available for them.

To access the state in programs, one has to use the state-exception monad for which similar primitives using the monad's state are defined:

- *Ref.ref* a: Returns a newly allocated reference set to the value a
- *Ref.lookup* r ($!r$): Obtains the value of reference r
- *Ref.update* r a ($r := a$): Replaces the value of reference r by a

Imperative_HOL finally provides the do-notation allowing to write monadic expressions as usual imperative statements. The do-notation is enclosed in *do*{...}. Monadic expression composition is done with *do*{ $v \leftarrow m$; m' v } translated to $m \trianglerighteq m'$. The binding part $v \leftarrow$ can be omitted in which case *do*{m; m'} is syntactically transformed to $m \trianglerighteq (\lambda\text{-}.\ m')$ leading to the discarding of the returned value of the first monadic expression.

2.2 Accessors

Then Imperative_HOL provides a means to manipulate references in a heap with the help of an imperative language in Isabelle/HOL. However, the direct manipulation of references prevents an efficient code generation in an OO language. In order to abstract from references and to get closer to an OO development, we have to see a reference to a record as a handle to an object, without being able to retrieve the record itself. To do this, we define accessors (of type $'a \rhd 'b$) as a means to describe an abstract attribute in an object.

We use them as arguments of the primitives *RAcc.get*, *set* and *map* applied to references and heap, and their monadic counterparts *lookup* ($r\cdot ac$), *update* ($r\cdot ac$.= v) and *rmap* operating on *'a Heap*.

For example, with the definition of accessors *\$fst* and *\$snd* for the first and second components of pairs, the term $do\{\ a \leftarrow p{\cdot}\$fst;\ p{\cdot}\$snd\ .{=}\ a\ \}$ defines a monadic operation replacing the second component of the pair referenced by p by its first component. On the logical level, the term *RAcc.set \$snd p* (*RAcc.get h \$fst p*) *h* would describe the result of this operation on the heap *h*.

2.3 Objects and Classes

Additionally, we use two constructions provided by Isabelle/HOL to get closer to OO developments. The first one allows to define a hierarchy of data, while the second allows to define methods hierarchically associated to the data of the first one.

Data Hierarchical definition of data (with sub-typing) is provided by extensible records (**record**). A record *runit* containing only one field named *nothing* of type *unit* – as records cannot be empty – is used as a top of a record hierarchy, in the same way as `Object` in Java or `Any` in Scala are the top of the class hierarchy. However, in contrast to the implicit object sub-typing, the record definition introduces an explicit type parameterized by its extension type as a *'a runit-scheme*.

record *runit* = *nothing* :: *unit*

Then we define a type synonym *'a any* that we will often use with a place-holder (-) as - *any*, to let the type inference set it as general as possible.

type-synonym *'a any* = *'a runit-scheme*

Methods Locales (**locale**) were originally created to parameterize theories for several interpretations but they also allow to parameterize a set of definitions by constants (**fixes**) and assumptions (**assumes**). Then we can use them to define functions in the context of a reference called *this* in the same way as for OO languages. Outside of a locale, the functions defined in this locale will take an additional argument being a reference to the record.

locale *object* = **fixes** *this* :: *'a::heap ref*

Locales can also be used as an equivalent of interfaces or abstract classes. They can be built upon each other with multiple inheritance (+) for which assumptions (including types) can be strengthen (**for**). Finally they can be instantiated by several implementations.

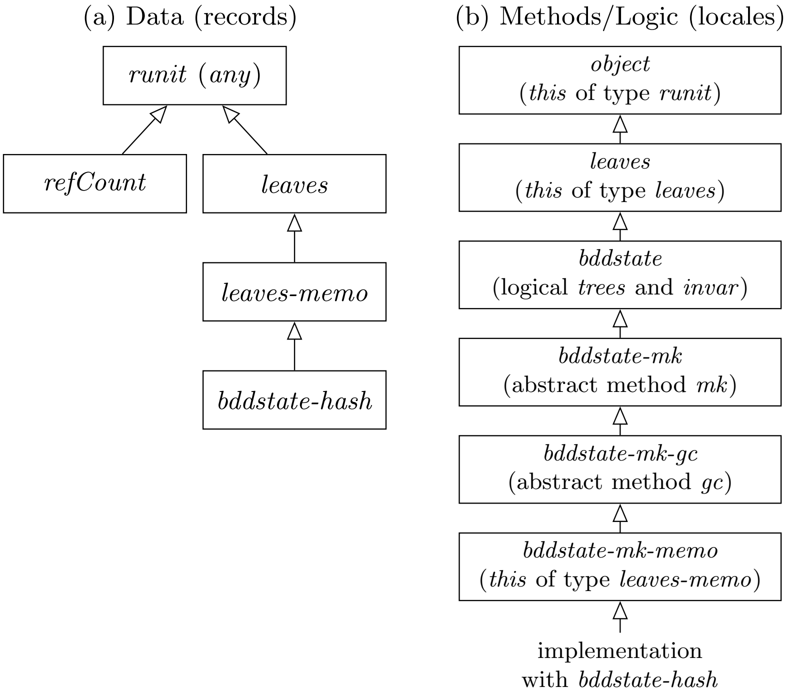
In this development, objects and classes are used at two levels:

- for the state of the BDD factory containing the two *True* and *False* leaves and the association tables for maximal sharing and memoization. This state and its reference is unique in the context of the algorithms and provided by the locale *object* as a *this* constant parameter.

– for the nodes containing a reference to a mutable extension *runit* and then *refCount* of it which is used to store the reference counter for the garbage collection.

Figures 1a and 1b present the hierarchies of records and locales used in this development. We also take advantage of locales to specify the logical functions used only in proofs (locale *bddstate*) and the abstract methods (locales *bddstate-mk* and *bddstate-mk-gc*).

Fig. 1: Hierarchies of data and methods


(a) Data (records)

(b) Methods/Logic (locales)

runit (any)
refCount
leaves
leaves-memo
bddstate-hash
object (this of type runit)
leaves (this of type leaves)
bddstate (logical trees and invar)
bddstate-mk (abstract method mk)
bddstate-mk-gc (abstract method gc)
bddstate-mk-memo (this of type leaves-memo)
implementation with bddstate-hash

3 Binary Decision Diagrams

BDDs are used to represent and manipulate efficiently Boolean expressions. We will use them as starting point of our algorithms, by defining a function constructing BDDs from their representation of type $('v,\ bool)\ expr$ in which $'v$ is the type of variable names. The definition of expressions is rather standard:

types $'a\ binop = 'a \Rightarrow 'a \Rightarrow 'a$

datatype $('v,\ 'a)\ expr =$
  $Var\ 'v$

```
| Const 'a
| BExpr 'a binop (('v, 'a) expr) (('v, 'a) expr)
```

and their interpretation is done by *interp-expr* taking as extra argument the variables instantiations represented as a function from variables to values:

```
primrec interp-expr :: ('v, 'a) expr ⇒ ('v ⇒ 'a) ⇒ 'a where
  interp-expr (Var v) vs = vs v
| interp-expr (Const a) vs = a
| interp-expr (BExpr bop e1 e2) vs =
  bop (interp-expr e1 vs) (interp-expr e2 vs)
```

We now define BDDs as binary trees in which references to an extensible record will be added.

```
datatype ('a, 'b) tree =
  Leaf 'a
| Node 'b (('a, 'b) tree) (('a, 'b) tree)

type-synonym
  ('a, 'b, 'c) rtree = ('a × 'c any ref, 'b × 'c any ref) tree
```

To define their constructors, we need a value of type *'a Object.any* which will be used for initialization. Then we use a locale parameterized by a *default-any* constant used in the definition of *newLeaf* (and *newNode* in the same way)

```
locale default-any =
  fixes default-any :: 'c::heap any
begin

definition newLeaf :: bool ⇒ (bool, 'v::heap, -) rtree Heap where
  newLeaf b = do{
    r ← ref default-any;
    return (Leaf (b, r))
  }

end
```

In this way, as long as subtrees having identic references are the same, we can represent sharing. To ensure this property giving meaning to references, we use the predicate *ref-unique ts*:

```
definition ref-unique :: ('a, 'v, -) rtree set ⇒ bool where
  ref-unique ts ≡
  ∀ t1 t2. t1 ∈ ts ⟶ t2 ∈ ts ⟶ ref-equal t1 t2 ⟷ struct-equal t1 t2
```

in which *ref-equal* means that two trees have the same reference attribute, and *struct-equal* is structural equality neglecting references, thus corresponding to the typical notion of equality of data in functional languages.

While the left-to-right implication of this equivalence is the required property (two nodes having the same reference are the same), the other implication ensures maximal sharing (same subtrees are shared, *i. e.* have the same reference).

Let us illustrate the concept of subtree sharing by an example. A non-shared BDD (thus, in fact, just a decision tree) representing the formula $(x \wedge y) \vee z$ is given by the following tree (omitting references):

```
Node x
     (Node z (Leaf false) (Leaf true)),
     (Node y (Node z (Leaf false) (Leaf true))
             (Leaf true))
```

There is a common subtree `(Node z (Leaf false) (Leaf true))` which we would like to share. We therefore adorn the tree nodes with references, using the same reference for structurally equal trees, for example:

```
Node (x, 1)
     (Node (z, 3) (Leaf (false, 4)) (Leaf (true, 5))),
     (Node (y, 2) (Node (z, 3) (Leaf (false, 4)) (Leaf (true, 5))),
                (Leaf (true, 5)))
```

The process of sharing is illustrated in Figure 2.

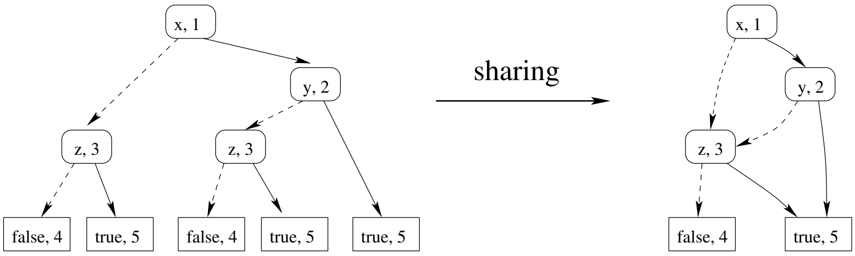


Fig. 2: Sharing nodes in a tree

Each node contains a variable index whose type is any type equipped with a linear order (as indicated by Isabelle's type class annotation) and each leaf contains a value of any type instantiated later in the development (for interpretations) to Booleans. To allow writing simple and generic algorithms (*i. e.* avoid particular cases), leaves and nodes should be usable in the same way. For example, we define a linear order on levels of trees by having level of leaves always greater than levels of nodes and using variable indices to compare nodes.

BDDs can be interpreted by giving values to variables which is what the *interp* function does:

fun *interp* :: ('*a*, '*v*, -) *rtree* ⇒ ('*v* ⇒ *bool*) ⇒ '*a* **where**
  interp (*Leaf* (*b*,*r*)) *vs* = *b*
| *interp* (*Node* (*v*,*r*) *l h*) *vs* = (*if vs v then interp h vs else interp l vs*)

With this definition, and without any other property, BDDs would be rather hard to manipulate. On the one hand, same variable indices could appear several times on paths from root to leaves. On the other hand, variables would not be in the same order, making comparison of BDDs harder. Moreover, a lot of space would be wasted. To circumvent this problem, one often imposes a strict order on variables, the resulting BDDs being called ordered (OBDDs):

```
fun tree-vars :: ('a, 'b, -) rtree ⇒ 'b set where
  tree-vars (Node (v,r) l h) = insert v (tree-vars l ∪ tree-vars h)
| tree-vars (Leaf b) = {}
```

```
fun ordered :: ('a, 'v::ord, -) rtree ⇒ bool where
  ordered (Leaf b) = True
| ordered (Node (i, r) l h) =
  ((∀ j ∈ (tree-vars l ∪ tree-vars h). i < j) ∧ ordered l ∧ ordered h)
```

An additional important property is to avoid redundant tests, which occurs when the two children of a node have the same interpretation. All the nodes satisfying this property can be removed. In this case, the OBDD is said to be reduced (ROBDD).

```
fun reduced :: ('a, 'v, -) rtree ⇒ bool where
  reduced (Node vr l h) = ((interp l ≠ interp h) ∧ reduced l ∧ reduced h)
| reduced (Leaf -) = True
```

This property uses a high-level definition (*interp*), but it can be deduced (*c.f.* lemma below) from the three low-level properties *ref-unique*, *ordered* (already seen) and *non-redundant*:

```
fun non-redundant :: ('a, 'v, -) rtree ⇒ bool where
non-redundant(Node vr l h)=((¬ref-equal l h) ∧ non-redundant l ∧ non-redundant h)
|non-redundant(Leaf -) = True
```

We then merge these properties in two definitions *robdd* (high-level) and *robdd-refs* (low-level):

```
definition robdd t == (ordered t ∧ reduced t)
```

```
definition robdd-refs t == (ordered t ∧ non-redundant t ∧ ref-unique (treeset t))
```

From these definitions, we can then show that ROBDDs are a canonical representation of Boolean expressions, *i. e.* that two equivalent ROBDDs are structurally equal at high (*robdd*) and low (*robdd-refs*) level:

```
robdd t1 ⟹ robdd t2 ⟹ interp t1 = interp t2 ⟹ struct-equal t1 t2
```

```
robdd-refs t1 ⟹
robdd-refs t2 ⟹ interp t1 = interp t2 ⟹ struct-equal t1 t2
```

Moreover we can show that the high-level property can be obtained from the low-level one:

```
ordered t ⟹ ref-unique (treeset t) ⟹ non-redundant t ⟹ reduced t
```

And then that *robdd* and *robdd-refs* are equal as soon as all the subtrees are maximally shared:

ref-unique (*treeset t*) $\Longrightarrow$ *robdd-refs t* = *robdd t*

4 Constructing BDDs

The simplest BDDs are the leaves corresponding to the *True* and *False* values. These ones have to be unique in order to permit sharing of nodes. We put them in the BDD factory whose data is this record:

record ('*v*, '*c*) *leaves* = *runit* +
  leafTrue :: (*bool*, '*v*, '*c*) *rtree*
  leafFalse :: (*bool*, '*v*, '*c*) *rtree*

and for which we define the accessors \$*leafTrue* and \$*leafFalse*.

We define the context of this state by constraining the type of the referenced record. This context together with the leaves record would be equivalent to a class definition `class Leaves extends Object` in Java.

locale *leaves* = *object this*
  for *this* :: ('*v*::*heap*, '*c*::*heap*, '*a*::*heap*) *leaves-scheme ref*

Then we extend it to add purely logical abstractions *trees* and *invar* that will be instantiated during the implementation to provide the correctness arguments we will rely on in the proofs.

locale *bddstate* = *leaves this*
  for *this* :: ('*v*::{*linorder*, *heap*}, '*c*::*heap*, -) *leaves-scheme ref* +
  fixes *trees* :: *heap* ⇒ (*bool*, '*v*, '*c*) *rtree set*
  fixes *invar* :: *heap* ⇒ *bool*

  assumes *leafTrue*:*invar s* $\Longrightarrow$ ∃ *r*. *RAcc.get s* \$*leafTrue this* = *Leaf* (*True*, *r*)
  assumes *leafFalse*:*invar s* $\Longrightarrow$ ∃ *r*. *RAcc.get s* \$*leafFalse this* = *Leaf* (*False*, *r*)
begin

To be well-formed, the heap needs to follow the implementation invariant and its trees need to be maximally shared, closed for the subtree relation and to contain the leaves.

definition *wf-heap* :: *heap* ⇒ *bool* **where**
  wf-heap s == (
  invar s
  ∧ *ref-unique* (*trees s*)
  ∧ *subtree-closed* (*trees s*)
  ∧ ((*RAcc.get s* \$*leafTrue this*) ∈ *trees s*)
  ∧ ((*RAcc.get s* \$*leafFalse this*) ∈ *trees s*))

end

Finally we add an abstract function *mk* and its specification especially ensuring it constructs a ROBDD under some preconditions.

```
locale bddstate-mk = bddstate - trees
  for trees :: heap ⇒ (bool, 'a::{linorder,heap}, -) rtree set +
  fixes mk :: 'a::{linorder,heap} ⇒ (bool, 'a, -) rtree ⇒ (bool, 'a, -) rtree
                ⇒ (bool, 'a, -) rtree Heap

  assumes mk-robdd-refs:
  effect (mk i l h) s s' t ⟹ wf-heap s ⟹ LevNode i < Min (levelOf ` {l, h})
    ⟹ {l,h} ⊆ trees s ⟹ (∀ t' ∈ trees s. robdd-refs t') ⟹ robdd-refs t
  assumes mk-interp:
  effect (mk i l h) s s' t ⟹ wf-heap s ⟹ {l,h} ⊆ trees s
    ⟹ ∀ vs. interp t vs = (if vs i then interp h vs else interp l vs)

  assumes mk-wf-heap:
  effect (mk i l h) s s' t ⟹ wf-heap s ⟹ {l,h} ⊆ trees s ⟹ wf-heap s'
  assumes mk-trees:
  effect (mk i l h) s s' t ⟹ wf-heap s ⟹ {l,h} ⊆ trees s
    ⟹ trees s' = insert t (trees s)
```

In this context we define the *app* function which applies a binary boolean operator to two BDDs.

```
function app :: (bool ⇒ bool ⇒ bool)
  ⇒ ((bool, 'a, -) rtree * (bool, 'a, -) rtree)
  ⇒ (bool, 'a::{linorder,heap}, -) rtree Heap where
  app bop (n1, n2) = do {
    if tpair is-leaf (n1, n2)
      then (constLeaf (bop (leaf-contents n1) (leaf-contents n2)))
      else (do {
         let (lh1, lh2) = select split-lh dup (n1, n2);
         let (l1, h1) = lh1;
         let (l2, h2) = lh2;
         l ← app bop (l1, l2);
         h ← app bop (h1, h2);
         mk (varOfLev (min-level (n1, n2))) l h
      })}
```

This is the only function whose termination proof is not automatic, but still very simple: it suffices to show that *select* decreases the size of a pair of trees (defined as the sum of the sizes of the trees).

Finally, *build* is a simple recursive traversal:

```
primrec build :: ('a, bool) expr ⇒ (bool, 'a, -) rtree Heap where
  build (Var i) = (do{ cf ← constLeaf False; ct ← constLeaf True; mk i cf ct})
| build (Const b) = (constLeaf b)
| build (BExpr bop e1 e2) = (do{
     n1 ← build e1;
     n2 ← build e2;
     t ← app bop (n1, n2);
     return t
})
```

5 Optimizations: Memoization and Garbage Collection

The *app* and *build* functions have been presented in their simplest form and without optimizations. We present in this section the two optimizations we have made to them.

5.1 Memoization

During the BDD construction, several identical computations can appear. This happens mostly within the recursive calls of the *app* function during which the binary operation stays the same and identical pairs of BDDs can arise by simplifications.

In order to avoid these redundant computations, the immediate solution is to use a memoization table – recording the arguments and the result for each of its calls and returning directly the result in case the arguments already have an entry in the table.

We add this memoization table in the state by extending the record containing the leaves and defining the $*appmemo* accessor.

Then the only changes to the *app* function are the memoization table lookup before the eventual calculation and the table update after:

```
function app-rec where
  app-rec bop (n1, n2) = do {
  m ← this·$appmemo;
  (case m-lookup (ref-of n1, ref-of n2) m of
      Some t ⇒ t
    | None ⇒ do {
      t ← ...;
      memo-return (ref-of n1, ref-of n2) t
    })
  }

definition app where
  app bop tp = do {
    RAcc.update $appmemo this m-empty;
    app-rec bop tp
  }
```

By adding an invariant on the memoization table, the proof changes follow the function ones. With a case distinction on the result of the table lookup for the arguments, if there is an entry for them, the result follows the invariant, else the original proof remains and the result following the invariant is stored in the table.

5.2 Garbage collection

Using an association table avoids duplication of nodes and allows to share them. However, recording all created nodes since the start of the algorithm can lead to a very huge memory usage. Indeed keeping a reference to a node in an association table prevents the JVM garbage collector to collect nodes that could have been discarded during BDD simplifications.

To prevent this waste of space, one solution is to use weak references [17]. Nevertheless, it would be impossible to formalize them as Isabelle's code extractor has no influence on the JVM garbage collector. Thus we would have to replace some references by weak references during the code generation and to rely on the correctness of this selective transformation.

An other solution is to use a mechanism of garbage collection removing unused entries from the association table. While this solution duplicates the JVM garbage collection, it allows its verification.

We chose to implement this garbage collection by a reference counting variant. The principle of reference counting is simply to store in the nodes the number of references to them. Instead of counting references for all nodes, we only count them for the BDD roots. This allows to keep the *mk* function independent from the reference count. Then, we parametrized the development with a garbage collection function *gc* whose specification ensured the preservation of reachable nodes. We call it in the *build* function when the association table becomes too large.

For this improvement, the proof additions were consequent. Indeed, several mutations to the reference counter appears in the functions, causing inner modifications in proofs. Moreover the invariants on *trees* had to be weakened to allow the garbage collection.

6 Implementation

It is now time to implement the abstract function *mk* as well as the purely logical functions *invar* and *trees*. We wrote two implementations for it and we present the most efficient one using an hash-map provided by the Collection Framework [21].

Following its specification, *mk* needs to ensure the maximal sharing of nodes. To do this, we use a table associating the components of a node (its children and variable value) to itself. Then by looking in this table, we know if a BDD has already been created and it can be returned. We add this table in the state:

record ('*v*, '*c*) *bddstate-hash* =
('*v*, ('*c refCount-scheme ref* × '*c refCount-scheme ref*, (*bool*, '*v*, '*c*) *rctree*) *hashmap*, '*c*) *leaves-memo* +
hash :: ('*v* × '*c refCount-scheme ref* × '*c refCount-scheme ref*, (*bool*, '*v*, '*c*) *rctree*) *hashmap*

And then we define two auxiliary monadic function *add* and *lookup* to abstract from the table accesses:

```
definition add
  :: 'v ⇒ (bool, 'v, -) rctree ⇒ (bool, 'v, -) rctree ⇒ (bool, 'v, -) rctree Heap where
  add i l h = do{
    x ← newNode i l h;
    RAcc.rmap $hash (ahm-update (i, ref-of l, ref-of h) x) this;
    return x
  }
```

```
definition lookup
  :: 'v ⇒ (bool, 'v, -) rctree ⇒ (bool, 'v, -) rctree ⇒ (bool, 'v, -) rctree option Heap
where
  lookup i l h = do{
  hm ← this·$hash;
  return (ahm-lookup (i, ref-of l, ref-of h) hm)
  }
```

which are used in the definition of *mk*:

```
definition mk where
  mk i l h = (if ref-equal l h then return l else do{
    to ← lookup i l h;
    (case to of None ⇒ add i l h | Some t ⇒ return t)  })
```

The garbage collector *gc* is then also implemented using two auxiliary monadic functions *referencedSet* – computing the set of nodes reachable from a node with a non-null reference count – and *hash-restrict* – restricting the domain of the hash table to the set given as argument:

```
definition gc :: unit Heap
where
  gc = do {
    hs ← referencedSet;
    hash-restrict hs
  }
```

and to avoid too frequent calls to the garbage collector, it is triggered only when the table size exceeds 10000:

```
definition gc-cond :: bool Heap
where
  gc-cond = do{
    hm ← this·$hash;
    return (ahm-size hm > 10000)
  }
```

```
definition cond-gc :: unit Heap
where
  cond-gc = do{ b ← gc-cond; if b then gc else return () }
```

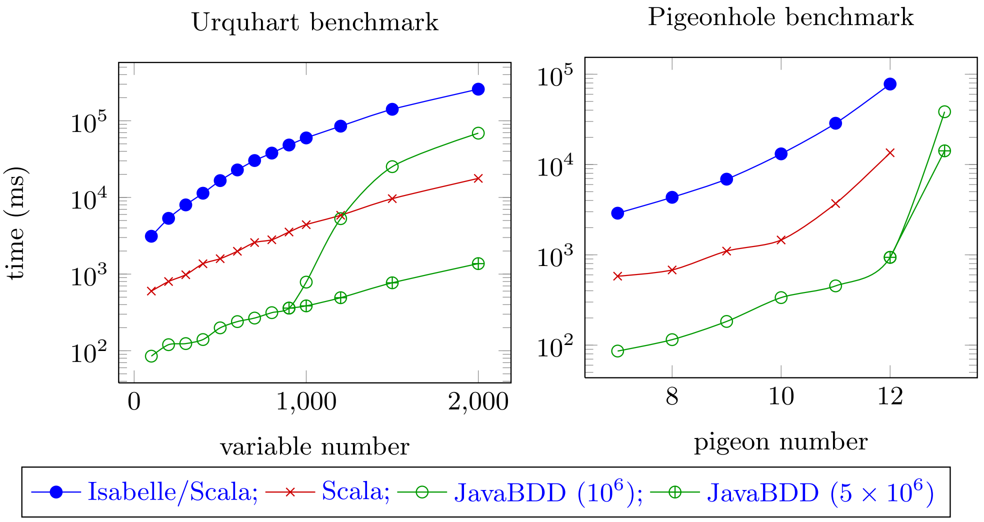
We can then use these functions satisfying the specifications of the locales to interpret them and obtain instantiated *app* and *build* functions for which we can generate code.

7 Performance evaluation

Finally we evaluate the performance of our BDD construction development.

As a comparison point we developed a BDD package directly in Scala whose code would be naively expected from the code generation from the Isabelle theories. This allows to evaluate the efficiency of the default code generation of Isabelle into Scala wrt our encoding of objects. We also compare these two implementations with a third one being an highly optimized BDD library called JavaBDD [18] providing a Java interface to several BDD libraries written in C or Java.

For this evaluation we construct BDDs for two kinds of valid formulas. The first one is the Urquhart's formulae U_n defined by $x_1 \Leftrightarrow (x_2 \Leftrightarrow \ldots (x_n \Leftrightarrow (x_1 \Leftrightarrow \ldots (x_{n-1} \Leftrightarrow x_n))))$. The second one is a formulae P_n stating the pigeonhole principle for $n+1$ pigeons in n holes *i. e.* given that $n+1$ pigeons are in n holes, at least one hole contains two pigeons.



For the generated code, the garbage collection is triggered when the table size exceeds the fixed value 10000.

In the Scala version, we use the standard hash map of the Scala library (`scala.collection.mutable.HashMap`) which has an adaptable size. Its garbage collection is triggered when the table size exceeds a threshold value. The table size causing the garbage collection is initially set to 1000 and if the table size after garbage collection exceeds this threshold, it is increased by one half.

On the other side, JavaBDD needs an initial table size increased after garbage collections by an initially fixed value. In the benchmarks, we set it to 10^6 and 5×10^6. We can see that increasing the initial table size for the JavaBDD version leads to better performances for large expressions but then more space is needed even for smaller ones. Then the difficulty to choose the right size is reported to the library user.

While the generated code is slower than the others, it is still able to construct BDDs for large expressions thanks to the garbage collection.

We suspect several causes of inefficiency and space usage compared to the Scala version:

- Monad operations are converted into method calls. The presence of monadic operators at each line could explain some performance penalties.
- During the code generation, a "Ref" class is introduced to allow reference manipulations in Scala. This is unnecessary for objects as long as we don't use references on primitive types or access to the value referenced only through accessors.
- Records extensions are translated to class encapsulations leading to several indirections at the time of attribute accesses.

Improving on these points is current work and we think that these optimizations in the code generation could improve the general performances, to the point that the generated code would be comparable to the hand-written code.

8 Conclusions

This paper has presented a verified development of a BDD package in the Isabelle proof assistant, with fully operational code generated for the programming language Scala. The development time for the formalization itself (around 6 person months) is difficult to estimate exactly, because it went hand in hand with the development of the methodology. In the light of the performance of the code obtained, the result is encouraging, and we expect to explore the approach further for the development of verified decision procedures.

As mentioned in the outset, bit-level optimizations could be introduced, at the price of adding one or several refinement layers, with corresponding simulation proofs. Even though feasible, this is not our current focus, since we aim at a method for producing reasonably efficient verified code with a very moderate effort.

Consequently, our method is not a panacea. As far as the class and object model is concerned: The type system has intentionally been kept simple in the sense that classes are essentially based on record structures and inductive data types as found in ML-style polymorphism. Such a choice is incompatible with some OO features such as late method binding, which appears to be acceptable in the context of high-integrity software. As mentioned in Section 7, we are aware of some inefficiencies that arise from nesting of objects during code extraction to Scala, and which have as deeper cause a mismatch between pointer-manipulating languages (such as C and C++, as also incorporated in the Imperative_HOL framework) and pure OO languages, such as Java and Scala. We will address this issue in our future work.

Also the limitation of representable structures to “trees with sharing” appears to be a severe limitation at first glance, but in combination with cute functional data structures [24], it is possible to represent quite general pointer meshes (see for example the verification of the Schorr-Waite garbage collector [16] using a “zipper” data structure).

References

1. Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6):81–91, 2011.
2. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.
3. François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, August 2011.
4. Sascha Böhme, Michał Moskal, Wolfram Schulte, and Burkhart Wolff. HOL-Boogie — An Interactive Prover-Backend for the Verifying C Compiler. *Journal of Automated Reasoning*, 44(1–2):111–144, February 2010.
5. Achim D. Brucker and Burkhart Wolff. Extensible Universes for Object-Oriented Data Models. In Jan Vitek, editor, *Proceedings of the European Conference of Object-Oriented Programming (ECOOP 2008)*, LNCS 5142, pages 438–462. Springer-Verlag, Paphos, Cyprus, July 2008.
6. Achim D. Brucker and Burkhart Wolff. Semantics, Calculi, and Analysis for Object-Oriented Specifications. *Acta Informatica*, 46(4):255–284, 2009.
7. Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.
8. Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. Imperative Functional Programming with Isabelle/HOL. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *TPHOLs ’08: Proceedings of the 21th International Conference on Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2008.
9. Mohamed Chaabani, Mohamed Mezghiche, and Martin Strecker. Vérification d’une méthode de preuve pour la logique de description $\mathcal{ALC}$. In *Proc. 10ème Journées Approches Formelles dans l’Assistance au Développement de Logiciels*, 2010. To appear.
10. David Cock, Gerwin Klein, and Thomas Sewell. Secure Microkernels, State Monads and Scalable Refinement. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs’08)*, volume 5170 of *Lecture Notes in Computer Science*, pages 167–182. Springer-Verlag, 2008.
11. Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In *Proc. Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, pages 23–42, 2009.
12. Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *Proceedings CAV 2007*, pages 173–177, 2007.

13. Frama-C. http://frama-c.com/.
14. Mathieu Giorgino and Martin Strecker. Towards the verification of efficient BDD algorithms. http://www.irit.fr/ Mathieu.Giorgino/Publications/GiSt2011BDD.html.
15. Mathieu Giorgino and Martin Strecker. BDDs verified in a proof assistant (Preliminary report). In A.V. Anisimov and M.S. Nikitchenko, editors, *Proc. TAAPSD*, Univ. Taras Shevchenko, Kiev, 2010.
16. Mathieu Giorgino, Martin Strecker, Ralph Matthes, and Marc Pantel. Verification of the Schorr-Waite algorithm - From trees to graphs. In *Proc.20th International Symposium on Logic-Based Program Synthesis and Transformation (Lopstr)*, LNCS. Springer, July 2010. To appear.
17. J. J. Hallett and A. J. Kfoury. A formal semantics for weak references. Technical report, in ISMM 06: Proceedings of the 2006 international symposium on Memory management, 2005.
18. JavaBDD. http://javabdd.sourceforge.net/.
19. KIV. http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/kiv/.
20. Sava Krstic and John Matthews. Verifying BDD Algorithms through Monadic Interpretation. In *Verification, Model Checking, and Abstract Interpretation (VMCAI 2002), volume 2294 of LNCS*, pages 182–195. Springer, 2002.
21. Peter Lammich and Andreas Lochbihler. The Isabelle Collections Framework. In *Proc. ITP'10*, volume 6172 of *LNCS*, pages 339–354. Springer, July 2010.
22. Filip Marić. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theor. Comput. Sci.*, 411:4333–4356, November 2010.
23. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer Verlag, 2002.
24. Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1998.
25. Veronika Ortner and Norbert Schirmer. Verification of BDD Normalization. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 2005*, LNCS 3603, pages 261–277. Springer, 2005.
26. W. Reif, J. Ruf, G. Schellhorn, and T. Vollmer. Do you trust your model checker? In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer Aided Design (FMCAD)*. Springer LNCS 1954, November 2000.
27. Alexander Schimpf, Stephan Merz, and Jan-Georg Smaus. Construction of Büchi Automata for LTL Model Checking Verified in Isabelle/HOL. In Tobias Nipkow and Christian Urban, editors, *22nd Intl. Conf. Theorem Proving in Higher-Order Logics (TPHOLs 2009)*, LNCS 5674, Munich, Germany, 2009. Springer.
28. Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.
29. Kumar Neeraj Verma, Jean Goubault-Larrecq, Sanjiva Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In *ASIAN 2000, 6th Asian Computing Science Conference*, LNCS 1961. Springer, 2000.
30. Friedrich W. von Henke, Stephan Pfab, Holger Pfeifer, and Harald Rueß. Case Studies in Meta-Level Theorem Proving. In Jim Grundy and Malcolm Newey, editors, *Proc. Intl. Conf. on Theorem Proving in Higher Order Logics*, LNCS 1479, pages 461–478. Springer, September 1998.

A Liskov Principle for Delta-Oriented Programming

Reiner Hähnle[1] and Ina Schaefer[2]

[1] Department of Computer Science and Engineering
Chalmers University of Technology, SE-41296 Gothenburg
reiner@chalmers.se
[2] Institute for Software Systems Engineering
Technical University of Braunschweig, D-38106 Braunschweig
i.schaefer@tu-braunschweig.de

Abstract. Formal verification techniques for software product families not only analyse individual programs, but must act on the artifacts and components which are reused to obtain software products. As the number of products is exponential in the number of artifacts, it is essential to perform verification in a modular fashion instead of verifying each product separately: the goal is to reuse not merely software artifacts, but also their verification proofs. When code reuse is based on standard class-based inheritance in OO programming, Liskov's principle is a well-known device to achieve modular verification. Software families, however, generally employ more flexible program modularization techniques than inheritance. Delta-oriented programming is an approach to implement a family of OO programs where code reuse is achieved via stepwise transformation of a core program. In this paper, we define a Liskov principle for delta-oriented programming and show that it achieves modular verification of software families developed in that paradigm.

1 Introduction

Diversity is prevalent in modern software systems in order to meet different customer requirements and application contexts [29]. Formal modeling and verification of software product families have attracted considerable interest recently [32, 5, 12, 8]. The challenge is to devise validation and verification methods that work at the level of families, not merely at the level of a single product. Given the combinatorial explosion in the number of possible products even for small software families, efficient verification techniques for families are essential. For verification techniques to scale, they have to be modular in the artifacts that are reused to build the different variants of the software family.

In the area of object-oriented programming, Liskov's principle of behavioral subtyping [24] is an important means and guideline to achieve modular verification. It is also an important theoretical tool to investigate theories of specification and refinement. However, in the majority of approaches to family-based software development the principles of reuse are not founded on class-based inheritance. Instead, more flexible program modularization techniques, such as aspect-oriented programming [19], feature-oriented programming [4], or delta-oriented programming [26] (which can be considered as an extension of feature-oriented programming [28]) are applied. For this family of languages, there exist a few insular approaches to incremental verification [8, 32]. However, there is, to the best of our knowledge, no notion corresponding to Liskov's principle for inheritance. As a consequence, there is no approach that would allow modular functional verification for software families.

In this paper, we analyse delta-oriented programming (DOP) of software families [26]. In DOP, a software family is developed from a designated core program and a set of delta modules that alter the core program to realize other program variants. In Sect. 2, we provide sufficient background on DOP to make the paper self-contained. Functional program properties are specified based on design-by-contract [25] by providing class invariants and method contracts. The core program is specified like any standard program, while the deltas can add or remove method contracts and class invariants to reflect the changes in the code carried out by a delta. As it is detailed in Sect. 3, by applying the deltas and their specifications to a core program and its specification, a program variant (called *product*) and its corresponding specification is generated.

To support modular reasoning for software families implemented by DOP, we develop a Liskov principle for delta modules in Sect. 4. This principle restricts the changes that a delta module may make to the specification of the core program. Based on this principle, in Sect. 5 we devise a modular proof principle that relies on the approximation of called methods by their first introduced variant. If the Liskov principle for DOP holds, we show that it suffices to analyze the core program and each delta in isolation to establish the correctness of all products. In Sect. 6, we discuss the consequences of DOP for reasoning about invariants. Furthermore, in Sect. 7, we show a proof transformation to construct proofs for program variants from the proofs carried out during the modular analysis. In Sect. 8, we review related work. We conclude and discuss future work in Sect. 9.

2 Delta-Oriented Programming

The basis of this paper is the modeling language ABS (Abstract Behavioral Specification Language) [9, 13] where program variability is represented by DOP. ABS is a class-based, concurrent OO language without class inheritance. Interfaces which are implemented by classes can be extended to provide a taxonomy similar to class inheritance. We consider sequential ABS programs, because there is no standard notion of contract or Liskov principle for concurrent programs.

2.1 Preliminaries

A family of ABS programs is represented by an ABS core program $\mathcal{C}$ and a partially ordered set of deltas $\mathcal{D}$, together called *delta model.* Deltas can add, remove, and modify classes from the core program. Modification of a class changes the internal class structure by adding and removing fields and methods and by changing method implementations. A method implementation can be completely replaced or wrapped using the **original** call. The keyword **original** denotes a call to the most recent version of a method, analogous to the "super" call in class-based inheritance, however, in contrast to the latter, **original** calls are statically resolved when building a concrete product. Accordingly, an occurrence of **original** requires to know exactly which delta has been used most recently in order to arrive at the current partial product. Therefore, we will restrict the usage of **original** to only those cases where the target of **original** can be uniquely determined. To avoid technical complications that are orthogonal to the problem of modular product family analysis treated here, we exclude recursive calls.

A partial order between deltas resolves conflicts if two deltas alter the same entity of an ABS program. This ensures that for a given set of deltas a unique ABS program variant is always generated. Without loss of generality, we assume that the partial order of the deltas is expressed as a total order on a partition [27] denoted as $[\delta_{11} \cdots \delta_{1n_1}] < \cdots < [\delta_{h1} \cdots \delta_{hn_h}]$. We assume that all deltas in one element $\left[\delta_{j1} \cdots \delta_{jn_j}\right]$ of the partition are compatible and that the partitions are pairwise disjoint. A set of deltas is called *compatible* if no class added or removed in one delta is added, removed or modified in another delta contained in the same set, and for every class modified in more than one delta, the fields and methods added, modified or removed are distinct. Thus, the order of application of the deltas in the same partition element does not matter. The parts of the partition, however, must be applied in the specified order to ensure that a unique product is generated for a selected subset of deltas. We call h the *height* of the delta model and $\max\{n_1, \ldots, n_h\}$ its *width*. The number of possible products in a delta model is bounded by $(2^w)^h$ (the number of subsets that can be selected).

On an abstract level, variability is usually represented by features, that is, user-visible product characteristics. A feature model [18] defines the set of valid feature combinations, i.e., the products of the software family. To connect feature models and program variability specified by the delta model, a product line specification is provided where an application condition over the features is attached to each delta. These conditions can be Boolean constraints over the features and specify for which feature configurations a delta can be applied.

A program for a particular feature configuration is generated by selecting the subset of deltas with a valid application condition and applying them to the core program in a linear order that is compatible with the partial order of the delta model. The generation of a program variant from a core program $\mathcal{C}$ and a delta model $\mathcal{D}$ is written as $\mathcal{C}\delta_1 \cdots \delta_p$ where for all $1 \leq i \leq p$, it holds that $\delta_i = \delta_{kl}$ and $\delta_{i+1} = \delta_{k'l'}$ such that $\delta_i \neq \delta_{i+1}$ and $k \leq k'$. We know that the number of applied deltas is bounded by $p \leq h * w$. We use the following obvious notation to access classes `C`, fields `f`, and methods `m` within (partial) products: $\mathcal{C}\delta_1 \cdots \delta_n.\mathtt{C.m.f}$, etc.

It is possible that a sequence of delta applications $\mathcal{C}\delta_1 \cdots \delta_n$ is not a product, for example, when an accessed method or field was not declared before. Since we want to reason only about well-defined products, this causes technical complications. One way to avoid them is to stipulate that all sequences of deltas lead to type-safe products which can be enforced by adding suitably composed intermediate deltas. As this would bloat delta models, we employ a more natural restriction sufficient for our purposes: assume $P = \mathcal{C}\delta_1 \cdots \delta_n$ is any product and $P\delta_{n+1} \cdots \delta_{n+k}$ is a product with a minimal number of deltas obtained from P (that is, the same product cannot be produced with less deltas); then any method is introduced or modified at most once in $\delta_{n+1}, \ldots, \delta_{n+k}$.[3] We call a delta model with this property *regular*. In addition, we assume that without loss of generality each delta occurring in a regular delta model is used in at least one product.

2.2 Running Example

Our main example is a simple product family of bank accounts depicted in Fig. 1. The core program contains a class `Account` implementing an interface `IAccount`. The class `Account` contains a field `balance` for storing the balance of the account and a method `deposit` to update the balance. The product family contains two deltas. Delta `DFee` modifies `Account` by introducing a transaction fee modeled by a parameter that is instantiated when a concrete program variant is generated. Delta `DOverdraft` adds a limit to the account restricting the possible overdraft.

The feature model for this product family contains the mandatory feature `Basic` implemented by the `Account` class. Furthermore, the product family has the optional feature `Fee` with a fee parameter and the optional `Overdraft` feature. The product line declaration at the bottom of Fig. 1 provides the connection between features and deltas in the **when** clauses. These state that the delta `DFee` realizes the feature `Fee` and that delta `DOverdraft` implements the feature `Overdraft`. The **after** clause provides the application ordering between the deltas which is generally described by an ordered partition (see above). The product family of bank accounts gives rise to four program variants: one with only the `Basic` features, one with the `Basic` and `Overdraft` feature, one with the `Basic` and `Fee` features and one with all three features where each product containing the feature `Fee` varies in the concrete value of the fee which is instantiated with a concrete value when a particular product is generated.

3 Specifying Deltas

To be able reason about behavioral properties of program variants, a property specification technique for core programs and deltas has to be provided that allows generating the program variants together with their specification.

3.1 Design By Contract

We use a specification discipline for both core programs and deltas that is derived from design by contract [25] and closely modelled after the JML approach [22].

[3] Removing and re-introducing a method corresponds to a modification.

```
module Account;

interface IAccount { Unit deposit(Int x); }

class Account implements IAccount {
  Int balance = 0 ;
  Unit deposit(Int x) { balance = balance + x; }
}
delta DFee(Int fee) {
  modifies class Account {
    modifies Unit deposit(Int x) { if (x>=fee) original(x-fee); }
  }
}
delta DOverdraft() {
  modifies class Account{
    adds Int limit;
    modifies Unit deposit (Int x) { if (balance + x > limit) original(x); }
 }
}
productline AccountPL {
  features Basic, Overdraft, Fee;
  delta DFee (Fee.amount) when Fee;
  delta DOverdraft after DFee when Overdraft;
}
```

Fig. 1. A Bank Account Product Family in ABS [9, 13]

Definition 1. *A* program location *is an expression that refers to an updatable heap location, such as variables, formal parameters, field access expressions, or array access expressions. We work with first-order signatures that include all locations of the program of interest. A* contract *for a method* `m` *consists of:*

1. *a first-order formula* `r` *called* precondition *or* requires clause*;*
2. *a first-order formula* `e` *called* postcondition *or* ensures clause*;*
3. *a set of program locations* `a`*, called* assignable clause*, that occur in* `m` *and whose value can potentially be changed during execution.*

We extend our notation for accessing class members to cover the constituents of contracts: `C.m.r` is the requires clause of method `m` in class `C`, etc.

Let $\mathtt{m}(\overline{\mathtt{p}})$ be a call of method `m` with formal parameters $\overline{\mathtt{p}}$. A *total correctness program formula* in dynamic logic [6, Chapter 3] is of the form $\langle \mathtt{m}(\overline{\mathtt{p}})\rangle \Phi$ and means that whenever `m` is called then it terminates and in the final state Φ holds where Φ is either again a program formula or a first-order formula. One part of the semantics of a method contract is expressed as a *total correctness formula* of the form $\mathtt{r} \to \langle \mathtt{m}(\overline{\mathtt{p}})\rangle \mathtt{e}$. (Partial correctness does not add anything essential to our discussion, so we omit it for brevity.) The second part of contract semantics is correctness of the assignable clause. One must ensure that m can change the value of no other program location than of those listed in a. It is possible to encode this property with the help of program formulas [17]. This encoding is of no further interest in this paper, so we do not give details, but we assume there is a program formula $A(\mathtt{a}, \mathtt{m})$ that states correctness of the assignable clause. The following monotonicity condition can be assumed:

$$\mathtt{a}' \subseteq \mathtt{a} \;\wedge\; A(\mathtt{a}, \mathtt{m}) \rightarrow A(\mathtt{a}', \mathtt{m}) \tag{1}$$

Definition 2. *A method* m *of class* C satisfies *its contract if the following holds:*

$$\mathtt{C.m.r} \rightarrow \langle \mathtt{m}(\overline{\mathtt{p}}) \rangle \mathtt{C.m.e} \quad \wedge \quad A(\mathtt{C.m.a}, \mathtt{C.m}) \tag{2}$$

In addition to contracts, we allow first-order formulas i to be attached as *invariants* to classes. We permit to write invariants directly in front of the element they relate to (e.g., a field declaration). However, as we consider all specifications to be globally visible in this paper, these simply are part of the invariant of a class. Hence, we assume that each class C has a unique invariant C.i. As usual, the semantics of invariants requires to establish two properties: (i) after initialization of a class its invariant holds and it does not invalidate the invariant of any other class, and (ii) if an invariant holds just before the execution of a method, then it holds again immediately after termination of that method. As a consequence of global visibility of invariants, the invariants of *all* classes must be maintained by *all* methods. In the absence of modularity constructs, this is the usual situation in specification of object-oriented programs.

The presence of contracts makes formal verification of complex programs feasible, because each method can be verified separately against its contract and called methods can be approximated by their contracts. The assignable clause of a method limits the program locations a method call can have side effects on.[4] In Sect. 7, we show how contracts are used in proofs.

Fig. 2 shows the specification of the core program of the Bank Account product family. The method deposit is specified with a method contract whose precondition in the @requires clause denotes that the balance should be positive. The postcondition in the @ensures clause denotes that the balance after the method call is at most the balance before the method call (accessed by the JML \old keyword) plus the value of the input parameter. As there is no explicitly specified invariant, the class invariant of Account is simply true.

3.2 Specification Deltas

We want to be able to denote in a structured manner those parts of contracts and invariants that must be modified in order to reflect the changes that are

[4] We are well aware that this basic technique is insufficient to achieve modular verification. Advanced techniques for modular verification, e.g. [3], would obfuscate the fundamental questions considered in this paper and can be superimposed later.

```
class Account implements IAccount {
    Int balance = 0;

    @requires x > 0;
    @ensures balance <= \old(balance) + x;
    Unit deposit(Int x) { balance = balance + x }
}
```

Fig. 2. Specification of Core Bank Account

embodied in a given delta. The specification approach of [8] allows (i) to add and remove invariants as well as (ii) to add and remove whole contracts in deltas. This is too coarse for our purposes, so we make the following refinement:

- in deltas, the addition, removal, and modification of contracts can be specified separately for requires clauses, ensures clauses, and assignable clauses;
- we permit the usage of the keyword **original** in clauses of contracts with the obvious semantics provided that the contract to which **original** refers can be uniquely determined;
- since the invariant of a (partial) product is always global and the implicit conjunction of all invariants introduced in the core and in the constituent deltas, modifying invariants and the usage of **original** in invariants makes no sense. Hence, invariants can only be explicitly added or removed in deltas.

A missing specification clause is equivalent to, for example, "`@requires` **original**" (or to "`@requires true`" in the case of the first occurrence of a method). Fig. 3 shows the modification to the specification caused by the delta `DFee`. The contract of method `deposit` is changed by replacing the postcondition. The precondition remains unchanged. Additionally, an invariant for the field `fee` is added to the class `Account` which states that the value of `fee` should be non-negative.

```
delta DFee(Int fee) {
  modifies class Account {
    adds @invariant fee >= 0;
    modifies @ensures balance <= \old(balance) + max(x-fee,0) ;
    modifies Unit deposit(Int x) { if (x>=fee) original(x-fee); }
  }
}
```

Fig. 3. Delta `DFee` with its Specification Delta

4 Liskov's Principle

Liskov's principle of behavioral subtyping [24] is an important means to achieve modularity for behavioral specification and verification. In this section, we recall Liskov's principle for standard class-based inheritance and transfer it to DOP.

4.1 Standard Object-Oriented Design with Code Inheritance

In standard object-oriented programming with code inheritance Liskov's [24] principle states the following:

1. The invariant of a subclass must imply the invariant of its superclasses.
2. The precondition of a method overridden in a subclass must be implied by the precondition of the superclass method and the postcondition of a method overridden in a subclass must imply the postcondition of the superclass method.
3. When assignable clauses are present, the assignable locations in a subclass must be a subset of the assignable locations in the superclass.

We distill the essence of the last two points into a relation on contracts:

Definition 3. *For two methods* $\mathtt{m}$*,* $\mathtt{m}'$ *let* $\mathtt{m.r}$*,* $\mathtt{m.e}$*,* $\mathtt{m.a}$*, and* $\mathtt{m}'.\mathtt{r}'$*,* $\mathtt{m}'.\mathtt{e}'$*,* $\mathtt{m}'.\mathtt{a}'$ *be different contracts (*$\mathtt{m} = \mathtt{m}'$ *is possible). We say that the first contract is* more general *than the second (or the second is more* specific *than the first) whenever the following holds:*

$$(\mathtt{m.r} \rightarrow \mathtt{m}'.\mathtt{r}') \wedge (\mathtt{m}'.\mathtt{e}' \rightarrow \mathtt{m.e}) \wedge (\mathtt{m}'.\mathtt{a}' \subseteq \mathtt{m.a}) \tag{3}$$

The following lemma is immediate by the definition of contract satisfaction (Def. 2), propositional reasoning, monotonicity of postconditions in total correctness formulas, and monotonicity of assignable clauses (1). It will be tacitly used in the following to establish satisfaction of method contracts.

Lemma 1. *If a method* $\mathtt{m}'$ *satisfies its contract then its satisfies as well any contract that is more general.*

Consequently, if a specification follows Liskov's principle, then *behavioral subtyping* is guaranteed provided that all methods satisfy their contract and maintain the invariants. This means that an object can be replaced by any object with a subtype without changing the behavior of the program.

4.2 Delta-Oriented Specification

We propagate delta-oriented programming (DOP) [26, 27] as the fundamental technique for code reuse, in contrast to inheritance. As a prerequisite for modular verification in this context, it is necessary to understand how Liskov's principle can be ported to a DOP setting and what it means exactly.

To develop a Liskov principle for DOP, we consider the code and specification elements that can be changed by deltas: adding and removing methods together with their contracts is uncritical, since our assumption on type-safety guarantees that such a method has never been called before, respectively, will not be called afterwards. It is sufficient to prove the contract of newly added methods, but that of existing methods cannot be affected. If a newly added method should be integrated into an existing program, modifications of existing methods have to be specified in other applied deltas. This leaves modification of existing methods and contracts as well as the removal and addition of invariants to look at.

To preserve the behavior of a method that is modified by a delta, it is sufficient to follow the same principle as in behavioral subtyping, i.e, to make contracts more specific (Def. 3). Specifically, this is the case, whenever the modified contract of m has a requires clause of the form C.m.r.**original** $\vee$ r′, an ensures clause of the form C.m.e.**original** $\wedge$ e′, and for the assignable clause a of the modified contract, a $\subseteq$ C.m.a.**original** holds.

The tricky issue is that references to **original** and, therefore, to method calls and contracts are only resolved when a concrete product is being built. In Sect. 5, we show that under certain restrictions one can verify a delta model without having to look at all its exponentially many products.

Regarding removal and addition of invariants, certainly, we must exclude the possibility to remove invariants (although in principle possible in deltas), because this might invalidate the contracts of arbitary methods added either in the core or in any delta. This would require to reprove all contracts in all exponentially many products. A straightforward counterpart of the first item in Liskov's principle stated at the beginning of Sect. 4.1 would require adding only invariants that are implied by previously existing ones. We discuss essentially this situation in Sect. 6.1 below. This approach is rather restrictive, but adding *new* invariants and reproving them in a compositional manner is non-trivial and discussed in Sect. 6.2.

5 Compositional Verification of Delta Models

The main advantage of having a Liskov-like principle for the specification of deltas is that we can follow a *compositional* verification approach. This means that we can ensure with only a polynomial number of proofs the behavioral correctness of an exponential number of products. This obviously is a key property in ensuring feasibility for product family verification, because even very small product families have an infeasible number of products.

In this section, we focus on the verification of method contracts and cover the verification of invariants in Sect. 6. We need to ensure that all methods in any product satisfy their contract. We do this in two steps:

Verification of the Core. This is standard and means simply to prove that all methods m in a core program $\mathcal{C}$ satisfy their contract (Def. 2).

Verification of the Deltas. For each method m added or modified in a delta δ, we must establish its contract. We allow the usage of the keyword **original**

in contracts only in the syntactically restricted form mentioned in Sect. 4.2. For each method $\mathtt{m}$, we must show the proof obligation

$$\delta.\mathtt{m}.\mathtt{r} \rightarrow \langle \mathtt{m}(\overline{\mathtt{p}})\rangle \delta.\mathtt{m}.\mathtt{e} \quad \wedge \quad A(\delta.\mathtt{m}.\mathtt{a}, \delta.\mathtt{m}) \tag{4}$$

Additionally, we need to ensure that the contract of each $\delta.\mathtt{m}$ is more specific than the contracts provided and verified for $\mathtt{m}$ in all previous deltas used for any product. As the actual set of applied deltas cannot be known before product generation, to get a compositional verification method avoiding the generation of exponentially many products, we have to assume "the worst".

For the verification of (4), let us first analyse the methods called inside $\delta.\mathtt{m}$: if a method $\mathtt{n}$ is called[5] in $\delta.\mathtt{m}$ and does not occur in δ itself, then we use the method contract associated with the *first* introduction of $\mathtt{n}$ in the given delta model (i.e., in a δ_{ij} with minimal index i). As subsequent contracts of $\mathtt{n}$ can only get more specific according to our Liskov principle, this ensures that the call is valid for all possible versions of $\mathtt{n}$. Likewise, we use the "largest" assignable set of locations. If $\mathtt{n}$ occurs in δ, we simply use the contract of $\delta.\mathtt{n}$.

Definition 4. *If all methods occurring in a delta δ satisfy their contract, where the contracts of called methods have been selected as outlined above, we say that the δ is* verified.

Next, we ensure that the contract of $\delta.\mathtt{m}$ is more specific than all previous contracts of the same method. As each method may occur at most once in a part of the partition $\mathcal{D}$ by compatibility of the deltas in the part, it is sufficient to compare the contract of $\delta.\mathtt{m}$ with the contract of the most recent (as defined below) occurrence of $\mathtt{m}$ from $\delta.\mathtt{m}$, say $\delta'.\mathtt{m}$. (If $\delta.\mathtt{m}$ was the first occurrence in $\mathcal{D}$, then there is nothing to do.) It suffices to show that the contract of $\delta'.\mathtt{m}$ is more general than the contract of $\delta.\mathtt{m}$.

Definition 5. *Assume that a method $\mathtt{m}$ occurs at least twice in a delta model with core $\mathcal{C}$ and partition $\mathcal{D} = ([\delta_{11} \cdots \delta_{1n_1}], \ldots, [\delta_{h1} \cdots \delta_{hn_h}])$, and one of the occurrences is in δ_{jk}. For convenience, we rename the core $\mathcal{C}$ into δ_{00}. Then an occurrence of $\mathtt{m}$ in δ_{il} is called* most recent *from $\delta_{jk}.\mathtt{m}$ if there is no occurrence of $\mathtt{m}$ in any $\delta_{i'r}$ with $i < i' < j$.*

Taken together, Defs. 4 and 5 provide a static (i.e., at the level of the product family) approximation of the deltas used in any possible concrete product.

A straightforward induction over the height of a delta model lifts the property that a method contract for a method $\mathtt{m}$ is more specific than the contract of the most recent occurrence from $\mathtt{m}$ to arbitrary previous occurrences of the method $\mathtt{m}$. This will be needed later in the proof of Thm. 1.

Lemma 2. *Let $\mathcal{C}$ and $\mathcal{D}$ be a delta model as in Def. 5. Assume that for the core $\mathcal{C}$ and for any δ occurring in $\mathcal{D}$ the following holds: the contract of any method $\mathtt{m}$ in δ is more specific than the contract of the most recent occurrence from $\delta.\mathtt{m}$.*

[5] In case, the call is done via the keyword **original** this simply means $\mathtt{n} = \mathtt{m}$ where $\mathtt{m}$ is not the one in δ.

Then for any two method contracts of a method `m` *occurring in any δ_{ik} and $\delta_{jk'}$ such that $i \leq j$ we have that the contract of* `m` *in δ_{ik} is more general than its contract in $\delta_{jk'}$.*

We formalize the considerations above in the following theorem:

Theorem 1. *Given a regular delta model consisting of a core $\mathcal{C}$ and a partition of deltas $\mathcal{D} = ([\delta_{11} \cdots \delta_{1n_1}], \ldots, [\delta_{h1} \cdots \delta_{hn_h}])$. Assume the following holds:*

1. *$\mathcal{C}$ satisfies its contract, i.e., equation* (2) *holds for all its methods.*
2. *For all δ occurring in $\mathcal{D}$:*
   (a) *δ is verified.*
   (b) *The contract of each method* `m` *added or modified in δ either is the first occurrence of* `m` *in the delta model or it must be more specific than the contract of the most recent method in $\mathcal{D}$ from δ.*`m`*.*

Then every product obtained from the given delta model satisfies its specification, i.e., each of its methods satisfies its contract.

Example. In the example of the bank account product family, the contract of method `deposit` that is modified by the delta `DFee` in Fig. 3 satisfies condition (2b). The contract of method `deposit` in delta `DFee` is more specific than the contract of method `deposit` in the class `Account` given in Fig. 2. Delta `DOverdraft` does not change any specification and fulfills condition (2b) trivially. During the verification of `DFee.deposit` the contract of `Account.deposit` needs to be used. One can apply Thm. 1 to the bank account example and infer that all four products satisfy their respective method contracts.

The significance of Thm. 1 lies in the fact that the number of proof tasks is only polynomial in h, w, and the number M of different methods occurring in a delta model: in the core and in each of at most $h * w$ deltas we need at most three proofs for each modified method which is in $\mathcal{O}(h * w * M)$. This is a clear advantage over providing a separate proof for each product of which there are exponentially many, resulting in $\mathcal{O}(2^{(h*w)} * M)$ many proofs.

Proof (of Thm. 1). The proof is by induction on the length p of delta sequences where we consider only those sequences that result in a product. The induction hypothesis says that each delta sequence of length p results in a product that satisfies its specification.

The induction base for $p = 0$ amounts to show the claim for the core $\mathcal{C}$ of the delta model which is taken care of by the first assumption.

Now assume that we have a product $P = \mathcal{C}\delta_1 \cdots \delta_p$ that satisfies its specification and $P' = P\delta_{p+1} \cdots \delta_{p+k}$ is any product with a minimal number of deltas obtained from it. We show that

1. any method `m` occurring in P', but not in P, satisfies its contract;
2. that the contracts of all other methods called in P' still hold.

Regarding the first item, by regularity of the delta model we can assume that there is exactly one δ in $\delta_{p+1}, \ldots, \delta_{p+k}$ where `m` is introduced or modified. By assumption (2a), from equation (4) we know that all methods in δ satisfy their contract where the method `n` called in `m` can be approximated by the "first", i.e., most general existing contract. In P', these calls to `n` are replaced by some implementation introduced or modified by some delta.

The actual contract of `n` was either introduced in δ itself or in a different $\delta_{p+1}, \ldots, \delta_{p+k}$ or somewhere in P. In the first case, since δ is verified, `n` was proven against its actual contract. If `n` came from one of the "new" deltas, by assumption (2b) the contract of `n` is more specific than the contract of the most recent occurrence in the delta model. By Lemma 2, the contract of that occurrence is more specific than the first occurrence of `n` in the delta model which was used for approximating the contract of `n` during verification of `m`. This means that the previous proof supplied by the induction hypothesis still applies.

Finally, assume the contract of `n` was introduced in P. By Lemma 2, we know that this contract must be either identical to or more specific than the first occurrence of `n` in the delta model. Since the latter contract was used in verification of `m` the result holds.

For proving the second item above, assume `m` is any method that is defined and verified in a $\delta \in P$. The case which we need to check is that `m` calls a method `n` whose specification was overridden in $\delta_{p+1}, \ldots, \delta_{p+k}$. From Lemma 2 we know that the contract of the later occurrence is more specific than the contract of `m` used in P. Therefore, the new contract is still applicable. Together with the assumption that all $\delta_{p+1}, \ldots, \delta_{p+k}$ are verified, this closes the proof. □

6 Verification of Invariants

As mentioned above, we assume that all invariants are global: each method must satisfy all invariants. Therefore, one can assume that there is exactly one invariant for each product. In more fine-grained approaches, one can limit the visibility of invariants by making them private and attaching them to specific class features or restrict their accessibility with type systems, however, this is an orthogonal issue to the problem at hand.

Invariants can be viewed as a special case of method contracts where the requires and ensures clause are identical. But this is exactly what makes it difficult to fit invariants into the above framework where contracts become more specific after the application of deltas causing requires and ensures clauses to diverge. Specifically, for the reason stated in Sect. 4.2 we exclude removal of invariants.

6.1 Core Invariants

Our first take on invariants is a direct rendering of the first item in Liskov's principle. As explained in Sect. 4.2, this means that only invariants are added that are implied by existing ones. This amounts to permit the introduction of invariants only in the core. All subsequent deltas use the same invariant. Proof obligations of the kind (2) and (4) are extended with the core invariant $\mathcal{C}.\mathtt{i}$:

$$(\mathtt{m.r} \ \wedge \ \mathcal{C}.\mathtt{i}) \rightarrow \langle \mathtt{m}(\overline{\mathtt{p}}) \rangle \mathtt{m.e} \qquad \wedge \qquad \mathcal{C}.\mathtt{i} \rightarrow \langle \mathtt{m}(\overline{\mathtt{p}}) \rangle \mathcal{C}.\mathtt{i} \tag{5}$$

We continue to use the specification and verification discipline of Sect. 5, but employ proof obligations of the form (5). The number of proofs stays the same, even though some may be harder to establish. The proof of Thm. 1 is done such that at the first occurrence of a method declaration its contract and the invariant is established. Hence, the invariant is available in subsequent verification steps of the deltas.

Even though it may seem rather restrictive to use only core invariants, there are a number of important advantages:

1. The number and complexity of proof tasks stays manageable.
2. Thm. 1 providing a compositional verification approach stays valid.
3. If an invariant $\mathtt{i}'$ in one delta was added, then this invariant must be shown to hold even for the methods not changed in that delta. This means that either $\mathtt{i}'$ has a signature disjoint from the core or it would have been possible to add and show $\mathtt{i}'$ already in the core. In the second case, the core invariant was chosen too weak. We discuss the first possibility in more detail in Sect. 6.2.

6.2 Family Invariants

As soon as invariants can change during delta application, it is no longer possible to reason precisely over product invariants on the level of the delta model. The reason is that invariants behave non-monotonically: if equation (5) holds for $\mathtt{i}$ it may not hold anymore for an $\mathtt{i}'$ that is logically weaker or stronger than $\mathtt{i}$.

It might seem harmless to make existing invariants stronger during delta application, that is, a δ in a delta model may introduce an invariant $\delta.\mathtt{i}$ which is conjoined to the existing invariant. This, however, requires to prove that all methods still satisfy the strengthened invariant. The problem is that at the level of the delta model we do not know which concrete deltas are going to be used to build a product. The best we can do is to approximate the required invariant for each delta δ by collecting the invariants of all previous deltas. A safe approximation is to establish the invariant $\bigwedge_{\delta' \leq \delta} \delta'.\mathtt{i}$ for each existing method (not only for the methods mentioned in the delta) as part of the verification of each δ in assumption (2a) of Thm. 1. This is more expensive than the core invariant approach outlined in Sect. 6.1, but there is still only a polynomial number of proofs in terms of the number of deltas and method calls.

The main drawback of this approach is not just the increased number of proofs, but that the invariant that can be shown on the family level might be much stronger than necessary for a specific product.

7 Proof Transformation

Our proof framework is based on a standard sequent calculus for first-order logic. Sequents are of the form $\Gamma, \Phi \Longrightarrow \Psi$ where without loss of generality all side formulas occur in $\mathit{\Gamma}$. Such a sequent is valid if for all models and program states the formula $\bigwedge_{\gamma\in\mathit{\Gamma}}\gamma\wedge\mathit{\Phi}\rightarrow\mathit{\Psi}$ holds. We assume that there are sequent rules that can reduce a program formula of the form $\langle\mathtt{q}\rangle\mathit{\Phi}$ appearing on the right-hand side of a sequent to a finite set of pure first-order verification conditions (VCs) *provided that* q *contains no method calls* (we deal with method calls in the rule given below). There are several calculi that contain such rule sets, for example, [6, Chapter 3]. Finiteness of the VCs is achieved by a loop invariant rule that approximates the behaviour of a loop with a suitable single invariant formula that holds for any execution of the loop body. In general, loop invariants cannot be found automatically and must be supplied by the user. For this reason, it is desirable to be able to reuse as much as possible of an existing verification proof when its targeted program or specification changes.

Assume we proved that each method of a core product satisfies its contract and that each delta has been verified. By soundness of our proof system, Thm. 1 ensures that each obtainable product satisfies its contract as well provided that assumption (2b) holds. By completeness of our proof system there must be a formal proof of this fact. Thm. 1 provides a *semantic* argument that such a proof must exist. In this section, we provide a justification for Thm. 1 that is based on *proof transformation.* Based on this, Thm. 1 can be proved syntactically.

This "syntactic version" of the proof of Thm. 1 must ensure that proofs can be transformed such that methods calls are replacable with calls to methods having a more specific contract. To this end, we need a concrete proof rule for applying method contracts. For simplicity, we ignore the parameters and the return value of method calls. Instead of $\mathtt{m}(\overline{\mathtt{p}})$, we simply write m. The following rule, based on the verification approach of [6], approximates a method call with the method's contract and makes it possible to verify each method separately.

$$\textsf{methodContract}\quad \frac{\mathit{\Gamma}\Longrightarrow\mathtt{m.r}\qquad \mathcal{U}_{\mathtt{m.a}}\mathit{\Gamma}\Longrightarrow\mathcal{U}_{\mathtt{m.a}}(\mathtt{m.e}\rightarrow\langle\omega\rangle\mathit{\Phi})}{\mathit{\Gamma}\Longrightarrow\langle\mathtt{m;}\omega\rangle\mathit{\Phi}}\tag{6}$$

The expression $\mathcal{U}_{\mathtt{m.a}}$ is an explicit substitution that sets all locations occuring in the assignable clause m.a to unknown locations modeled by fresh Skolem symbols. In the following, assume that the contract of $\mathtt{m}'$ is more specific than the contract of m, i.e., (3) holds (this is justified by assumption (2a) of Thm. 1). Then the validity of the following rule is easy to prove:

$$\textsf{weakenSubst}\quad \frac{\mathcal{U}_{\mathtt{m.a}}\mathit{\Gamma}\Longrightarrow\mathcal{U}_{\mathtt{m.a}}\mathit{\Phi}}{\mathcal{U}_{\mathtt{m}'.\mathtt{a}'}\mathit{\Gamma}\Longrightarrow\mathcal{U}_{\mathtt{m}'.\mathtt{a}'}\mathit{\Phi}}\tag{7}$$

Now assume an application of (6) occurs in a formal proof and we want to replace the call to method m by a call to $\mathtt{m}'$ which has a more specific contract.

We reduce a call to method $\mathtt{m}'$ to one by $\mathtt{m}$ as follows:

$$\dfrac{\dfrac{\overset{\text{premiss (6)}}{\mathit{\Gamma}\Longrightarrow\mathtt{m.r}}\qquad\overset{(3)}{\mathit{\Gamma},\mathtt{m.r}\Longrightarrow\mathtt{m}'.\mathtt{r}'}}{\mathit{\Gamma}\Longrightarrow\mathtt{m}'.\mathtt{r}'}\qquad\dfrac{\dfrac{\dfrac{\overset{(3)}{\mathtt{m}'.\mathtt{e}'\Longrightarrow\mathtt{m.e}}}{\mathcal{U}_{\mathtt{m.a}}\mathtt{m}'.\mathtt{e}'\Longrightarrow\mathcal{U}_{\mathtt{m.a}}\mathtt{m.e}}\qquad\overset{\text{premiss (6)}}{\mathcal{U}_{\mathtt{m.a}}\mathit{\Gamma}\Longrightarrow\mathcal{U}_{\mathtt{m.a}}(\mathtt{m.e}\to\langle\omega\rangle\mathit{\Phi})}}{\mathcal{U}_{\mathtt{m.a}}\mathit{\Gamma}\Longrightarrow\mathcal{U}_{\mathtt{m.a}}(\mathtt{m}'.\mathtt{e}'\to\langle\omega\rangle\mathit{\Phi})}}{\mathcal{U}_{\mathtt{m}'.\mathtt{a}'}\mathit{\Gamma}\Longrightarrow\mathcal{U}_{\mathtt{m}'.\mathtt{a}'}(\mathtt{m}'.\mathtt{e}'\to\langle\omega\rangle\mathit{\Phi})}}{\mathit{\Gamma}\Longrightarrow\langle\mathtt{m}';\omega\rangle\mathit{\Phi}} \tag{8}$$

After applying the method contract rule (6) to the replaced call with $\mathtt{m}'$, we apply a cut to the left premiss which leaves us with the left premiss of the original rule application (7) and one conjunct of (3). To the right premiss, we apply first the weakening rule (7) which is again justified by (3). Then, we weaken the premiss of the implication on the right, replacing $\mathcal{U}_{\mathtt{m.a}}\mathtt{m}'.\mathtt{e}'$ with $\mathcal{U}_{\mathtt{m.a}}\mathtt{m.e}$. The weakening is justified by first weakening again $\mathcal{U}_{\mathtt{m.a}}$ to the empty substitution with (7) and then again (3) of which all parts have now been used. We are left with the right premiss of the original rule application (7).

A proof transformation such as the one just sketched opens up the possibility of systematic proof reuse in situations where we cannot reason any longer on the family level, because constraints such as Def. 4 or (2b) in Thm. 1 are too restrictive. One idea is to make use of the syntactic structure in the specification deltas. Assume, for example, that $\mathtt{m}'.\mathtt{r}' = \textbf{original} \land \theta = \mathtt{m.r} \land \theta$ for some θ which violates (3). In this case, the proof (8) cannot be completed any longer. Nevertheless, a large part can be salvaged, only the second premiss from left is replaced with $\mathit{\Gamma}, \mathtt{m.r} \Longrightarrow \theta$ for which a new proof must be found. This approach will be developed in a follow-up paper.

8 Related Work

Behavioral subtyping is often criticized as too restrictive for practical purposes [30]. For this reason, a number of modifications have been suggested, such as incremental reasoning [31] or lazy behavioral subtyping [15]. None of these are directly applicable to DOP.

Product line analysis can be classified in three main categories [32]: first, product-based analysis considers each product variant separately. Product-based analyses can use any standard analysis technique for single products, but are in general infeasible for product lines due to the exponential number of products. Second, family-based analysis checks the complete code base of the product line in a single run to obtain a result about all possible variants. Family-based product line analyses are currently used for type checking [1, 11] and model checking [10, 2, 21] of product lines. They rely on a monolithic model of the product line which hardly scales to large and complex product lines. Third, feature-based analysis considers the building blocks of the different product variants (the deltas in DOP) in isolation to derive results on all variants. Feature-based analyses are used for compositional type checking [27] and compositional model checking of product lines [23]. The compositional verification approach presented here can be classified as feature-based, since the core and every delta are verified in isolation.

For deductive verification of behavioral product properties, a product-based analysis approach is proposed in [8]. Assuming one product variant has been fully verified, from the structure of a delta to generate another program variant, it is analyzed which proof obligations remain valid in the new product variant and need not be reestablished. While this approach does not limit the variability between two product variants, it requires to consider each of exponentially many products. Since there is no systematic link between two variants, like the Liskov-like principle employed in this paper, it is difficult to optimize proof reuse.

In [32], a combination between a feature-based and a product-based verification approach for behavioral properties is proposed where for each feature module a partial proof script for Coq is generated. These proof scripts are composed and checked for single products. In [5], feature-based proof techniques for type system soundness of language extensions are proposed where proofs for single language features are incrementally constructed. In [12] Coq proofs for the soundness of a small compiler are composed feature-wise by modeling the concept of variation points. Composition scripts have to be built by hand and it is not clear whether the technique is applicable to functional verification of general programs and properties. Our approach relies on a compositional proof principle and is truly modular for behavioral program properties.

Besides delta-oriented programming, other program modularization techniques have been applied to compositionally implement software variability, for instance, feature modules [4], aspects [19], or traits [16, 7]. Apart from some initial work regarding modular deductive verification for aspects [20] and traits [14], no compositional verification approach based on an adaptation of a Liskov principle exists which is comparable to the approach presented in this paper.

9 Discussion and Future Work

This is a theoretical and conceptual paper which constitutes the first systematic incremental specification and verification framework for diverse systems implemented in DOP. DOP is amenable to formal analysis, because its granularity is at the method-level which coincides with the best-understood contract-based approaches (JML, Spec#). Another reason is that the result of a delta application is a standard program which has an undisputed correctness semantics.

The main contribution of this paper is to provide a Liskov principle for DOP which gives rise to an efficient compositional verification approach for software families. Like in Liskov's principle for class inheritance, we also employ a number of restrictions to make it work: (i) delta application leads to type-safe products (Sect. 2.1), (ii) the contracts of subsequent deltas most become more specific (Sect. 4.2, item (2b) of Thm. 1), (iii) invariants cannot be removed, but only added (Sect. 4.2, Sect. 6.2), (iv) methods called in deltas use the contract of the first implementation of that method (Def. 4). Restriction (i) is highly desirable for DOP independently of verification [27]; (ii)–(iii) originate from Liskov's principle for OO programs: future work will consist in devicing mitigating strategies similar as in the OO world [15]. Finally, (iv) is specific to our approach.

In Sect. 7 we sketch first ideas on how to deal with restrictions (ii) and (iv) by giving a proof-theoretic justification of Thm. 1 that constitutes a syntax-driven method to systematically factor out reusable parts of proofs. This is the theoretical basis for being able to combine feature-based and product-based analysis which appears as the only practical path to pursue.

References

1. S. Apel, C. Kästner, A. Grösslinger, and C. Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering*, 17(3):251–300, 2010.
2. P. Asirelli, M. H. ter Beek, S. Gnesi, and A. Fantechi. Deontic logics for modeling behavioural variability. In *VaMoS*, pages 71–76, Essen, Germany, January 2009.
3. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
4. D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.*, 30(6), 2004.
5. D. S. Batory and E. Börger. Modularizing theorems for software product lines: The Jbook case study. *J. UCS*, 14(12):2059–2082, 2008.
6. B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2007.
7. L. Bettini, F. Damiani, and I. Schaefer. Implementing Software Product Lines using Traits. In *Proc. of Object-Oriented Programming Languages and Systems (OOPS), Track of ACM SAC*, 2010.
8. D. Bruns, V. Klebanov, and I. Schaefer. Verification of software product lines with delta-oriented slicing. In *International Conference on Formel Verification of Object-oriented Software (FoVeOOS 2010), Revised Selected Papers*, volume 6528 of *LNCS*, pages 61–75, 2011.
9. D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, and P. Y. H. Wong. Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *LNCS*, pages 417–457. Springer, 2011.
10. A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *ICSE*. IEEE, 2010.
11. B. Delaware, W. Cook, and D. Batory. A Machine-Checked Model of Safe Composition. In *FOAL*, pages 31–35. ACM, 2009.
12. B. Delaware, W. Cook, and D. Batory. Theorem Proving for Product Lines. In *OOPSLA'11*, 2011. (to appear).
13. Full ABS Modeling Framework, Mar. 2011. Deliverable 1.2 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.
14. J. Dovland, F. Damiani, E. B. Johnsen, and I. Schaefer. Verifying Traits: A Proof System for Fine-Grained Reuse. In *Workshop on Formal Techniques for Java-like Programs (FTfJP'11)*, 2011.

15. J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. *Journal of Logic and Algebraic Programming*, 79(7):578–607, 2010.
16. S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, 28(2), 2006.
17. C. Engel, A. Roth, P. H. Schmitt, and B. Weiß. Verification of modifies clauses in dynamic logic with non-rigid functions. Technical Report 2009-9, Department of Computer Science, University of Karlsruhe, 2009.
18. K. Kang, J. Lee, and P. Donohoe. Feature-Oriented Project Line Engineering. *IEEE Software*, 19(4), 2002.
19. C. Kästner, S. Apel, and D. S. Batory. A Case Study Implementing Features Using AspectJ. In *SPLC*, pages 223–232. IEEE, 2007.
20. G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE*, pages 49–58. ACM, 2005.
21. K. Lauenroth, K. Pohl, and S. Toehning. Model checking of domain artifacts in product line engineering. In *ASE*, pages 269–280, 2009.
22. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, and D. M. Zimmerman. *JML Reference Manual*, Sept. 2009. Draft revision 1.235.
23. H. Li, S. Krishnamurthi, and K. Fisler. Modular Verification of Open Features Using Three-Valued Model Checking. *Autom. Softw. Eng.*, 12(3), 2005.
24. B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.
25. B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, Oct. 1992.
26. I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Proc. of 15th Software Product Line Conference (SPLC 2010)*, Sept. 2010.
27. I. Schaefer, L. Bettini, and F. Damiani. Compositional type-checking for delta-oriented programming. In *10th International Conference on Aspect-Oriented Software Development, AOSD 2011*, pages 43–56. ACM, 2011.
28. I. Schaefer and F. Damiani. Pure Delta-oriented Programming. In *FOSD'10*, pages 49–56. ACM Press, 2010.
29. I. Schaefer and R. Hähnle. Formal methods in software product line engineering. *IEEE Computer*, 44(2):82–85, Feb. 2011.
30. N. Soundarajan and S. Fridella. Inheritance: From code reuse to reasoning reuse. In *Proceedings of the 5th International Conference on Software Reuse*, pages 206–215. IEEE Computer Society, 1998.
31. N. Soundarajan and S. Fridella. Incremental reasoning for object oriented systems. In *From Object-Orientation to Formal Methods, Essays in Memory of Ole-Johan Dahl*, volume 2635 of *LNCS*, pages 302–333. Springer, 2004.
32. T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. Proof composition for deductive verification of software product lines. In *Proc. Int'l Workshop Variability-intensive Systems Testing, Validation and Verification (VAST)*, pages 270–277. IEEE Computer Society, 2011.

Hunting Bugs Inside Web Applications[⋆]

David Hauzar and Jan Kofroň

Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics
Charles University in Prague, Czech Republic

Abstract. In recent years, focus of business world has been moved towards the Internet. Web applications provide a generous interface nonstop thus offering to malicious users a wide spectrum of possible attacks. Consequently, the security of web applications has become a crucial issue. The state-of-the-art tools for a bug discovery in languages used for a web application development, such as PHP, suffer a relatively high false-positive rate and low coverage of real errors; this is caused mainly by an unprecise modeling of dynamic features of such languages and path-insensivity of the tools. In this paper, we will demonstrate weak points of the tools and describe our novel approach to these issues. It combines path-sensitive static analysis, concrete and symbolic execution, literal analysis, taint analysis and type analysis to find vulnerabilities in PHP applications. We will show how our technique handles some of the situations where other tools fail and illustrate it with examples.

1 Introduction

Recently, as business world has moved its focus towards the Internet, a number of applications have been moved on-line, and this trend is still continuing. According to the CENSUS [17], the online retail sales in the US in 2010 reached over 160 billion US dollars. Safety and security of the web applications involved in such transactions is therefore the top priority.

A typical web application is available and operational 24/7, thus not putting any time pressure on hackers and malicious users trying to exploit security holes inside them; a quite generous interface these applications provide further widens the hackers' field. Amongst the 25 most common programming errors, those specific to web applications form a significant part of this group [5]; the examples include improper neutralization of SQL commands, cross-site request forgery, and missing authorization.

The most common programming language used at the server side is PHP [12]. Although it is currently losing a bit of market share, there is a huge number of applications written in this language that deserve attention and effort towards their security [16]. While the current, the fifth version of PHP, was released already some time ago, it was the fourth version from more than ten years ago

[⋆] This work was partially supported by Charles University Foundation grant 431011 and by the Ministry of Education of the Czech Republic (grant MSM0021620838).

that introduced objects into the language, which facilitated the development of larger projects. Despite that, however, PHP features many special attributes that make it different from common programming languages, especially as far as dynamism is concerned. The examples are inclusion of a file specified by a runtime-computed filename and the *eval* construct allowing runtime construction of code that is executed afterwards.This makes it hard or sometimes even impossible to apply the same techniques and tools for finding bugs or for correctness verification as in the case of "non-web" programming languages.

1.1 Problem statement and goals

Security issues related to web applications have a significant impact on the trustworthiness and reliability of on-line transactions and not only in business; consider, e.g., leakage of classified information from a "secured" database. A lot of attention has been paid to the development of methods and tools that would help debugging these applications or establishing their correctness in some sense, since the methods for "non-web" languages cannot be easily applied. The current state-of-the-art tools, however, still suffer from low error coverage, a relatively high false-positive rate, and often also from a weak support of language constructs, such as classes, dynamic includes, and the *eval* statement [8, 18].

In this paper, we propose a method for the identification of bugs inside web applications caused by data flow of unsanitized inputs from the user to sinks (SQL queries, URL constructions, output in general, etc.) inside web applications written in PHP. We describe our method and demonstrate benefits of our approach over those present in related tools and we illustrate our method with an example.

In Sect. 2, we describe the most problematic errors inside web applications written in PHP. In Sect. 3, we discuss the properties of the tools in this area and present the results of running one of them on an example. Sect. 4 proposes our approach to analysis of PHP source code, and demonstrates our approach on examples. Sect. 5 summarizes the paper and proposes directions for a future work.

2 Errors inside web applications

A huge number of security holes inside web applications can be grouped under one category which allows data to propagate from a user input (e.g. form fields on a web page) into database queries, URLs, JavaScript code, etc. (*sinks*) without checking if they are malicious [13]. These can be prevented by filtering user input, escaping the output, and by keeping track of the input data [13].

Filtering input is a process of preventing invalid data from entering the application. Blacklist filtering excludes malicious data, while whitelist filtering excludes all data except for that explicitly listed; thus it is distinctively safer than blacklist filtering due to the possibility of a missing item in the list. Escaping or encoding special characters that the application outputs prevents injection of malicious code or data. Keeping track of the input data involves identifying the data that the input can influence, identifying the influence points, and determining the extent of the influence.

2.1 Improper flow of data

Taint analysis can be used to find the data paths from *sources* to *sinks* in an application. Taint analysis marks data representing sources as tainted and then propagates the taint markings. Data is tainted if it can be influenced by a user and it is not sanitized.

Sources are program points through which data enters a program. An attacker can provide malicious data as a user input encoded in URL and HTTP headers, data stored in cookies, and elements of the `$_SERVER` array. It is also appropriate to track data obtained from a filesystem, session data store, and output of database queries. Although in theory this data should be safe, in practice there are security exploits giving an attacker a control over the data. It is clear that some sources represent a larger security threat than others and it is necessary not only to “taint” data but also to distinguish between different taint sources.

Sinks represent the program points (commands) where an inappropriate input can cause a security threat. Examples of the sinks include commands for sending data to a browser, sending data to a database, executing data, names of dynamically included files, opening files, and executing arbitrary system commands.

The process of sanitization is specific for each kind of sources and sinks. Moreover, the extent of sanitization depends on the level of required protection and also on the application logic. A basic level of protection can be achieved by escaping an output. Escaping data that can be manipulated by an attacker using a built-in function `htmlentities` prior to sending them to a browser prevents cross site scripting (XSS) attacks while SQL injection attacks can be prevented by escaping data before sending them to a database—e.g., by using a built-in function `mysql_real_escape_string` in case of MySQL database.

Escaping an output does not prevent sensitive information leakage, or inserting invalid data into a database, though. Consider an application that allows a user to choose from multiple topics, stores topic name in URL, and then displays messages related to the selected topic. Consider that some topics are available only for registered users. If the application does not filter the topics for non-registered users, even a non-registered user can manipulate the input and see messages available only for registered users. This vulnerability can be prevented by appropriate filtering of user input. Filtering the input should be employed whenever input data reaches a database query, dynamic include, operations that executes the input, or open a file. The user input going from a source to a sink is considered as sanitized with respect to whitelist filtering if data in a source have a finite number of possible values. It is considered as sanitized with respect to blacklist filtering if there are some restrictions over possible values in the source. Note that the range of possible values in the source and restrictions over possible values depend on the specification of a particular application and should be inspected by a developer or a security auditor. Also note that symbolic execution is necessary to determine this information.

Filtering itself does not assure the absolute security of the application. Consider an application that tracks a user name via a URL (or a hidden form field, or a cookie), reads an e-mail address from a form and then associates the e-mail with the user name in the database. The fact that the user name is tracked in the URL means that it is a part of the input, the attacker can manipulate it and change the e-mail of another user. Note that escaping or filtering the user name does not prevent this vulnerability. This kind of attacks is called semantic URL attacks, spoofed form submissions, and spoofed HTTP requests. An indicator of vulnerabilities that can lead to such attacks is updating information that is identified by data that can be manipulated by the user. In these cases, the developer should thoroughly check whether it is not a security issue.

A common fix to the vulnerabilities mentioned above is to use a session mechanism (via URL or cookie). However, even this protection can be broken by Cross-Site Request Forgeries (CSRF) attacks, session fixation, or session hijacking. These vulnerabilities are indicated by the use of data from sessions in critical commands. The fix of the example is to (1) regenerate session identifier when user logs in and to (2) generate a random token prior to requesting data from a user, store the token on a server, embed it to the URL or the form and then check whether the request contains the token.

3 State of the art

Huang et al. [9] developed an intraprocedural static analysis for PHP applications in WebSSARI tool. Xie [21] discusses the limitations of their approach, in particular that it is inter-procedural and it does not model dynamic features such as dynamic arrays, objects, dynamic variables, and dynamic includes. To identify vulnerabilities, the approach performs a taint analysis. Their approach does not allow fo a custom sanitization. Data are considered to be sanitized if they are processed with a specified sanitization function.

The approach of Xie et al. [21] uses inter-procedural analysis to find SQL injection vulnerabilities in PHP applications. They model automatic conversion of particular scalar types, uninitialized variables, simple tables, and include statements. However, they leave important parts of PHP unmodeled. In particular, they do not model references, object oriented features of PHP, and they ignore recursive function calls. To model sanitization process, the approach performs taint analysis. Sanitization can occur via calls to specified sanitization functions, casting to safe types, and a regular expression match. That means the approach keeps a database of sanitizing regular expressions.

Wasserman et al. [19, 20] use grammar-based string analysis following Minamide [11] to find a set of possible string values of a given variable at a given program point and gains this information to detect SQL injections. However, the employed analysis has an incomplete support for references and does not track type conversions.

Pixy [10] performs taint analysis of PHP programs and it provides information about the flow of tainted data using dependence graphs [3]. It uses literal analysis to resolve include statements and perform alias analysis. However, it does not model aliases between variables and members of an array. Next, Pixy lacks type inference, does not model PHP's variable-variables construct as well as variable-indices and provides only a very limited support of object oriented features. Moreover, similarly to WebSSARI it performs only simple taint analysis and does not allow for custom sanitization routines.

Balzarotti et al. [3] extended Pixy to perform the analysis of the sanitization process and thus are able to deal with a custom sanitization. They combine static and dynamic analysis techniques to verify PHP programs. They perform string analysis through language-based replacement and represent values of variables in concrete program points using finite state automata. They also track what parts of strings are tainted. Static analysis that they employ is based on Pixy and thus it has the same limitations. Moreover, the database of attack strings may not be complete. Consequently, it can miss vulnerabilities and cause false alarms.

Yu et al. [22] developed an automata-based approach for verification of string operations in PHP programs and incorporate the widening operator to tackle the problem of handling variables updated in loops. Similarly as [3], they extended Pixy to perform the analysis of the sanitization process; however, they do not employ the dynamic phase.

Biggar et al. [4] perform context sensitive, flow sensitive, interprocedural static analysis of PHP in order to gain information usable for code optimizatons in their PHP compiler. They combine alias analysis, type inference and literal analysis, model arrays, PHP's variable-variables construct, objects, references, scalar operations, casts, and weak type conversions. However, their analysis is closely tailored with their intent—to gain information usable for code optimizations. They gather information that must hold and track information that may hold only in a very limited way. In most cases, they approximate information that may hold as unknown. This is not appropriate when the intent is to explore all possible behavior of the code.

The approach of Artzi et al. [1] generates test inputs automatically, monitors web applications for crashes, and validates that the output conforms to the HTML specification. The approach utilizes symbolic execution to capture logical constraints on inputs, based on these constraints, it creates new inputs that would increase the code coverage. By running an application on concrete inputs and using PHP runtime, they avoid the problem of modeling dynamic statements of PHP, undefined semantics of PHP, and their approach is naturally path-sensitive.

To our knowledge, a path-sensitive approach to a static analysis for PHP has not been yet published. A completely path-sensitive analysis is expensive. However, there has been a lot of research done in the context of other languages, especially C language to tackle this problem. ESP [6] involves light-weight path-sensitive analysis that selectively joins or separates the contributions according to the different paths based on a heuristics that conditional tests resulting in different property-related behavior should be tracked separately, while other branches should be merged. In the context of taint analysis, property-related behavior would be given by taint statuses of variables. Unfortunately, this heuristic sometimes fails. Dhurjati et al. [7] tackle this problem by iteratively adjusting the merge criterion with new path predicates that are selected using several heuristics. Balakrishnan et al. [2] improve path-insensitive analysis to obtain the effects of path-sensitive analyses by a detection of semantically unfeasible paths using path-insensitive abstract interpreter and performing a sequence of backward and forward runs. Next, they use a technique of syntactic language refinement to exclude semantically unfeasible paths from a program during static analysis. Snelting et al. [14] use program slicing and constraint solving to construct and analyze path conditions—conditions that are defined on program's input variables and must hold for information flow between two program points. Their approach is not complete, the solution of the conditions that they construct can be false witness. That is, it may not lead to intended information flow. Taghdiri et al. [15] tackle this problem by employing counterexample-guided abstraction refinement (CEGAR). They recognize false witness by executing them and monitoring their executions, and eliminating them by automatically refining path conditions in an iterative way.

3.1 Demonstrating existing tool on examples

In this section, we show the limits and weak points of the Pixy tool [10] on a few PHP code fragments. We decided to demonstrate just the Pixy tool, since its analysis engine represents, to the best of our knowledge, the best analysis engine available for finding vulnerabilities in a PHP code. The Stranger [22] tool employs more sophisticated techniques such as a string analysis and thus provides more information for vulnerabilities detection; however it is built on the same analysis engine and shares the same limitations.

Consider an example in Fig. 1. Due to the fact that Pixy does not model array aliasing correctly, a possible XSS attack is reported at line 6. Another issue connected with arrays is that it does not handle variable indices. A different source of false-positives is path-insensitivity; the `$name` variable is sanitized by the routine `htmlspecialchars` in all cases. However, Pixy reports a possible XSS attacks at lines 16 and 17. The last type of a code fragment that causes a false positive alarm that we present here starts at line 19. There, a file named "*included.php*" is included (note that the body of the while cycle is not executed at all, since the string `$filename` does not contain the "`..`" substring). As Pixy omits modeling of the strings at this point, it is not able to evaluate the `$filename` value, ignores the `include` statement and reports an error at line 26 regardless of the content of the included file; this can cause a false positive, as well as a false negative alarm.

```
1   $users[1] = 'fred';
2   $users[2] = $_GET['from_user'];
3
4   $t_users = & $users;
5   // Pixy reports the XSS vulnerability
6   echo $t_users[1];
7
8   if ($tainted) {
9           $name = $_GET['name'];
10          $index = 2;
11  } else {
12          $name = 'bob';
13          $index = 1;
14  }
15  // Pixy reports the XSS vulnerability here
16  echo $tainted ? htmlspecialchars($name) : $name;
17  echo $tainted ? htmlspecialchars($users[$index]) : $users[$index];
18
19  $ext = ".php";
20  $filename = 'included' . $ext;
21  while (strpos($filename, '..')) {
22          $filename = preg_replace('..', '.', $filename);
23  }
24  include($filename);
25  // Pixy reports the XSS vulnerability
26  echo $users[2];
27
28  // Pixy misses the XSS vulnerability here
29  printFirstIndex('tainted', $users[1], $users[2]);
30  function printFirstIndex($varName, $untainted, $tainted) {
31          echo $$varName;
32  }
33
34  $user_ids = 2;
35  // because $user_ids is scalar, the following line does nothing
36  $user_ids[2] = $_GET['user_id'];
37  // Pixy reports the XSS vulnerability here
38  echo $user_ids[2];
```

Fig. 1. Dynamic features of PHP causing false alarms and missed vulnerabilities in Pixy tool.

Besides false positives, Pixy also reports several cases in the false-negative manner. The first PHP construct that is not correctly handled by Pixy is a *variable-variables* construct, represented by the `$$varName` at line 31. Another type of false negative stems from insufficient modeling of the type system. The fragment starting at line 34 demonstrates this issue.

The last limitation of Pixy we mention here is that Pixy does not model attributes of objects. So, according to the use of the objects, both false negative and false positive alarms can arise.

At the end of Sect. 4, we demonstrate how these situations are handled when following the approach proposed in this paper.

4 Our approach

In Sect. 2 we claim that most of security errors inside web applications can be prevented by sanitizing data paths from sources of untrusted data to critical commands—sinks. Our approach is to provide the developer with sufficient information so that she/he can assure a correct sanitization. In our case, this means employing an analysis that computes data flow information using *dependence graphs* [3], identifies sources of sensitive data, sinks, and at each program point tracks:

- the taint and the sanitization status for each variable at this program point,
- the set of possible values of each variable at this program point,
- the set of conditions defined on the program's variables that must hold at this program point, and
- the set of possible types of each variable at this program point.

In this section, we describe how we gain this information, how we use it to detect vulnerabilities, and demonstrate it on examples.

4.1 Outline

The main challenge of the analysis is the combination of an arbitrary user input and the dynamism of PHP. To address this problem, we propose the analysis that consists of the following steps:

1. Construction of the control-flow graph (CFG).
2. Static analysis of constructed CFG.
3. Detection of vulnerabilities.
4. A path-sensitive validation of vulnerabilities.

Our analysis is a combination of ideas from previous work. We face the problem of construction of CFG in presence of dynamic statements such as dynamic includes, *eval* statements, and polymorphic calls the same way as Pixy [10] does. That is, we first resolve only such dynamic statements that are directly given by literals and leave the remaining statements unresolved. Then, we gain information about possible values of variables, types, and aliases using static analysis of this CFG. We use this information to construct more precise CFG that is analyzed again. We repeat this process as long as new dynamic statements are resolved.

We extended modeling of PHP's data structures introduced in Biggar [4] by introducing certainity information to every edge of the points-to graph. Certainity tracks the fact whether the given information at given program point holds regardless of the path from the entry point of the application to this program point, and in that case the information is marked as *certain*, or only if the executing goes with given path, and in that case it is marked *uncertain*.

Our static analysis stems from the analysis introduced in Biggar [4]. We adapted it to track certainity information and taint information. We adapted literal analysis introduced in Biggar [4] to propagate also symbolic values through built-in operations.

4.2 Modeling of PHP data structures

Correct modeling PHP data structures constitutes the basis of the analysis. We use a points-to graph similar to the one introduced in [4] to model variables, array indices, and object fields. The points-to graph contains three types of nodes. A *storage node* represents a symbol-table, an array, or an object. An *index node* represents a variable, an array index, or an object field. Each index node is a child of a single storage node. Next, a *value node* represents a scalar value and it is a child of a single index node. The points-to graph contains directed edges from a storage node to each index node that belongs to the storage node and directed edges from each index node to the value or the storage nodes representing possible values of the index node. Finally, in the case of aliases, there is a reference edge between two storage or two index nodes.

Note, that if we modeled aliasing by multiple edges pointing to the same value, then each assignment via an uncertain reference would have to be a weak update. Using reference edges allows the node being defined to perform a strong update and perform weak updates on its uncertain aliases. A strong update means that an assignment completely overwrites a reference relation or a previous value of a given variable; weak update means that the old values of the variable have to be preserved. It occurs if the information about the target of the assignment is uncertain. That is, the assignment has more possible targets.

There is one storage node for each array or object, one storage node for each class holding static fields of the class, one storage node for each called function local symbol table, and one additional storage node for the global symbol table.

Uncertainty is captured in this model as follows: each edge has the certainty information—it represents either certain or uncertain information. Next, each storage node contains an *unknown* field representing index nodes that have statically unknown indices, e.g., $\$a[\$dyn]$ or $\$\$dyn$ where the value of the variable $\$dyn$ is statically unknown. The analysis propagates the uncertainty information through assignments and performs strong updates when possible.

The same way as in [4], the value of an uninitialized node takes its value from the *unknown* field of the appropriate storage node. If the node does not exist, the uninitialized value is set to `null`.

4.3 Static analysis

Our static analysis stems from context sensitive, control-flow sensitive interprocedural, path-insensitive static analysis introduced in Biggar [4]. The same way as in Biggar [4], for each program variable and each program point, we track information about its aliases, literal values, types. Additionally, we track taint and sanitization status, the set of conditions over variables that must hold in this program point, and the certainty of this information. At joint points, the combining operation for both literal values of variables and types is union; if some information is not present in all branches, it is uncertain.

The same way as Biggar [4] our analysis performs symbolic execution of a PHP program. Information about possible values of literals is tracked by propagating literal values through built-in operations For each operation, we use two versions of instructions—explicit and symbolic. If all inputs of an operation are concrete, an explicit version of the operation is used, if some input of the operation is symbolic, the symbolic version of the operation is used. By using concrete operations, we reduce the imprecision caused by modeling of such operations. We face a problem of undefined semantics of PHP in the same way as in [4] and [1] by using the reference PHP implementation instead of reimplementing the concrete versions of instructions. As to modeling of the symbolic versions of instructions, we model arithmetic operations as well as operations with strings. For modeling string operations, we use automata-based approach presented in [22]. It makes it possible to handle string concatenation, string replacement, and string restriction. We use string restriction to restrict the value of a string variable based on the branch condition.

Information about the taint status is propagated through built-in operations in the following way. We use different taint markings for different sources of data.

Contrary to other approaches, we do not remove taint status after processing data with any operation. This has two reasons: (1) A correct escaping operation is identified not only by the source of data but also by the sink—the critical command in which data is used. (2) We use the taint information to detect data that can be manipulated by the user. Then, we use this information to detect additional vulnerabilites to those caused by improper escaping. Instead of removing the taint status, we track also the sanitization status. Thus, for each sanitization routine and for each taint marking, we track whether data with the taint marking are sanitized using the sanitization routine. We propagate taint and sanitization statuses built-in operations as follows: the taint status is propagated through an operation if there exists an input of the operation that is tainted, sanitization status is propagated if all inputs of an operation that are tainted are sanitized.

4.4 Identifying vulnerabilities

After CFG is constructed and static analysis is completed, we use CFG and alias information to construct the static single assignment form (SSA) of the program. Then, we infer the set of conditions that must hold at each program point using conditional statements (e.g., if and while statements). So, at the start of a positive branch corresponding to a given conditional statement, the condition corresponding to this statement is added. And at the start of a negative branch, negation of this condition is added. At the joint point corresponding to the conditional statement, the condition is removed.

Now, we have all the data necessary to identify potential vulnerabilities in the analyzed application. We divide the vulnerabilities into several categories and introduce filters to display security warnings of selected categories only.

To identify vulnerabilities, we have to find all critical commands that input suspicious data. That is, tainted data—those which can be influenced by the user of the web application and *null* values, which can arise due to the bugs in filtering. Next, we analyze the taint and sanitization statuses of this data and identify those that are not properly escaped. For each taint marking and critical command there is a list of escaping operations. The list is stored in the configuration of the program and can be configured by a developper.

We handle custom sanitization routines in the following way. First, by employing literal analysis we are able to prove the absence of given attack patterns in the same way as in [22]. Then, we track sanitization status also for string replacement operations. These operations are potential sanitizers. When the analysis detects a vulnerability, it supplies a list of such potential sanitizers to the developer. An inspection of potential sanitizers can help a developer to reveal false alarms.

Next, we mark all data that are used in critical commands and are tainted or contain *null* values as potentially not filtered. We mark the critical commands that update data identified (e.g., the *where* clause in SQL queries) by tainted data with the highest importance and the critical commands that use tainted data in other way with a lower importance. The developer can then analyze a filtering status of such data by inspecting their possible values. We also make it possible to inspect the restrictions on data by showing the conditions that must hold at a given program point. In this way, the developer can discover errors in the blacklist filtering.

Next category of potential vulnerabilites consists of those that can make the CSRF attacks possible. We identify the critical commands that update data and use data related to the current session and are not guarded by any condition comparing the token in the request with some data stored at the server, e.g., using a session mechanism.

Finally, for vulnerabilities that are caused by the flow of data from a source to a sink through some data path, we construct dependence graphs using the technique of slicing. We reduce this graph so that it contains only those parts that are relevant for the checked vulnerability. Dependence graphs can be manually inspected by the developer and used to identify vulnerable influence points in the data flow and thus to reveal the cause of the detected vulnerability.

4.5 Path-sensitivity

The analysis described in the previous section is path-insensitive. This is one of the reasons why the vulnerabilities reported by the analysis may not be real (false positives). Consequently, all paths leading to a given vulnerability can be unfeasible. To deal with this problem, we use certainty information gained during the analysis to identify vulnerabilities that do not depend on path-sensitivity. We report these vulnarebilities immediately together with the reduced dependency graph to the user. Next, for each vulnerability that is uncertain we try to prove the unfeasibility of paths leading to the vulnerability and report only the vulnerabilities corresponding to paths that were not proven to be unfeasible, again, together with the reduced dependency graphs.

To prove the unfeasibility of paths leading to a vulnerability, we identify program points that contribute to the uncertainty of the vulnerability. These program points correspond to (1) join points of branching statements where some branches do not lead to the vulnerability or causes of the vulnerability is different and to (2) access to data that cannot be certainly identified and that can lead to the vulnerability. An example of the case (1) is at line 14 of Figure 1. The question whether the variable $\$name$ is tainted remains uncertain and it depends on the condition $\$tainted = true$. An example of the case (2) is in the line 17 of Figure 1. Again, the question whether the variable $\$users[\$index]$ is tainted remains uncertain and it depends on the same condition.

We collect conditions that must hold in order for these program points lead to a vulnerability and conditions that must hold to reach the critical command corresponding to the vulnerability. Then we use a SMT solver to prove the conjunction of these conditions. If the SMT solver finds a contradiction in these conditions, the vulnerability is unfeasible. If the SMT solver proves the conditions, the vulnerability can be still unfeasible, because of dependencies between variables in the conditions. Using the information from the dependence graph, the information about the solution of the conditions, and symbolic execution, we try to find these dependencies and add the conditions corresponding to these dependencies.

Note, that we do not model all operations precisely, hence the approach cannot be complete. Therefore, false positives can still appear even after the path-sensitive analysis.

4.6 Demonstration of our approach—Evaluation

We demonstrate our approach on two examples. First, we show that our approach is capable of handling the code fragments in Fig. 1 correctly. These code fragments contain dynamic statements. Then, we show how our approach can detect more complex vulnerabilities present in the code in Fig. 2.

Handling dynamic features. Using the code fragments in Fig. 1 we show, how our approach models PHP data structures, references, operations, and how it resolves dynamic statements. We also show how it is capable of proving the unfeasibility of a vulnerability detected by the path-insensitive step of the analysis.

In Fig. 1, the statement at line 1 creates an index node representing the variable $\$users$ and makes it a child of the storage node that represents the global symbol table. Because it is the first use of this variable, the presence of the square brackets means that the variable is of the array type. Hence, the statement creates a storage node representing the array and a directed edge from the index node representing the variable $\$users$ to this storage node. The statement also creates an index node representing the index 1 in the array, an edge from the node representing the array to the value node, a value node representing the literal $'fred'$ and an edge between the index node and the value node. The statement at line 2 creates another index node representing the index 2 and a value node, appropriate edges and associates a taint status with the value node. Next, the statement at line 4 creates an index node representing the variable t_users, makes it a child of the storage node that represents the global symbol table and then creates a reference edge between this node and the index node representing the variable $\$users$. This reference is then used at line 6 to find out the appropriate storage node. The index node representing the index 1 in this storage node is not tainted, thus no vulnerability is reported—contrary to Pixy that reports a false alarm.

In the same way as Pixy, path-insensitive phase of our approach detects a potential vulnerability at lines 16 and 17. However, the information about the taint status is uncertain in both cases. The condition that must hold to make the variable $name$ tainted is $tainted = true$, the condition that must hold to reach the appropriate critical command is $tainted = false$. The conjunction of these conditions is unfeasible, thus no vulnerability is reported. Potential vulnerability at line 17 is filtered out in a similar way.

All inputs to the operation of concatenation at line 20 are concrete. Hence the analysis can use the concrete version of the instruction which results in a concrete, precise value. Similarly, the analysis uses the concrete version in the case of the operation *strpos* at line 21. Consequently, the value of the variable $\$filename$ is known at line 24, the dynamic include can be resolved. Then, more precise CFG is constructed and the static analysis step is performed with this refined CFG. Note that because we model the important operations symbolically, our analysis is able to handle even more complex cases where inputs of operations are not concrete.

The analysis correctly handles a variable-variables construct at line 31. The literal analysis determines that the value of variable $\$varName$ is $'tainted'$; the index node corresponding to this value is then searched in the storage node corresponding to the local symbol table of the function and then also in the storage node corresponding to the global symbol table. Here, the node corresponding to the variable $\$tainted$ is found in the former one and a vulnerability is reported.

The analysis performs type inference, hence, it is known that the variable $\$user\_ids$ is scalar at line 36, and does not perform the assignment. Clearly, this statement is likely a bug, so the analysis reports a warning in such cases.

Discovering more complex vulnerabilities. Now, we demonstrate how our approach handles the code in Fig. 2 and how it helps to find vulnerabilities within it. The presented code can be a part of an application that provides an interface for viewing messages of selected topics. Every user has associated one topic that she/he is not allowed to view and has to pay if she/he wants to change this topic. The user can also update her/his email address and under certain circumstances, tracked in the session, insert a message. We will describe all steps of the analysis:

(1) Construction of CFG and static analysis

First, the analysis constructs CFG of the analyzed program. Here, it encounters a problem at line 16. The method to be called depends on the type of the object and it is not known yet. Hence, this call is ignored and since it is the last statement of the "main" part, the construction of CFG is complete. Constructed CFG contains the nodes corresponding to the switch statement and the calls to constructors.

Next, the static analysis phase is performed on constructed CFG. This analysis determines a set of possible types of the object $\$action$ at line 16 and thus the set of possible methods that could be called at this line. Using this information, the analysis constructs new CFG. The CFG will contain new branch for each possible type of the object $\$action$. The first node in each branch will be a call to the method *exec_action*. In each branch the type of the object $\$action$ is fixed and so the analysis is capable to determine the method to be called (similarly to VMT). The method *exec_action* contains a dynamic include at line 22. From the first pass of the static analysis phase, a set of possible values of the variable $\$\_GET['action']$ is known. The set equals to $\{'view\_message'$, $'update\_topic'$, $'update\_email'$, $'insert\_message'$, $*\}$. Note that $*$ represents an arbitrary value. Moreover, because there is no additional node in the new CFG that manipulates this variable, this information would not be refined by performing the static analysis phase on new CFG. Clearly, the precise value of the variable $\$\_GET['action']$ is tied with the type of the object that calls the method *exec_action* in the *switch* statement; however this information is lost at the joint point of the switch statement. There are several options how to proceed: (1) The analysis can employ a path-sensitive step and try to prove unfeasibility of the values in the set. Alternatively, (2) it can resolve the include statement for all values in the set. Finally, (3) it can keep the include unresolved and report a warning to the developer. In the former two cases, the value $*$ corresponds to a vulnerability. It is uncertain, thus it will be validated in the path-sensitive step. Note that if files corresponding to infeasible values would be included and if there would be any vulnerabilites in the included files, the analysis would employ the path-sensitive phase for these vulnerabilities and prove unfeasibility of these values on demand. Also note that such an indirect relation between the type of the object and the value of a global variable not only complicates the analysis but also readability of the code and should be avoided in the first place. We will investigate all these strategies and possibly make the strategy selection configurable. Then, the version of the method *do_action* corresponding to the type of the object *action* is processed. Note, that the exact type of the object $\$action$ is known. Now, there is no dynamic statement present and hence the construction of CFG is straightforward.

(2) Identifying vulnerabilities

Based on the information gained from the static analysis step, the analysis detects vulnerabilities and divides them into several categories. The first category of vulnerabilities is formed by unescaped data in critical commands. The analysis detects that the variable $\$topic$ used in the critical command *mysql_query* has the taint status from the $\$\_GET$ array and has not sanitization status necessary for this command—*mysql_real_escape_string*. This information is certain, thus this vulnerability is reported to the user together with reduced dependence graph showing the propagation of the taint status. Next, the variable $\$message$ used in the critical command *echo* has the same taint status and again has not sanitization status necessary for this command—*htmlentities* in this case. However, the taint status is uncertain, thus this vulnerability is passed to the path-sensitive phase of the analysis.

The second category of vulnerabilities is formed by those that are present due to improper (custom) sanitization routines. We detect that the tainted variable $\$new\_email$ used in the critical command *mysql_query* at line 66 is not sanitized with any standard sanitizing function. However, it has the sanitization status *preg_replace*. Moreover, the analysis models this function and the intersection of the symbolical value of the variable with the attack patterns is empty. Because the list of the attack patterns is not complete, this does not imply that data are properly sanitized. However, the importance of this potential vulnerability is relatively low, hence, only a warning pointing to the *preg_replace* function is reported.

Another category is formed by the vulnerabilities that are caused by an unfiltered input of critical commands that update data. The variable $\$\_GET['user']$

in the critical command *mysql_query* at line 54 is not filtered, thus a vulnerability is reported. Note that updating data of an arbitrary user is suspicious, however, it could be intended. Next, a vulnerability is reported due to a usage of the variable $\$ban\_topic$ in the same command. At first sight, this variable seems to be properly filtered, however, there is the *null* value present in the set of its possible values. The absence of the default branch in the *switch* statement starting at line 46 causes the variable $\$ban\_topic$ to be uninitialized and thus produces the *null* value that is then inserted into the database. Note that this enables a user to view messages of an arbitrary topic—see the command that selects messages from the database at line 34. Note that this vulnerability is uncertain and it is thus passed to the path sensitive phase.

The next category of potential vulnerabilities is the use of unfiltered data in other (not data-updating) critical commands. The variable $\$topic$ in the critical command that selects data from the database at line 34 is not filtered and thus reported. Note that in this case, the filtering is done via the condition $message.topic \,!= users.ban\_topic$ in the database query; this report is therefore a false alarm. Next, the variable $\$user$ in the same command is not filtered. This is a real vulnerability, because it enables an attacker to view messages as a different user. Note that the variable $\$importance$ in the same command is filtered, and not reported. All these vulnerabilities are certain, and, therefore, immediately reported. Finally, the variable $\$message$ used in the *echo* command at line 89 is also detected not to be filtered. However, in this case the information is uncertain and thus passed to the path-sensitive phase.

The last category consists of vulnerabilities that make CSRF attacks possible. A CSRF vulnerability can arise when a critical command that updates some data uses data tainted with a session status and this command is not guarded by any condition comparing the token in the request with the token stored at the server. This is the case of the command *mysql_query* at line 65, thus a vulnerability is reported.

(3) Path-sensitive validation of vulnerabilities

Uncertain vulnerability corresponds to use of unescaped and unfiltered data in the critical command at line 89. At this program point, the condition $\$insert = false$ holds. We conjoin this condition with conditions gained using program points that contribute to the uncertainty of the vulnerability. The only such program point is the joint point of the *switch* statement starting at line 80. To expose the vulnerability, the variable $\$message$ must be tainted. It gains an uncertain taint status at this program point. Therefore, it is tainted only if the control flow goes with the first or second branch. This corresponds to the condition $(\$message = 1 \vee \$message = 2)$. The conjunction of these two conditions is passed to the SMT solver. Since it is satisfiable, the unfeasibility of the vulnerability is not detected. However, there is a dependency between the variable $\$user\_status$ and the variable $\$insert$ captured using the *if* statement starting at line 76. This dependency corresponds to the condition:

$$((\$user\_status = 1 \vee user\_status = 2) \Rightarrow \$insert = true) \wedge$$
$$((\$user\_status \neq 1 \wedge user\_status \neq 2) \Rightarrow \$insert = false)$$

This condition is conjoined with the previous (conjoined) condition and again passed to the SMT solver. Now the SMT solver proves the unfeasibility[1] of the vulnerability.

The next vulnerability which is uncertain is the vulnerability corresponding to the presence of the *null* value in the set of values of the variable $\$ban\_topic$ at the line 54. The critical command is not guarded by any condition. The variable keeps the value *null*, if the condition $(\$\_GET['ban\_topic'] \neq' world' \wedge \$\_GET['ban\_topic'] \neq' science')$ holds. This condition is satisfiable and there are no data dependencies between any variables in the condition and thus the vulnerability is feasible and it is passed to the user together with the solution of the condition.

```
 1  <?php
 2  switch ($_GET['action']) {
 3    case 'view_message':
 4      $action = new ViewMessageAction();
 5      break;
 6    case 'update_topic':
 7      $action = new UpdateBannedTopicAction();
 8      break;
 9    case 'update_email':
10      $action = new UpdateEmailAction();
11      break;
12    case 'insert_message':
13      $action = new InsertMessageAction();
14      break;
15  }
16  $action−>exec_action();
17
18  abstract class Action {
19    abstract protected function do_action();
20    public function exec_action() {
21      include $_GET['action'] . ".inc";
22      $this−>do_action();
23    }
24  }
25
26  class ViewMessageAction extends Action {
27    protected function do_action() {
28      $topic = $_GET['topic'];
29      $user = $_GET['user'];
30      $importance = $_GET['importance'];
31      if ($importance < MIN_IMPORTANCE || $importance >=
              MAX_IMPORTANCE) {
32        exit();
33      }
34      $result = mysql_query( "SELECT ∗ FROM messages, users
35        WHERE messages.topic = '" . $topic . "' AND
36          messages.topic != users.ban_topic AND
37          messages.importance <= '" . $importance . "' AND
38          users.name = '" . mysql_real_escape_string($user) . "'" );
39      // a code that displays messages follows
40    }
41  }
42
43  class UpdateBannedTopicAction extends Action {
44    protected function do_action() {
45      if (payment_successfull()) {
46        switch($_GET['ban_topic']) {
47          case 'world':
48          case 'science':
49            $ban_topic = $_GET['ban_topic'];
50            break;
51        }
52        $result = mysql_query( "UPDATE users SET
53                  ban_topic = '" . $ban_topic . "'
54                WHERE user = '" . mysql_real_escape_string($_GET
                        ['user']) . "'" );
55      }
56    }
57  }
58
59  class UpdateEmailAction extends Action {
60    protected function do_action() {
61      session_start();
62      $user = $_SESSION['user'];
63      $new_email = $_GET['email'];
64      $new_emaill = preg_replace("/[ˆA−Za−z0−9.\−@]/","",
              $new_email);
65      $result = mysql_query( "UPDATE users SET
66          email = '" .$new_emaill . "'
67        WHERE user = '" . $user . "'" );
68    }
69  }
70
71  class InsertMessageAction extends Action {
72    protected function do_action() {
73      session_start();
74      $user_status = $_SESSION['user_status'];
75      if ($user_status == 1 || $user_status == 2) {
76        $insert = true;
77      } else {
78        $insert = false;
79      }
80      switch ($user_status) {
81        case 1:
82          $message = $_GET['message'];
83        case 2:
84          $message = substr(0, 15, $_GET['message']);
85        case 3:
86          $message = "You cannot insert messages.";
87      }
88      if ($insert == false) {
89        echo $message;
90        exit();
91      }
92      // a code that inserts the message follows
93    }
94  }
95  ?>
```

Fig. 2. Code fragment that contains several vulnerabilities.

[1] *user_status* has to be 1 or 2 AND *insert* has to be false which contradicts the first part of the last formula.

5 Conclusion and future work

While symbolic execution, literal analysis, taint analysis, and type analysis for PHP applications have been already developed, as far as we know, this is the first approach of combining them into a single analysis. The combination of these features is powerful and open doors for a precise analysis of dynamic languages such as PHP. The novel contribution of our approach is its path-sensitivity. Even though false alarms caused by path-insensitivity of existing tools were reported, to our best knowledge, no approach that addresses this issue has been published. Last but not least, our approach detects most of the vulnerabilities that can be present in web applications by checking whether a user input is properly filtered, output of the application is escaped, and keeping track of the input data. This is done by employing advanced taint analysis, analyzing possible literal values of variables in critical commands, and using dependence graphs to keep track of data.

We believe that the analysis is scalable, expensive path-sensitive step of the analysis is performed only when it is necessary, e.g. to confirm vulnerabilities that can be false alarms. This is possible because the path-insensitive step of the analysis tracks whether the collected information is precise or can be refined by the path-sensitive step. Moreover, the path-sensitive step can be completely disabled or performed on demand, e.g., to confirm vulnerabilities of particular category. Our approach is not complete. False positives can still appear even after the path-sensitive step. However, precise analysis combined with path-sensitive step and employed vulnerability detection promises both a lower false-positive rate and higher error coverage than related approaches.

Once a prototype implementation is completed, we will evaluate its scalability on real web applications. Future work will also investigate and evaluate existing techniques to analyze and refine path-conditions and possibly adapt them to be usable in the context of PHP applications.

References

1. S. Artzi, a. Kiezun, J. Dolby, F. Tip, D. Dig, a. Paradkar, and M. D. Ernst. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Transactions on Software Engineering*, 36(4):474–494, July 2010.
2. G. Balakrishnan, S. Sankaranarayanan, F. Ivancic, O. Wei, and A. Gupta. Slr: Path-sensitive analysis through infeasible-path detection and syntactic language refinement. In M. Alpuente and G. Vidal, editors, *Static Analysis*, volume 5079 of *Lecture Notes in Computer Science*, pages 238–254. Springer Berlin / Heidelberg, 2008.
3. D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. *2008 IEEE Symposium on Security and Privacy (SP 2008)*, pages 387–401, May 2008.
4. P. Biggar and D. Gregg. Static analysis of dynamic scripting languages, 2009.

5. Common weakness enumeration. `http://cwe.mitre.org/top25/`.
6. M. Das, S. Lerner, and M. Seigle. Esp: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 57–68, New York, NY, USA, 2002. ACM.
7. D. Dhurjati, M. Das, and Y. Yang. Path-sensitive dataflow analysis with iterative refinement. In K. Yi, editor, *Static Analysis*, volume 4134 of *Lecture Notes in Computer Science*, pages 425–442. Springer Berlin / Heidelberg, 2006.
8. J. Fonseca, M. Vieira, and H. Madeira. Testing and comparing web vulnerability scanning tools for sql injection and xss attacks. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, pages 365–372, Washington, DC, USA, 2007. IEEE Computer Society.
9. Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 40–52, New York, NY, USA, 2004. ACM.
10. N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting Web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 6 pp.–263. Ieee, 2006.
11. Y. Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 432–441, New York, NY, USA, 2005. ACM.
12. PHP—Personal Home Pages. `http://www.php.net`.
13. C. Shiflett. *Essential PHP security*. O'Reilly, 2006.
14. G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15:410–457, October 2006.
15. M. Taghdiri, G. Snelting, and C. Sinz. Information flow analysis via path condition refinement. In *Proceedings of the 7th International conference on Formal aspects of security and trust*, FAST'10, pages 65–79, Berlin, Heidelberg, 2011. Springer-Verlag.
16. Tiobe Software. TIOBE Programming Community Index for June 2011. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.
17. U.S. Census Bureau News, May 2011. `http://www.census.gov/retail/mrts/www/data/pdf/ec_current.pdf`.
18. M. Vieira, N. Antunes, and H. Madeira. Using web security scanners to detect vulnerabilities in web services. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pages 566 –571, 29 2009-july 2 2009.
19. G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 32–41, New York, NY, USA, 2007. ACM.
20. G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. *Proceedings of the 13th international conference on Software engineering - ICSE '08*, page 171, 2008.
21. Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *15th USENIX Security Symposium,*, pages 179–192, July 2006.
22. F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for PHP. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 154–157, 2010.

Juggrnaut - An Abstract JVM

Jonathan Heinen and Christina Jansen

Software Modeling and Verification Group
RWTH Aachen University, Germany
http://moves.rwth-aachen.de/

Abstract. We introduce a new kind of hypergraphs and hyperedge replacement grammars, where nodes are associated with types. We use them to adapt the abstraction framework *Juggrnaut* presented by us in [8, 9] – for the verification of Java Bytecode programs. The framework is extended to handle additional concepts needed for the analysis of Java Bytecode like null pointers and method stacks as well as local and static variables. We define the abstract transition rules for a reasonable subset of opcodes and show how to compute the abstract state space. Finally we complete the paper with some experimental results.

1 Introduction

With growing size and complexity of software – especially of security relevant software – automatic verification techniques become more and more essential. Nowadays object-oriented languages are common in many software projects. They introduce new challenges to software verification like sharing and destructive updates as the possibility to allocate new objects during runtime resulting in an infinite state space.

In [8, 9] we presented an abstraction framework to handle pointer programs that is based on the natural representation of heaps by directed labelled graphs where heap cells (or objects) are represented by nodes, pointers between objects by edges. Abstraction is achieved by replacing parts of well-known structures by a single hyperedges (as depicted in Fig. 1 for a tree with linked leafs). Thus abstract heaps are represented by hypergraphs while known heap structures (the data structures) are defined by hyperedge replacement grammars.

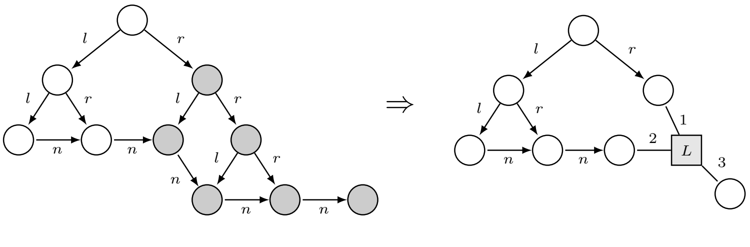


Fig. 1. Representation of abstracted heap parts by non-terminal hyperedges.

Up to now we presented a theoretical view on the *Juggrnaut*-framework. While concentrating on pointer properties like destructive updates, sharing and the dynamic allocation of heap cells we did not consider object-oriented aspects like member variables (fields), object methods, polymorphism and other type depending instructions (e.g. **instanceof**). Furthermore (recursive) method calls and local variables, potentially yielding an unbounded number of variables, were not considered so far. In this paper we extend the framework to handle a significant subset of Java Bytecode including the aspects listed above. In Section 2.1 we provide the theoretical foundations, introducing a new model of hypergraphs with typed nodes to model typed objects in common object-oriented languages (e.g. Java, C++, Objective C, . . .). The newly introduced hypergraphs enforces the adaption of hyperedge replacement grammars and of the requirements formulated in [9]. In Section 3.1 we extend the *Juggrnaut*-framework towards an abstract Java Virtual Machine (JVM). We consider the impact of Java Bytecode specific concepts like static and dynamic method calls as well as static and local variables. In this context we also define some of the transition rules of the abstract JVM. In the extended version of this paper [7] proofs omitted due to space restrictions can be found.

2 Basic Concepts

Given a set S, $S^\star$ is the set of all finite sequences (strings) over S including the empty sequence ε. For $s \in S^\star$, the length of s is denoted by $|s|$, the set of all elements of s is written as $[s]$, and by $s[i]$ we refer to the i-th element of s. Given a tuple $t = (A, B, C, \dots)$ we write A_t, B_t etc. for the components if their names are clear from the context. Function $f{\restriction}S$ is the restriction of f to S. Function $f : A \to B$ is lifted to sets $f : 2^A \to 2^B$ and to sequences $f : A^\star \to B^\star$ by point-wise application. We denote the identity function on a set S by id_S.

2.1 Hypergraphs and Heaps

In [9] heap structures are represents by hypergraphs. Hypergraphs contain edges connecting arbitrary many nodes. They are labelled from a ranked alphabet Σ with terminal and nonterminal labels. A ranking function $\mathrm{rk} : \Sigma \to \mathbb{N}$ maps each label l to a rank, defining the number of nodes an l-labelled edge connects.

Example 1. Consider the right graph of Fig. 1. Nodes (depicted as circles) represent objects on the heap. Edges are labeled from an alphabet Σ. Terminal edges (labelled by terminals) connecting two nodes represent pointers, whereas nonterminal edges (depicted as shaded boxes) represent abstracted heap parts. In Fig. 1 an L-labelled hyperedge connects three nodes, i.e. $\mathrm{rk}(L) = 3$. The order on attached nodes is depicted by numbers labelling connection lines (that we call tentacles) between edges and nodes. For terminal edges the direction induces the order.

Definition 1 (Tentacle). *A tuple* $(a, i), a \in \Sigma, i \in [1, \mathrm{rk}(a)]$ *is a* tentacle.

While objects in common object-oriented languages are of a well-defined type this is not reflected in this representation. We extend the graph model to (labelled and typed) hypergraphs over a typed alphabet assigning a type to each node.

Definition 2 (Typed Alphabets). *A* typed alphabet *is a tuple* $(L, (T, \preceq), \mathit{types})$ *with* L *a set of labels, type poset* $(T, \preceq)$ *and typing function.* $\mathit{types} : L \to T^{\star}$*.*

Note that the ranking function is now implicitly given by $\mathrm{rk}(f) = |\mathit{types}(f)|$. $\mathit{types}(X)[i]$ defines for tentacle (X, i) the type of nodes it connects.

Definition 3 (Hypergraph). *A* (labelled and typed) hypergraph (HG) *over a typed alphabet* Σ *is a tuple* $H = (V, E, \mathit{lab}, \mathit{type}, \mathit{att}, \mathit{ext})$*, with* V *a set of nodes,* E *a set of edges.* $\mathit{lab} : E \to L_{\Sigma}$ *en edge and* $\mathit{type} : V \to \Pi$ *a node labelling function. The attachment function* $\mathit{att} : E \to V^{*}$ *maps each hyperedge to a node sequence and* $\mathit{ext} \in V^{\star}$ *is a (possibly empty) sequence of pairwise distinct external nodes. For* $e \in E$ *we require that* $\mathit{types}(\mathit{lab}(e)) \succeq \mathit{type}(\mathit{att}(e))$*, i.e. any tentacle is connected to a node of its corresponding type or a subtype.*

The set of all hypergraphs over Σ *is denoted by* HG_{Σ}*.*

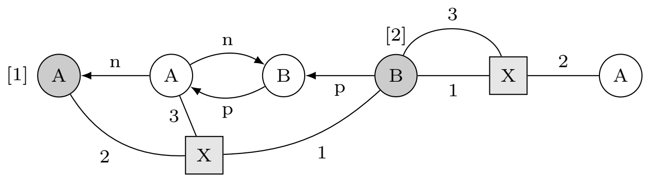


Fig. 2. A labelled and typed hypergraph.

Example 2. Fig. 2 depictes a labelled and typed hypergraph over the alphabet $\Sigma = (\{n, p, X\}, (\{A, B\}, B \preceq A), [n \mapsto AA, p \mapsto BA, X \mapsto BAA])$. Nodes are annotated with their type. The order on the (grey) external nodes is given by numbers in square brackets next to them. Edges are connected corresponding to *types*, e.g. tentacles $(X, 3)$ and $(X, 1)$ of the right X-edge are mapped to the same node. They are correctly connected as $B \preceq \mathit{types}(X)[1]$ and $B \preceq \mathit{types}(X)[3]$.

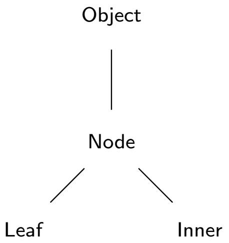
We use hypergraphs to model heaps where terminal edges represent pointers and nonterminal edges represent embedded heap structures of a well defined shape. Though the hypergraph depicted in Fig. 2 is correct according to the definition of a hypergraph it does not represent a reasonable heap. Consider e.g. the two $(X, 1)$-tentacles connected to the same node. The X-edges should represent a well-defined structure maybe inducing outgoing pointers at tentacle $(X, 1)$. This however would violate the common concept of pointers where it is impossible to have multiple outgoing pointers with the same label (e.g. the second node from left, that has two outgoing n-pointers). Every pointer on the heap has to be represented either concrete as a terminal edge or abstracted within a hyperedge. In order to formalise the requirements we introduce the notion of entrance($\triangledown$)- and reduction($\blacktriangle$)-tentacles. Nodes can be left by $\triangledown$-tentacles, i.e. they represent outgoing pointers while $\blacktriangle$-tentacles represent incoming ones only. Given set A we use: $A_{\blacktriangle} = A \times \{\blacktriangle\}$, $A_{\triangledown} = A \times \{\triangledown\}$ and $A_{\triangledown\blacktriangle} = A_{\blacktriangle} \cup A_{\triangledown}$, $a_{\blacktriangle}$ $(a_{\triangledown})$ for elements of $A_{\blacktriangle}(A_{\triangledown})$. Functions as relations over A are defined over $A_{\blacktriangle\triangledown}$ implicitly.

Definition 4 (Heap Alphabet). Heap alphabet $\Sigma_N = (\mathbb{F} \cup N, (\mathbb{T}, \preceq), \mathit{types})$ *is a tuple with a set of field labels* $\mathbb{F}$*, a set of nonterminals* N *and* $\mathbb{T}$ *a set of types. Typing function* $\mathit{types} : \mathbb{F} \cup N \to \mathbb{T}_{\triangledown\blacktriangle}{}^{\star}$*, restricted for fields:* $\mathit{types}{\restriction}\mathbb{F} : \mathbb{F} \to \mathbb{T}_{\triangledown}\mathbb{T}_{\blacktriangle}$.

For Java programs the *heap alphabet* is given by the class-definitions. $\mathbb{T}$ corresponds to the classes with subtype relation $\preceq$, while $\mathbb{F}$ are the field names. Function *types* maps fields to their defining class (as $\triangledown$-tentacle) and to the class they points to (as $\blacktriangle$-tentacle), i.e. given the class definition **class** A{ B f;} $\mathit{types}(f) = A_{\triangledown}B_{\blacktriangle}$. For $t \in \mathbb{T}$ we define $\mathit{fields}(t) = \{f \in \mathbb{F} \mid \mathit{types}(f)[1] \succeq t_{\triangledown}\}$.

```
class Node{
  Inner parent;
}
class Inner extends Node{
  Node left, right;
}
class Leaf extends Node{
  Leaf next;
}
```

(a) Class definition



(b) Type poset

Fig. 3. Class definition and resulting poset.

Example 3. Given the Java-class definitions from Fig. 3(a) we get the set of types $\mathbb{T} = \{\text{Object}, \text{Node}, \text{Inner}, \text{Leaf}\}$ and as terminal edge labels the field-names $\mathbb{F} = \{\mathit{Node.parent}, \mathit{Inner.left}, \mathit{Inner.right}, \mathit{Leaf.next}\}$. The poset $(\mathbb{T}, \preceq)$ defined by the subtype relation is given in Fig. 3(b). The type sequence for parent is $\mathit{types}(\mathit{Node.parent}) = \text{Node}_{\triangledown}\text{Inner}_{\blacktriangle}$, for left and right $\mathit{types}(\mathit{Inner.left}) = \mathit{types}(\mathit{Inner.right}) = \text{Inner}_{\triangledown}\text{Node}_{\blacktriangle}$ and $\mathit{types}(\mathit{Leaf.next}) = \text{Leaf}_{\triangledown}\text{Leaf}_{\blacktriangle}$. And the resulting function *fields* as $\mathit{fields}(\text{Node}) = \{\text{Node.parent}\}$, $\mathit{fields}(\text{Inner}) = \{\text{Inner.left}, \text{Inner.right}, \text{Node.parent}\}$, $\mathit{fields}(\text{Leaf}) = \{\text{Leaf.next}, \text{Node.parent}\}$.

By $\triangledown_H(v) = \{e \in E_H \mid (\exists i \in \mathbb{N} : \mathit{types}(\mathit{lab}(e))[i] \in \mathbb{T}_{\triangledown} \wedge \mathit{att}(e)[i] = v\}$ we define the set of edges connected to the node $v \in V_H$ through an entrance tentacle.

Definition 5 (Heap Configuration). *A* heap configuration (HC) *over a heap alphabet* Σ_N *is a tuple* $H = (V, E, \mathit{lab}, \mathit{type}, \mathit{att}, \mathit{ext})$*, where* V *is a set of nodes,* E *a set of edges.* $\mathit{lab} : E \to \mathbb{F} \cup N$ *is the edge and* $\mathit{type} : V \to \mathbb{T}$ *the node labelling function.* $\mathit{att} : E \to V^*$ *maps each hyperedge to a sequence of attached nodes and* $\mathit{ext} \in V^{\star}_{\blacktriangle\triangledown}$ *is a (possibly empty) sequence of pairwise distinct external vertices.*

For terminal edge $e \in E, \mathit{lab}(e) \in \mathbb{F}$ *we require that* $\mathit{types}(\mathit{lab}(e)) \succeq \mathit{type}(\mathit{att}(e))$ *while for* $e \in E, \mathit{lab}(e) \in N$ *we require that* $\mathit{types}(\mathit{lab}(e)) \succeq_N \mathit{type}(\mathit{att}(e))$*, where* $t_{\blacktriangle} \succeq_N t'_{\blacktriangle}$ *iff* $t \succeq t'$ *and* $t_{\triangledown} \succeq_N t'_{\triangledown}$ *iff* $t = t'$.

For $v_{\blacktriangle} \in \mathit{ext}$ *we require* $\triangledown(v) = \emptyset$*, while for* $v \in V : v_{\blacktriangle} \notin \mathit{ext}$ *we require:*

$$\mathit{lab}(\triangledown(v)) = \mathit{fields}(\mathit{type}(v)) \;\wedge\; x, y \in \triangledown(v) \Rightarrow \mathit{lab}(x) \neq \mathit{lab}(y) \tag{1}$$

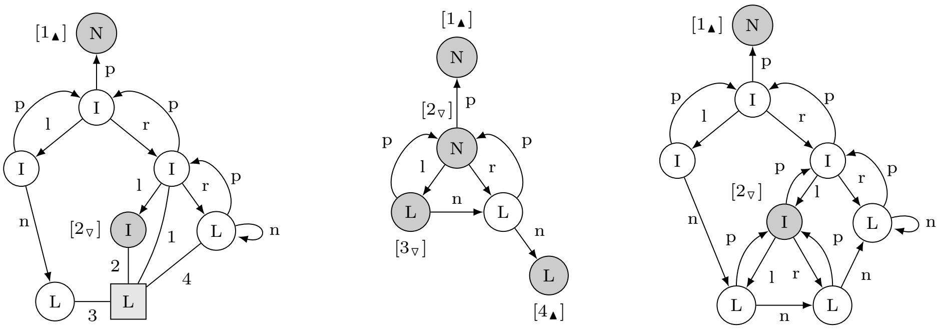
$$\vee\; \mathit{lab}(\triangledown(v)) \subseteq N \wedge \mid \triangledown(v) \mid = 1 \tag{2}$$

The set of all heap configurations over Σ_N *is denoted by* HC_{Σ_N}.

While terminal $\triangledown$-tentacles represent single outgoing pointers, nonterminal $\triangledown$-tentacles represent all outgoing pointers of a node. Therefore nodes are either connected to all terminal $\triangledown$-tentacle define by the type (1) or a single nonterminal $\triangledown$-tentacle (2).

External nodes are considered to be outside the graph and their outgoing pointers a either all outside the graph and the external node is annotated as reduction-node ($\blacktriangle$) or are all inside the graph and the external-node is therefore an entrance node ($\triangledown$) as we can enter the graph from this external node.

Example 4. In Fig. 4(a) a HC for the heap alphabet from Ex. 3 extended by the nonterminal L with $types(L) = I_{\blacktriangle} I_{\triangledown} L_{\triangledown} L_{\blacktriangle}$ is given. Here I is the short form for Inner, L for Leaf. Nonterminal edges labelled by L represent trees with linked leaves. The external nodes are: the root node (2) of a subtree, the parent of the root (1), left most leaf (3) and the n-reference(4) of the rightmost leaf of the tree. The labels for external nodes are extended by an index to mark them as $\triangledown$- or $\blacktriangle$-nodes. As external node 1 is a $\blacktriangle$-node it has no outgoing edges, while external node 2 has abstracted outgoing edges represented by the $\triangledown$-tentacle $(L, 2)$.



(a) Heap configuration (b) Replacement Graph (c) Result

Fig. 4. An abstract heap configuration.

We use Σ to denote a heap alphabet Σ_N without nonterminals, i.e. $N = \emptyset$. If a HC does not contain nonterminal edges, i.e. is defined over a heap alphabet Σ we call it concrete ($H \in \mathrm{HC}_{\Sigma}$), otherwise abstract ($H \in \mathrm{HC}_{\Sigma_N}$).

2.2 Data Structure Grammars

Hyperedge replacement grammars can be used to describe heap structures. These grammars are defined as a set of rules each consisting of a nonterminal on the left hand side and a hypergraph on the right hand side.

Definition 6 (Hyperedge Replacement Grammar (HRG)). *A hyperedge replacement grammar (HRG) over a typed alphabet Σ_N is a set of production rules $p = X \to H$, with $X \in N$ and $H \in HG_{\Sigma_N}$, where $types(X) \preceq type(ext_H)$.*

We denote the set of all hyperedge replacement grammars over Σ_N by HRG_{Σ_N}.

Derivation steps of a HRG are defined by hyperedge replacement. A hyperedge e is replaced by a hypergraph by mapping external nodes of the latter with attached notes of e. The replacement is possible iff amount and types of nodes connected by e and of the external nodes in the replacement graph correspond to each other. This aspect is covered in the following adaption of the definition from [9] for labelled and typed hypergraphs.

Definition 7 (Hyperedge Replacement). *Given hypergraphs* $H, I \in \mathrm{HG}_{\Sigma_N}$ *and an edge* $e \in E_H$*, with* $\mathit{type}(\mathit{ext}_I) \preceq \mathit{type}(\mathit{att}_G(e))$ *the replacement of edge* e *in* G *by* I *is defined as* $K = H[I/e] = (V_K, E_K, \mathit{lab}_K, \mathit{type}_K, \mathit{att}_K, \mathit{ext}_K)$*:*

$$\begin{array}{llll}
V_K & = V_H \uplus (V_I \setminus \mathit{ext}_I) & E_K & = (E_H \setminus \{e\}) \uplus E_I \\
\mathit{type}_K & = \mathit{type}_H{\restriction}V_K \uplus \mathit{type}_G & \mathit{lab}_K & = \mathit{lab}_H{\restriction}E_K \uplus \mathit{lab}_I \\
\mathit{ext}_K & = \mathit{ext}_H & & \\
\mathit{att}_K & \multicolumn{3}{l}{= \mathit{att}_H \uplus \mathit{att}_I \circ (\mathit{id}_{V_I}{\restriction}V_K \cup \{\mathit{ext}_I(i) \mapsto \mathit{att}_H(e)(i) \mid i \in [1, |\mathit{ext}_I|]\})}
\end{array}$$

Lemma 1. *Given labelled and typed hypergraphs* $H, I \in \mathrm{HG}_{\Sigma}$ *and* $e \in E_H$ *with* $\mathit{types}(\mathit{lab}_H(e)) \preceq \mathit{type}_I(\mathit{ext}_I)$*. The result of replacing* e *by* H *is again a hypergraph:* $K = H[I/e] \in \mathrm{HG}_{\Sigma}$*. (Proof in [7])*

Example 5. Reconsider the HC H from Fig. 4(a) containing exactly one nonterminal edge e in H labelled with L. The rank of L is equal to the number of external nodes of the concrete HC I in Fig. 4(b), thus we can replace e by I and get $K = H[I/e]$, depicted in Fig. 4(c). Note that the result is again a HC $K \in \mathrm{HC}_{\Sigma}$, because $\mathit{types}(L) \preceq_N \mathit{lab}_I(\mathit{ext}_I)$ as stated in the following theorem.

Theorem 1 (Edge Replacement in HCs). *Given* $H, I \in \mathrm{HC}_{\Sigma_N}$ *and* $e \in E_H$ *with* $\mathit{types}(\mathit{lab}_H(e)) \preceq_N \mathit{lab}_I(\mathit{ext}_I)$ *it holds that* $H[I/e] \in \mathrm{HC}_{\Sigma_N}$ *(Proof in [7])*

Definition 8 (Data Structure Grammar). *A* Data Structure Grammar *(DSG) over an abstract heap alphabet* Σ_N *is a set of production rules* $p = X \to R$*, with* $X \in N$ *and* $R \in \mathrm{HC}_{\Sigma_N}$*, where* $\mathit{types}(X) \preceq_N \mathit{lab}_R(\mathit{ext}_R)$*.*

We denote the set of all data structure grammars over Σ_N *by* DSG_{Σ_N}*.*

Given grammar $G \in \mathrm{DSG}_{\Sigma_N}$ and graph $H \in \mathrm{HC}_{\Sigma_N}$ we write $H \Rightarrow_G H'$ if there exists a production rule $X \to R \in G$ and an edge $e \in E_H$ with $\mathit{lab}_H(e) = X$ and $H' = H[R/e]$. We write $\Rightarrow_G^*$ for the transitive closure of $\Rightarrow_G$. We say H' is derivable from H over G iff $H \Rightarrow_G^* H'$.

Corollary 1. *Given a data structure grammar* $G \in \mathrm{DSG}_{\Sigma_N}$ *and* $H \in \mathrm{HC}_{\Sigma_N}$ *any derivable graph is a HC:* $H \Rightarrow_G^* H' \in \mathrm{HC}_{\Sigma_N}$*.*

Example 6. Fig. 5 depicts a DSG for trees with linked leaves and parent pointers. The DSG consists of four rules for nonterminal L with $\mathit{types}(L) = I_{\blacktriangle} I_{\triangledown} L_{\triangledown} L_{\blacktriangle}$, introduced in Ex. 4. Every right hand side is a HC with $\mathit{type}(\mathit{ext}) = \mathit{types}(L)$.

The rules define the data structure recursively. The smallest tree represented by L is a tree where the child nodes of the root node are the two leaves. Bigger

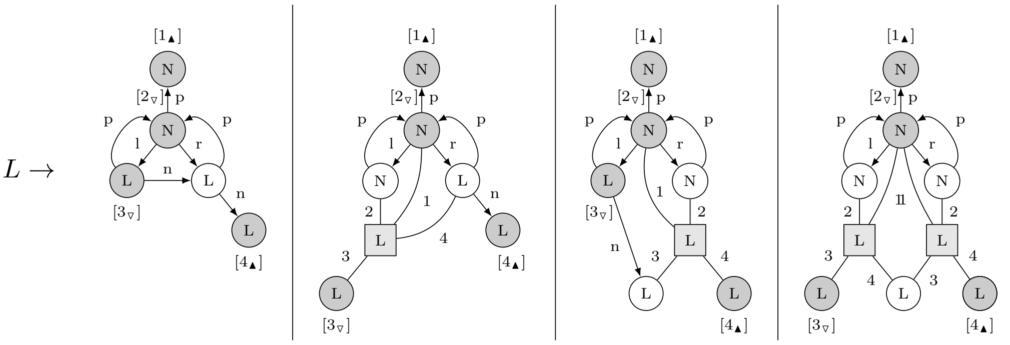


Fig. 5. DSG for trees with linked leafs.

trees are defined recursively as trees where either one (second and third rule) or both (fourth rule) children of the root node are trees, with proper linked leaves.

As our grammar definition does not include a start symbol we define languages in dependence of a start configuration.

Definition 9 (Language). *For $G \in DSG_{\Sigma_N}$ we define the language $L_G(H)$ induced by a start graph $H \in HC_\Sigma$ over G as the set of derivable concrete HCs: $L_G(H) = \{H' \in \mathrm{HC}_\Sigma \mid H \Rightarrow^*_G H'\}$.*

It follows from Corollary 1 that using an abstract HC as start graph and a data structure grammar we can only derive graphs from HC_Σ, i.e. concrete HCs. It remains to show, that the given restrictions do not restrict the expressiveness, i.e. that the languages representable by DSGs are exactly the HC languages ($\subseteq \mathrm{HC}_\Sigma$) representable by HRGs.

Theorem 2. *Given a HRG G over Σ_N. Then a grammar G' over Σ'_N can be constructed such that for any hypergraph S over Σ_N with $L_G(S) \subseteq \mathrm{HC}_\Sigma$ there exists a heap configuration S' with $L_{G'}(S') = L_G(S)$. (Proof in [7])*

For nonterminal X we use $X^\bullet$ to denote the X-handle, i.e a hypergraph consisting of a single nonterminal edge labelled with X and one node for each tentacle:

$$\begin{array}{llll} V_{X^\bullet} & = \{v_i \mid i \in [1, |types(X)|]\} & E_{X^\bullet} & = \{e\} \\ type_{X^\bullet} & = \{v_i \mapsto types(X)[i] \mid i \in [1, |types(X)|]\} & lab_{X^\bullet} & = \{e \mapsto X\} \\ att_{X^\bullet} & = \{e \mapsto v_1, v_2, \ldots v_{|types(X)|}\} & ext_{X^\bullet} & = \emptyset \end{array}$$

Definition 10 (Local Greibach Normal Form [9]). *A grammar $G \in DSG_{\Sigma_N}$ is in* Local Greibach Normal Form (LGNF) *if for every $\triangledown$-tentacle (X, i) there exists $G_{(X,i)} \subseteq G$ with:*

$$L_{G_{(X,i)}}(X^\bullet) = L_G(X^\bullet) \quad and \quad \forall X \Rightarrow R \in G_{(X,i)} : \triangledown_R(ext_R(i)) \subseteq \mathbb{F}$$

Lemma 2. *Any DSG can be transformed into an equivalent DSG in LGNF [9].*

3 An Abstract Java Virtual Machine

Java programs are compiled to Java Bytecode programs that are executed by a *Java Virtual Machine* (JVM). In this section we will define an abstract JVM based on HCs as defined in the previous chapter.

3.1 Java Bytecode and the JVM

Based on the formal definition of Java Bytecode and the JVM from [17], we differentiate between the static environment and the dynamic state of a JVM.

Static Environment of a JVM [17]. A JVM executes programs with respect to a static environment $cEnv : \mathsf{Class} \cup \mathsf{Interface} \rightarrow \mathit{ClassFile}$. For each class of a Java program (top-level, inner or anonymous) a separate class file is compiled.

Definition 11 (Java Class File). *In a Java Bytecode program a* class file *is a tuple* $cf = (name, isInterface, modifiers, super, implements, fields, methods)$, *where* $name \in \mathit{Class} \cup \mathit{Interface}$ *is the unique identifier of the class or interface, if it defines an interface or a class is determined by isInterface,* $modifiers \in \mathcal{P}(\mathit{Modifier})$ *are the modifiers (*static, private, public, ...*),* $super \in \mathit{Class}$ *is the super class and* $implements \in \mathcal{P}(\mathit{Interface})$ *are the implemented interfaces, fields is a mapping* $fields : \mathit{Field} \rightarrow \mathcal{P}(\mathit{Modifier}) \times \mathit{Type}$*, with* $\mathit{Type} = \mathit{Class} \cup \mathit{Interface} \cup \{\mathit{boolean}\}$ *defining the fields of the class, their modifiers and types and the mapping* $methods : \mathit{MSig} \rightarrow \mathit{MDec}$ *defines the methods of the class, where* MSig *is the set of method signatures* $\mathit{MSig} = \mathit{Meth} \times \mathit{Type}^\star$ *with* Meth *the set of method identifier and* MDec *the set of method declarations as defined below.*

$\mathit{ClassFile}$ *denotes the set of all class files of a given Java Bytecode program.*

The sets Class and $\mathit{Interface}$ contain the identifiers of the classes/interfaces, distinguished by $isInterface$: $\mathsf{Class} = \{name_{cf} \mid cf \in \mathit{ClassFile} \wedge \neg isInterface_{cf}\}$ and $\mathsf{Interface} = \{name_{cf} \mid cf \in \mathit{ClassFile} \wedge isInterface_{cf}\}$. We define the sets of available fields $\mathsf{Class/Field} = \{name_{cf}.field \mid cf \in \mathit{ClassFile} \wedge field \in \mathsf{Field}_{cf}\}$ uniquely identified by the combination of class and field name, and the set of methods $\mathsf{Class/MSig} = \{name_{cf}.mSig \mid cf \in \mathit{ClassFile} \wedge mSig \in \mathsf{MSig}_{cf}\}$, uniquely defined by class name and method signature.

Definition 12 (Method Declaration). *A* method declaration *is a tuple* $md = (modifiers, returnType, code, excs, maxReg, maxOpd)$*, with* $modifiers \in \mathcal{P}(\mathit{Modifier})$ *and* $returnType \in \mathit{Type} \cup \{\mathsf{void}\}$*,* $code \in \mathit{Instruction}^\star$ *is a sequence of instruction (*$\mathit{Instruction}$ *is defined in 3.3), excs a set of exceptions (not considered in this paper) and maxOpd is the maximum size of the operand stack, while maxReg is the highest register used by the method. Registers hold the values of local variables.*

The set of all method declarations of a given Java program is denoted by MDec.

$method(c, mSig)$ returns $c'.mSig \in \mathsf{Class/MSig}$ where c' is the class where the corresponding method is declared, i.e. returns the method of the given signature inherited from c. $method$ is given implicitly by $\mathit{ClassFile}$.

State of a JVM. Heap and method stack determine the state of a JVM.

The heap formally is a function $heap : Ref \rightarrow Heap$. $Heap = \mathsf{Class} \times \mathsf{Class/Field} \rightarrow \mathsf{Val}$ [17], is a set of objects defined by the type and evaluation of references.

The method stack consists of frames $stack : \mathsf{Frame}^*$ with $(pc, reg, opd, method) \in \mathsf{Frame}$, where $method \in \mathsf{Class/MSig}$ the corresponding method, $pc \in \mathbb{N}$ the program counter defining the current position in the method, $reg \in \mathbb{N} \to \mathsf{Val}$ defines the values of the registers that are used by the JVM to store the local variable information and $opd \in \mathsf{Val}^\star$ is the operand used to store intermediate results of calculations. The top frame of the stack defines the state of the active method.

3.2 Modelling JVM States by Heap Configurations

Our aim is to model (abstract) states of the JVM by HCs. Starting with a very basic model, only representing the heap and restricting the programs to classes and member variables only we will step by step extend the representation.

The Basic Model. We consider programs in the most basic case where each class file defines a class and all fields are member variables, i.e. $\mathsf{Interface} = \emptyset$. There are no static fields yet. States represent HCs over the heap alphabet Σ with $\mathbb{T} = \mathsf{Class}$, $\mathbb{F} = \mathsf{Class/Field}$ and $types(c.f) = c_\triangledown t_\blacktriangle$, where $t = fieldType(c.f)$.

Interfaces and *null*. Java differentiates between classes and interfaces. Interfaces cannot be instantiated, i.e. there are no objects of an interface type and *null* can be referenced but does not reflect a proper type.

We extend the heap alphabet Σ_N to $\mathbb{T} = \mathsf{Class} \cup \mathsf{Interface} \cup \{\bot\}$, where $\bot$ represents *null*. For all $t \in \mathbb{T}$ we let $\bot \preceq t$, i.e. $\bot$ is the least element. Other elements of $\mathbb{T}$ are ordered as described above – including the interfaces.

We model *null* as an external $\blacktriangle$-node, i.e. a node that can be referenced but is not part of the heap. We consider HCs $(V, E, lab, type, att, \mathtt{null}_\blacktriangle)$ over Σ_N, where $\bot \in \mathbb{T}_{\Sigma_N}$ and $\{v \in V \mid type(v) = \bot\} = \{\mathtt{null}\}$, i.e. the null reference is unique. This is important for comparisons. As $\bot$ is the least element any pointer can point to `null`.

Example 7. Fig. 6 represents a binary tree. Every node is of type Tree: **class** Tree{Tree left , Tree right;} denoted by T. Leafs are Tree objects pointing to *null*. There are only two different types: Tree and $\bot$ with $\bot \preceq Tree$. The external node ($ext = \mathtt{null}_\blacktriangle$) is of type $\bot$ to realise pointers to *null*.

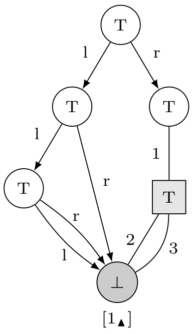


Fig. 6. A Tree

Static Variables. Beside member variables (fields) there are also static variables. These variables are not linked to an object and are accessible from any context. We make them accessible through an external node `static` of (a new) type $\mathsf{static} \in \mathbb{T}$. For any static field $f \in \mathsf{Class/Field}$ we define $types(c.f) = \mathsf{static}_\triangledown t_\blacktriangle$ ($t = fieldType(c.f)$), i.e. type static has one outgoing pointer for each static variable. We extend the sequence of external nodes by a node `static` and get $(V, E, lab, type, att, \mathtt{null}_\blacktriangle \mathsf{static}_\triangledown)$. We require `static` to be the sole static-node, i.e. $\{v \in V \mid type(v) = \mathsf{static}\} = \{\mathtt{static}\}$. `static` is an $\triangledown$-node to the heap.

Literals and Boolean Values. Literals (constants) are a special case of static variables, whose values are implicitly used within a Java program. As we do not consider general data values the only possible literals are the boolean values false and $\mathtt{true}$, in Java Bytecode represented as integer value zero and one.

To model boolean values we add two nodes of the (newly introduced) type $\mathsf{int} \in \mathbb{T}$ representing integer value zero and one, accessible through $\mathtt{static}$ by edges labeled *int(0)* and *int(1)*, i.e. $\mathit{types}(\mathit{int}(0)) = \mathit{types}(\mathit{int}(1)) = \mathsf{static}_{\triangledown}\mathsf{int}_{\blacktriangle}$.

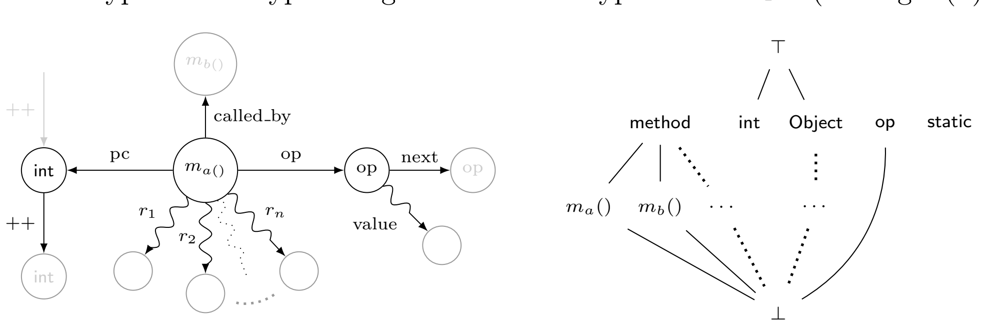
Complete Heap Representation. Given a Java Bytecode program as a set of class files *ClassFile*, we use HCs over the alphabet Σ with $\mathbb{T} = \mathsf{Class} \cup \mathsf{Interface} \cup \{\mathsf{static}, \mathsf{int}, \bot\}$, $\mathbb{F} = \mathsf{Class/Field} \cup \{\mathit{int}(0), \mathit{int}(1)\}$ and

$$\begin{aligned} \mathit{types} = {} & \{c.f \mapsto c_{\triangledown}t_{\blacktriangle} \mid c.f \in \mathsf{Class/Field}_o \wedge \mathit{fieldType}(c.f) = t\} \\ \cup {} & \{c.f \mapsto \mathsf{static}_{\triangledown}t_{\blacktriangle} \mid c.f \in \mathsf{Class/Field}_s \wedge \mathit{fieldType}(c.f) = t\} \\ \cup {} & \{\mathit{int}(0) \mapsto \mathsf{static}_{\triangledown}\mathsf{int}_{\blacktriangle}, \mathit{int}(1) \mapsto \mathsf{static}_{\triangledown}\mathsf{int}_{\blacktriangle}\} \end{aligned}$$

We use a HC of the form $H = (V, E, \mathit{lab}, \mathit{type}, \mathit{att}, \mathtt{static}_{\triangledown}\mathtt{null}_{\blacktriangle})$, where none of the nodes is of an interface type $\{v \in V \mid \mathit{type}(v) \in \mathsf{Interface}\} = \emptyset$, node $\mathtt{null}$ is the only node of type $\bot$ $\{v \in V \mid \mathit{type}(v) = \bot\} = \{\mathtt{null}\}$ and node $\mathtt{static}$ the only one of type static $\bot$ $\{v \in V \mid \mathit{type}(v) = \mathsf{static}\} = \{\mathtt{static}\}$. The only two int-nodes $\{v \in V \mid \mathit{type}(v) = \mathsf{int}\} = \{v_{\mathit{int}(0)}, v_{\mathit{int}(1)}\}$ are successors of the node $\mathtt{static}$: $\exists e_0, e_1 \in \triangledown(\mathtt{static}) : \mathit{lab}(e_i) = \mathit{int}(i) \wedge \mathit{att}(e_i)[2] = v_{\mathit{int}(i)}$.

Method Stack. So far we only considered the heap component. Now we will model the method stack and its components within the same HC by representing each stack frame as a node of a special method type. This allows us to abstract the stack and handle recursive functions without bounded method stack size. It is preferable to model both parts in one homogeneous representation as abstracting heap and stack independently would imply loosing the relation between the two.

We model each frame by one node. For each method $c.m \in \mathsf{Class/MSig}$ we define a proper type $\mathsf{c.m}$, reflecting the *method* component of the frame. Each method type is a subtype of a general method type $\mathsf{method} \in \mathbb{T}$ (see Fig. 7(b)).



(a) Method notes represent frames

(b) Extended type order

Fig. 7. Method nodes represent frames of the method stack.

For the program counter we add one int-node for each value, i.e. we add the nodes $\{v_{\mathit{int(i)}} \mid i \in [0, \mathit{max}(\{|\mathit{code}(c.m)| \mid c.m \in \mathsf{Class/MSig}\})]\}$ and fields $\mathit{int(i)}$ as pointers from static to int. Further we add the field $++$ with $\mathit{types}(++ = \mathsf{int}_{\triangledown}\mathsf{int}_{\blacktriangle})$ representing the successor relation between int-nodes. The program counter is modelled as a pointer $\mathit{method.pc} \in \mathbb{F}$ to the corresponding int-node, i.e. $\mathit{types}(\mathit{method.pc}) = \mathsf{method}_{\triangledown}\mathsf{int}_{\blacktriangle}$. For the operand stack we add an op-type for stack elements with next and value successors, $\mathit{types}(\mathit{op.next}) = \mathsf{op}_{\triangledown}\mathsf{op}_{\blacktriangle}$ and $\mathit{types}(\mathit{op.value}) = \mathsf{op}_{\triangledown}\top_{\blacktriangle}$, where $\top \in \mathbb{T}$ with int and Object as subtypes (see Fig. 7(b)), i.e. $\mathit{op.value}$ can reference int- and Object-nodes. We add a pointer to each method-node $\mathit{op} \in \mathbb{F}$ to the operand stack ($\mathit{types}(\mathit{op}) = \mathsf{method}_{\triangledown}\mathsf{op}_{\blacktriangle}$).

As registers offer random access we model each register i by a proper pointer r_i. The amount of registers depends on the method, therefore we define the r_i-pointer for each specification of the method-type. Given $c.m \in \mathsf{Class/MSig}$ we define for each $i \in [1, \mathit{maxReg}_{c.m}]$: $\mathit{types}(r_i) = \mathsf{c.m}_{\triangledown}\top_{\blacktriangle}$ (see Fig. 7(a)).

We model the method stack itself by an additional field $\mathit{called\_by} \in \mathbb{F}$ with $\mathit{types}(\mathit{called\_by}) = \mathsf{method}_{\triangledown}\mathsf{method}_{\blacktriangle}$ referencing the predecessor of the node within the stack (where the least element in the stack points to `null`). The method at the top of the stack is the active method. The corresponding node contains the information that can be modified currently. Thus the top method node can access the heap and is modelled as an external $\triangledown$-node. We get HCs of the following form: $(V, E, \mathit{lab}, \mathit{type}, \mathit{att}, \mathtt{method}_{\triangledown}\mathtt{static}_{\triangledown}\mathtt{null}_{\blacktriangle})$.

Example 8. In Fig. 7 a typical recursive traversal algorithm is given in Java (a) and the corresponding Java Bytecode (b) (details on Bytecode in Section 3.3). In Fig. 7(c) a state of the program from (b) is depicted. In this state a new method *trav(Tree t)* is called, the program counter is still zero. The *trav* method was called various times, three methods are concrete, furthers are abstracted in the nonterminal edge X. Each method call was a *trav(t.left)* call as each program counter points to $i(4)$. Note that in X method and tree nodes are abstracted.

Abstract JVM States So far we considered HCs over the concrete alphabet Σ, thus concrete HCs. To represent an abstract state we extend the alphabet by a set of nonterminals N as defined before (see Example. 3), where we restrict types over N to $\mathit{types} : N \to (\mathsf{Class}_{\triangledown\blacktriangle} \cup \mathsf{Class/MSig}_{\triangledown\blacktriangle} \cup \mathsf{Interface}_{\blacktriangle} \cup \{\bot_{\blacktriangle}, \top_{\blacktriangle}\})^{\star}$, i.e. only class- and method-nodes can be connected to $\triangledown$-tentacles. From this restriction it follows that $\mathit{type}(v) \in \mathsf{Interface} \cup \{\top, \bot\} \Rightarrow v_{\blacktriangle} \in [\mathit{ext}]$.

Given $H \in \mathrm{HC}_{\Sigma_N}$ we call a node $v \in V_H$ concrete if its successors are concrete ($\triangledown(v) \subseteq \mathbb{T}$), abstract otherwise ($\triangledown(v) \subseteq N$). Concrete and abstract parts coexist on a HC. A HC without abstract nodes is called concrete, otherwise abstract.

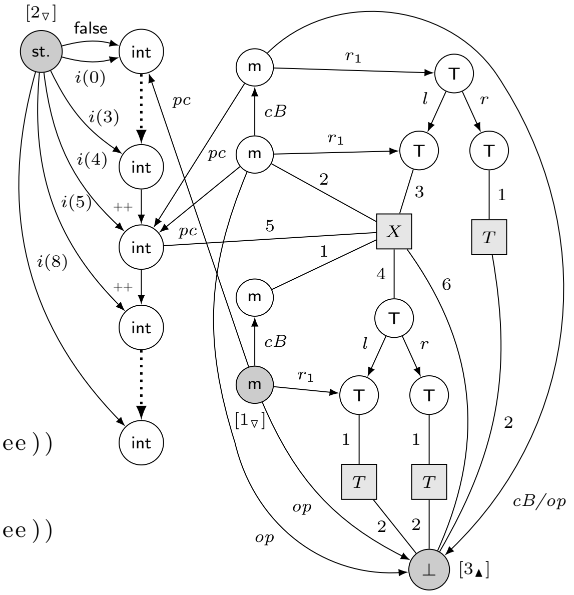
Concretisation. Concretisations are defined through the application of grammar rules, i.e. given $H, H' \in \mathrm{HC}_{\Sigma_N}$ H' is a concretisation of H iff $H \Rightarrow H'$. $L(H)$ is the set of all concrete HCs represented by the (abstract) HC H. Given a DSG in LGNF we can systematically concretise an abstract node v by replacing the connected $\triangledown$-tentacle by the corresponding rules from the grammar. Correspondingly we define for $H \in \mathrm{HC}_{\Sigma_N}$ and abstract node $v \in V$:

```
public class Tree{
  Tree left;
  Tree right;
  static void trav(Tree t){
    if(t != null){
      trav(t.left);
      trav(t.right);
    }
  }
}
```

(a) Java Class Definition

```
0  Load(Tree, 0)
1  Cond(ifNull, 8)
2  Load(Tree, 0)
3  GetField(Tree, Tree.left)
4  InvokeStatic(void, Tree.trav(Tree))
5  Load(Tree, 0)
6  GetField(Tree, Tree.right)
7  InvokeStatic(void, Tree.trav(Tree))
8  Return(void)
```

(b) Java Bytecode: trav(Tree t)



(c) State of the abstract JVM

Fig. 8. Recursive Tree Traversal.

$$conc_G(H, v) = \{H[e/R] \mid \triangledown_H(v) = \{e\} \wedge att[i](e) = v \wedge (lab(e) \to R) \in G_{(lab(e),i)}\}$$

and if v is a concrete node then $conc_G(H, v) = H$, thus *conc* has no effect. Note that for any $H \in \mathrm{HC}_{\Sigma_N}$ and $G \in \mathrm{DSG}_{\Sigma_N}$: $L_G(H) = L_G(conc_G(H, v))$

We call a method $m : \mathrm{HC}_{\Sigma_N} \to \mathrm{HC}_{\Sigma_N}$ a concrete modifier for $H \in \mathrm{HC}_{\Sigma_N}$ iff $L_G(m(H)) = m(L_G(H))$, i.e. the modification m is safe and most precise under the abstraction. As long as a modifier uses the information of concrete nodes and their incident edges only and preserves abstracted parts it is a concrete modifier.

Abstraction. Abstraction is defined through backward application of grammar rules, i.e. given $H, H' \in \mathrm{HC}_{\Sigma_N}$ H is an abstraction of H' iff $H' \Rightarrow H$. In practice abstraction is realised by the search of embeddings of rule graphs and followed by embedding replacement with the corresponding nonterminals.

Definition 13 (Embedding). *Given $I, H \in \mathrm{HC}_{\Sigma_N}$ an* embedding *of I in H consists of two mappings $emb : (V_I \to V_H, E_I \to E_H)$ with following properties:*

$$\begin{array}{lcll}
emb(v) & \neq & emb(v') & \forall v \neq v' \in V_I \setminus \{v \in V_I \mid v_\blacktriangle \in ext_I\} \\
emb(e) & \neq & emb(e') & \forall e \neq e' \in E_I \\
lab_I(e) & = & lab_H(emb(e)) & \forall e \in E_I \\
type_I(v) & \preceq & type_H(emb(v)) & \forall v \in \{v \in V_I \mid v_\blacktriangle \in ext_I\} \\
type_I(v) & = & type_H(emb(v)) & \forall v \in V_I \setminus [ext_I] \\
emb(att_I(e)) & = & att_H(emb(e)) & \forall e \in E_H \\
e \notin emb(E_I) & \Rightarrow & [att_I(e)] \cap emb(V_I) = \emptyset & \forall e \in E_H
\end{array}$$

Given $I, H \in \mathrm{HC}_{\Sigma_N}$. $Emb(I, H)$ denotes the set of all embeddings of I in H.

Given $G \in \mathrm{DSG}_{\Sigma_N}, I, H \in \mathrm{HC}_{\Sigma_N}, emb \in Emb(I,H)$ and $X \in N$ the replacement of I in H is $replace(I,H,emb,X) = K$, with:

$$\begin{array}{llll}
V_K &= V_H \setminus emb(V_I \setminus [ext_I]) & E_K &= (E_H \setminus emb(E_I)) \uplus \{e\} \\
lab_K &= lab_H{\restriction}E_K \cup \{e \mapsto N\} & type_K &= type_H{\restriction}V_K \\
att_K &= att_H{\restriction}E_K \cup \{e \mapsto emb(ext_I)\} & ext_K &= ext_H
\end{array}$$

$abstr_G(H) = \{replace(R,H,emb,X) \mid X \to R \in G, emb \in Emb(R,H)\}$ are the HCs we get via one abstraction step ($H' \in abstr_G(H)$ *iff* $H' \Rightarrow_G H$), $abstr^\star_G(H)$ is the transitive closure ($H' \in abstr^\star_G(H)$ *iff* $H' \Rightarrow^\star_G H$). $fullAbstr_G(H)$ are the fully abstracted HCs, i.e. $\{H' \in abstr^\star_G(H) \mid \forall X \to R \in G : Emb(R,H') = \emptyset\}$. Note that the set $fullAbstr_G(H)$ in general is not a singleton but finite. We call G *backward confluent* iff $fullAbstr_G(H)$ is a singleton for any $H \in \mathrm{HC}_{\Sigma_N}$.

Garbage Collection Javas garbage collector removes objects that are no longer reachable. We model its behaviour by the following method *garbage collection*:

Definition 14 (Garbage Collection). *Any node not reachable from an external node is considered to be* garbage. *Given $H \in \mathrm{HC}_{\Sigma_N}$, $n_1, n_2 \in V_H$, n_2 is reachable from n_1 ($n_1 \rightsquigarrow n_2$) iff $\exists e \in \nabla(n_1), n' \in att(e) : n' = n_2 \vee n' \rightsquigarrow n_2$.* $\mathrm{gc}(H) = (V', E', lab{\restriction}E', type{\restriction}V', type{\restriction}E', ext)$ *denotes the result of garbage collection on $H \in \mathrm{HC}_{\Sigma_N}$ with $V' = \{v \in V_H \mid \exists v' \in [ext_H] \rightsquigarrow v\}$ and $E' = \nabla(V')$.*

Any example considered so far was garbage free. Note that gc is not a concrete modifier, i.e. $L_G(\mathrm{gc}(H)) = \mathrm{gc}(L_G(H))$ does not hold in general as gc uses connectivity information of nonterminal edges. However $L_G(\mathrm{gc}(H)) \subseteq \mathrm{gc}(L_G(H))$.

3.3 Execution of Java Bytecode

The following abstract instructions [17] cover the whole instructions set:

Prim(PrimOp)	Dupx()	Pop()
Load(Type, RegNo)	Store(Type, RegNo)	Goto(LineNumber)
Cond(PrimOp, LineNumber)		
GetStatic(Type, Class/Field)	PutStatic(Type, Class/Field)	InvokeStatic(Type, Class/MSig)
Return(Type)		
New(Class)	Return(Type)	InstanceOf(Type)
GetField(Type, Class/Field)	PutField(Type, Class/Field)	Checkcast(Type)
InvokeSpecial(Type, Class/MSig)	InvokeVirtual(Type, Class/MSig)	
Athrow	Jsr(LineNumber)	Ret(RegNo)

We defined and implemented the transition rules for the above abstract instructions up to the grey ones (used for exception handling). The Type information in the instructions can be used to check for type safeness but are ignored by the JVM. Focusing on heap structures we do not consider data values. This results in a notable cutback of primary operations. The following are supported:

if_acmpeq   if_acmpne   if_icmpeq   if_icmpne
iconst_0   iconst_1   iand   ior

The primary *if*-operations, used by the Cond instruction, are realised by comparing the corresponding nodes referred by the stack, iconst_0 and iconst_1 pushes the corresponding int-nodes on the stack. iand and ior can be defined explicitly for the four possible inputs. Note that the set $\{0, 1\}$ is closed under both operations. We present a selection of instructions and their transition rules. Most of the instructions are realised as concrete modifiers expecting external nodes and the actual operand stack to be concrete. We use a slightly modified, restricted version of *fullAbstr* that excludes the external nodes from the abstractable parts.

Graph Transformations We define some basic actions, shared by several Bytecode instruction (as push, pop, etc.) to use them later for the transition rules.

$new(H, t) = (H', v_{new})$ adds a new node to a HC. Given $H \in \mathrm{HC}_{\Sigma_N}$ and $t \in \mathbb{T}$ we get $H' = (V_H \uplus \{v_{new}\}, E_H, lab_H, type_H \cup \{v_{new} \mapsto t\},\ att_H, ext_H)$

suc (H, v, f) returns for $H \in \mathrm{HC}_{\Sigma_N}$, $v \in V_H$ and $f \in \mathbb{F}$ the f-successor of v $suc(H, v, f) = v'$, i.e. $\{v'\} = att_H(\{e \in \triangledown_H(v) \mid lab_H(e) = f\})[2]$.

setSuc (H, v, f, v') alters for $H \in \mathrm{HC}_{\Sigma_N}$, $v, v' \in V_H$ and $f \in \mathbb{F}$ the edge representing the f-pointer of v: $setSuc(H, v, f, v') = (V_H, E_H, lab_H, type_h, att_h[e \mapsto v\ v'], ext_h)$, where e is the single element of $\{e \in \triangledown_H(v) \mid lab(e) = f\} = \{e\}$.

$pushOp(H, v)$ pushes a reference to $v \in V_H$ onto the operand stack by adding a node of type op $(H_{new}, v_{op}) = new(H, \mathsf{op})$ and connecting the *next*-edge to the operand stack $v_{top} = suc(H, ext_H[1], op)$ and the *value*-edge to node v: $H' = setSuc(setSuc(H_{new}, v_{op}, value, v), v_{op}, next, v_{top})$. The reference to the operand stack is updated for the top method: $pushOp(H, v) = setSuc(H', ext'_H[1], op, v_{op})$.

$popOp(H) = (H', v)$ pops the top-element $v_{top} = suc(H, ext_H[1], op)$ by altering the *op*-edge to the next operand $H' = setSuc(H, ext'_H[1], op, suc(H, v_{top}, op))$. The value of the removed stack element $v = suc(H, v_{top}, value)$ is returned.

$peekOp(H, i) = suc^n(H, ext_H[1], op)$, returns for $H \in \mathrm{HC}_{\Sigma_N}, n \in \mathbb{N}$ the n_{th} element of the operand stack, where suc^n is defined recursively as $suc^n(H, v, f) = suc^{n-1}(H, suc(H, v, f), f)$ for $n > 0$ and $suc^0(H, v, f) = H$.

$incPc(H)$ increments the program counter $v_{pc} = suc(H, ext_H[1], pc)$, by altering it to the successor node $incPc(H) = setSuc(H, ext_H[1], pc, suc(H, v_{pc}, ++))$.

$inst(H)$ returns the current instruction. Referring to the static environment *cEnv* of the program, the instruction is determined by the type and *pc*-successor of the top method node. $inst(H) = c[pc]$, where $c = \mathsf{code}(type_H(ext_H[1]))$ and $pc = intValue(suc(H, \mathtt{method}_H, pc))$.

Transition Rules We define the following transition rules:

Load(RegNo). The Load instruction reads a reference from the Register RegNo and pushes it to the operand stack. We determine the node corresponding to the value of the register and push it to the stack.

$$\frac{\mathit{inst}(H) = \mathsf{Load(t,i)}}{H \to \mathit{incPc}(\mathit{pushOp}(H, \mathit{suc}(H, \mathtt{method}_H, r_i)))}$$

Dupx() Duplicates the topmost element of the operand stack.

$$\frac{\mathit{inst}(H) = \mathsf{Dupx()} \qquad \mathit{popOp}(H) = (H', v)}{H \to \mathit{incPc}(\,\mathit{pushOp}((\,\mathit{pushOp}(H', v)), v)))}$$

GetStatic(Class/Field) Reads a static variable and pushes the result to the operand stack.

$$\frac{\mathit{inst}(H) = \mathsf{GetStatic(c.f)}}{H \to \mathit{incPc}(\,\mathit{pushOp}(H, \mathit{suc}(H, \mathit{ext}_H[2], c.f))}$$

PutField(Class/Field) writes a value to a field of an objects. As the node that represents the object could be abstract we concretise it before we set the field. The update could be destructive and could yield garbage. Therefore we perform a garbage collection on the result and afterwards try to abstract. Note that this instruction is not deterministic.

$$\frac{\mathit{inst}(H) = \mathsf{PutField(f)} \qquad \mathit{popOp}(H) = (H', v') \qquad \mathit{popOp}(H') = (H'', v'')}{H \to \mathit{fullAbstr}(\mathrm{gc}(\,\mathit{incPc}(\,\mathit{setSuc}(K, v'', f, v)))), K \in \mathit{conc}(H'', v'')}$$

InvokeVirtual(Class/MSig) is the call of an object method. We add a new method node and set the registers to the parameters given in msig $= \mathit{Name}(p)$, with $p \in \mathsf{Type}^{\star}$. This instruction uses information from the static environment cEnv of the program.

$$\frac{\mathit{inst}(H) = \mathsf{InvokeVirtual(c.msig)} \qquad \mathit{popOp}(H) = (H', v)}{H \to \mathit{call}(H', \mathit{method}_{\mathit{cEnv}}(\mathit{type}(v), c.\mathit{msig})))}$$

where $\mathit{call}(H, c.m(p)) = \mathit{setSuc}(K, \mathit{op}, \mathit{peekOp}(H, |p|+1))$ with:

$$\begin{aligned}
V_K &= V_H \uplus \{v_m\}\\
E_K &= E_H \uplus \{e_{\mathit{calledBy}}, e_{\mathit{op}}, e_{\mathit{pc}}\} \uplus \{e_{r_i} \mid i \in [1, \mathit{maxReg}_{c.m(p)}]\}\\
\mathit{lab}_K &= \mathit{lab}_H \uplus \{e_x \mapsto x \mid e_x \in E_K \setminus E_H\}\\
\mathit{type}_K &= \mathit{type}_H \cup \{v_m \mapsto c.m(p)\}\\
\mathit{att}_K &= \mathit{att}_K\\
&\quad \cup \{e_{\mathit{calledBy}} \mapsto \mathit{ext}_H[1], e_{\mathit{op}} \mapsto \mathit{ext}_H[3], e_{\mathit{pc}} \mapsto \mathit{suc}(H, \mathit{ext}_H[2], \mathit{int}(0))\}\\
&\quad \cup \{e_{r_i} \mapsto \mathit{peekOp}(i) \mid i \in [1, |p|]\}\\
&\quad \cup \{e_{r_i} \mapsto \mathit{ext}_H[3] \mid i \in [|p|+1, \mathit{maxReg}_{c.m(p)}]\}\\
\mathit{ext}_K &= v_m \triangledown\ \mathit{ext}_H[2]\ \mathit{ext}_H[3]
\end{aligned}$$

4 Experimental Results

We implemented the above concepts in a prototype tool that for a program in Java Bytecode, a hyperedge replacement grammar and a start heap generates the abstracted state space. The following table gives some experimental results:

Method	Rules	States	Parsing	Generation
ReverseList (singly linked)	3	113	0:010 s	0:006 s
TraverseTree (recursive)	49	574	0:472 s	0:264 s
Lindstrom (no marking)	14	4,297	0:245 s	0:198 s
Lindstrom (single marking)	14	224,113	0:245 s	2:360 s
Lindstrom (extended marking)	14	937,510	0:245 s	9:074 s

TraverseTree is the Java program from Fig. 8. The *Lindstrom Traversal Algorithm* [11] traverses a tree with constant additional memory by altering the pointers of the elements. The algorithm (Fig 9) was analysed by us before [8].

The column *rules* gives the size of the provided grammar, *states* the size of the corresponding state space, *parsing* the time for parsing the Bytecode, grammar and start heap and *generation* the time needed to generate the abstracted state space. The examples where calculated on a 2 GHz Intel Core i7 Laptop.

```
static void trav(Tree root){
  if(root == null) return;
  Tree sen = new Tree();
  Tree prev = sen;
  Tree cur  = root;
  while(cur != sen){
    Tree next = cur.left;
    cur.left  = cur.right;
    cur.right = prev;
    prev = cur;
    cur  = next;
    if (cur == null){
      cur  = prev;
      prev = null;
} } }
```

Fig. 9. Lindstrom Traversal

In none of the given examples null pointer dereferencing occurs. We can check shape properties of states – so far manually. To describe complex properties we use LTL with pointer equations, e.g. $x.l = y$ and flag *terminal* as atomic propositions. For Lindstrom we proofed *termination*, *completeness* (each node is visited) and *correctness* (at the end the input tree is not altered)[8].

We need quantification over objects as in [14] to verify these properties. We realise quantification by adding markings, i.e. static variables not visible to the program. Markings are determined by exhaustive object exploration where objects are concretised and abstracted as needed. For Lindstrom we get 41 different abstract markings, in each one object is marked as x. We can proof that for any of these the LTL formulas $\mathbf{FG}(cur \neq x)$ and $\neg(cur \neq x\,\mathbf{U}\,terminal)$ hold. The former stats that variable *cur* only finally often points to the marked object and as this could be any object (and *null*) the calculation has to terminate somewhen. The latter stats that somewhen before terminating *cur* points to the marked node, thus the algorithm is complete. For correctness we also mark the left and right successor of x by x_l and x_r and check that the successors are the same at the end of the traversal: $x = root \to \mathbf{G}(x = root) \land \mathbf{G}(terminal \to (x_l = x.l \land x_r = x.r))$. Markings increase the state space (see row single and extended marking). The above only works for quantification over objects in the start heap. The termination check is only correct if no objects are generated at runtime.

5 Related Work

The basic idea of using hyperedge replacement grammars for abstraction of heap structures was supposed in [8, 9]. However it was not suitable for the analysis of Java Bytecode as typed objects were not reflected. Various other techniques for the analysis and verification of heap manipulating programs where supposed. The most famous are *shape analysis via three-valued logic* [16] and *separation logic* [15]. The latter is an extension of Hoare logic and uses recursive predicates to describe the shape of heap parts. There is a one to one corespondents between recursive predicates and nonterminals of our representation, as stated by [5]. Separation logic is classically used in Hoare Triple style verification where decidability of entailment is essential. Entailment not decidable in general has to be proven decidable for any recursive predicate. There are decidable logics for list, and trees [1, 2]. In [13] separation logic is extended for Bytecode by adding type information. There are several separation logic tools as SpaceInvador [18](linear data structures) or Smallfoot [2] ((doubly) linked lists and trees). The advantage of tools based on deductive methods is scaleability [18] while restricted to a small set of predefined data structures. Using state space exploration we do not need to decide entailment and thus support arbitrary user defined data structures.

Shape analysis via three-valued logic is another abstraction technique [15]. Nodes are summarised by properties expressed as predicates in a three-valued logic. Predicates are typically shape properties like reachability, cycle membership, etc. Most of these are implicitly given by our representation and should be extractable [4]. Not relaying on these properties they are implicitly considered during state space exploration. While reflecting predicates nevertheless nodes with different predicates are summarise if they form a neighbourhood of a well defined structure resulting in additional structural information. Unfortunately structures expressible by HRG and separation logic are restricted to those with bounded tree-width [6], e.g. the set of all graphs is not expressible as a HRGs.

In [12] TVLA [10] is used to verify the Lindstrom Algorithm. The given proof depends on 24 predicates encoding deep knowledge of the algorithm, resulting in a less automatic proof than the one provided in section 4 where only structural properties are provided. To reduce the number of predicates and to keep the example manageable the input-code is modified in [12] such that the heap is always a tree. This is not necessary in our approach as it is robust against local violations of the data structure. Our abstraction results in slightly larger state space but also in shorter runtime (TVLA: 183,564 states in over 36 minutes [3]).

6 Conclusion

We introduced labeled and typed hypergraphs and corresponding HRGs, where nodes are associated with a type from a type hierarchy. We showed how they can be used to model abstracted JVM states and how to compute an abstract state space for a Java Bytecode program. Experimental results attest that the approach has a practical value. For the future we will consider automatic inference of DSGs during state space generation, and extended verification techniques.

References

1. Berdine, J., Calcagno, C., O'Hearn, P.W.: A Decidable Fragment of Separation Logic. In Lodaya, K., Mahajan, M., eds.: FSTTCS. Volume 3328 of LNCS, Springer (2004) 97–109
2. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular Automatic Assertion Checking with Separation logic. In de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P., eds.: FMCO. Volume 4111 of LNCS, Springer (2005) 115–137
3. Bogudlov, I., Lev-Ami, T., Reps, T.W., Sagiv, M.: Revamping TVLA: Making Parametric Shape Analysis Competitive. In Damm, W., Hermanns, H., eds.: CAV. Volume 4590 of LNCS, Springer (2007) 221–225
4. Courcelle, B.: The Expression of Graph Properties and Graph Transformations in Monadic Second-Order Logic. In Rozenberg, G., ed.: Handbook of Graph Grammars, World Scientific (1997) 313–400
5. Dodds, M., Plump, D.: From Hyperedge Replacement to Separation Logic and Back. ECEASST **16** (2008)
6. Drewes, F., Kreowski, H.J., Habel, A.: Hyperedge replacement graph grammars. In Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations. World Scientific Publishing (1997) 95–162
7. Heinen, J., Jansen, C.: Juggrnaut - An Abstract Jvm. Technical Report AIB-2011-21, RWTH Aachen (September 2011)
8. Heinen, J., Noll, T., Rieger, S.: Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures. ENTCS **266** (2010) 93 – 107 Proceedings of the 3rd International Workshop on Harnessing Theories for Tool Support in Software (TTSS).
9. Jansen, C., Heinen, J., Katoen, J.P., Noll, T.: A Local Greibach Normal Form for Hyperedge Replacement Grammars. In Dediu, A.H., Inenaga, S., Martín-Vide, C., eds.: LATA. Volume 6638 of LNCS, Springer (2011) 323–335
10. Lev-Ami, T., Sagiv, S.: TVLA: A System for Implementing Static Analyses. In Palsberg, J., ed.: SAS. Volume 1824 of LNCS, Springer (2000) 280–301
11. Lindstrom, G.: Scanning List Structures Without Stacks or Tag Bits. Inf. Process. Lett. **2**(2) (1973) 47–51
12. Loginov, A., Reps, T.W., Sagiv, M.: Automated Verification of the Deutsch-Schorr-Waite Tree-Traversal algorithm. In Yi, K., ed.: SAS. Volume 4134 of LNCS, Springer (2006) 261–279
13. Luo, C., He, G., Qin, S.: A Heap Model for Java Bytecode to Support Separation Logic. In: APSEC, IEEE (2008) 127–134
14. Rensink, A.: Model Checking Quantified Computation Tree Logic. In Baier, C., Hermanns, H., eds.: CONCUR. Volume 4137 of LNCS, Springer (2006) 110–125
15. Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: LICS, IEEE Computer Society (2002) 55–74
16. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric Shape Analysis via 3-valued Logic. ACM Trans. Program. Lang. Syst. **24**(3) (2002) 217–298
17. Stärk, R.F., Schmid, J., Börger, E.: Java and the Java Virtual Machine: Definition, Verification, Validation. Springer (2001)
18. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable Shape Analysis for Systems Code. In Gupta, A., Malik, S., eds.: CAV. Volume 5123 of LNCS, Springer (2008) 385–398

A Verified Implementation of Priority Monitors in Java

Ángel Herranz and Julio Mariño[*]

Babel research group
Universidad Politécnica de Madrid

Abstract. Java monitors as implemented in the `java.util.concurrent.locks` package provide *no-priority nonblocking* monitors. That is, threads signalled after blocking on a *condition* queue do not proceed immediately, but they have to wait until both the signalling thread and also other threads that might have requested the lock release it. This can be a source of errors (if threads that get in the middle leave the monitor in a state incompatible with the signalled thread re-entry) or inefficiency (if repeated evaluation of preconditions is used to ensure safe re-entry). A concise implementation of priority nonblocking monitors in Java is presented. Curiously, our monitors are implemented on top of the standard no-priority implementation. In order to verify the correctness of our solution, a formal transition model (that includes a formalisation of Java locks and conditions) has been defined and checked using Uppaal. We consider this a first step towards a full correctness proof using deductive methods.

Keywords: Monitors, Java, model checking, priority, nonblocking.

1 Introduction

A *model-driven* approach to the development of concurrent software [6] advocates the use of high-level, language independent entities that can be subject to formal analysis (e.g., to early detect risks due to concurrency in reactive, critical systems) and that are later translated into a concrete programming language by means of safe transformations.

This, rather than the unrestricted use of the concurrency mechanisms in the target language, aims at improving portability (always a must in embedded systems) and preventing hazards due to the use of error-prone or poorly documented primitives. We have used this approach for teaching concurrency for more than fifteen years at our university, first using Ada95 [3], and now Java, which is certainly lacking in many aspects when compared to the former.

Indeed, the early mechanisms for thread synchronization provided by Java were so limited that one of the pioneers of concurrent programming, Brinch

[*] Authors partly funded by grants DESAFIOS10 (TIN2009-14599-C03-00) from the Spanish Ministry of Science and Innovation, PROMETIDOS (P2009/TIC-1465) from the Madrid Regional Government and COST Action IC0701 on Formal Verification of Object Oriented Software.

CADT Counter
ACTION Inc: $C\_Type[io]$
ACTION Dec: $C\_Type[io]$

SEMANTICS
TYPE: $C\_Type = \mathbb{Z}$
INVARIANT: $\forall\, c \in C\_Type. c \geq 0$
INITIAL(c)**:** $c = 0$

CPRE: *True*
Inc(c)
POST: $c^{\text{out}} = c^{\text{in}} + 1$

CPRE: $c > 0$
Dec(c)
POST: $c^{\text{out}} = c^{\text{in}} - 1$

```
class Counter {
    final Lock mutex =
        new ReentrantLock(true);
    final Condition strictlyPos =
        mutex.newCondition();
    private int c = 0;

    public void inc() {
        mutex.lock();
        c++;
        // signal some pending dec
        strictlyPos.signal();
        mutex.unlock();
    }

    public void dec() {
        mutex.lock();
        // check the CPRE now
        if (c == 0) {
            try {strictlyPos.await();}
            catch(Exception e) {}
        }
        // if we are here, that means
        // that c > 0
        c--;
        // it CAN be proved that
        // no more signals are needed
        mutex.unlock();
    }
}
```

Fig. 1. A minimal example showing the risks of Java's no-priority locks and conditions.

Hansen, wrote a famous essay [5] alerting on the risks of their usage. As a response to these adverse reactions, more powerful concurrency mechanisms were introduced in later versions of Java, although without being explicitly presented as realizations of the *classical* ones.

In this work we will focus on the *locks & conditions* library (`java.util.concurrent.locks`), that can be seen as an attempt to implement *monitors* (actually, Brinch Hansen's contribution) in Java. Figure 1 shows an (apparently correct) implementation of a shared resource, specified in a formal notation, using the aforementioned library.

On the left side of the figure, a shared counter is specified as a kind of abstract data type with implicit mutually exclusive access and an interface that specifies the only operations (**Inc** and **Dec**) that can be used to modify its state. Moreover, these *actions* are pure and transactional. In this case, we also state the invariant that the counter cannot be negative at any time, and specify the conditional synchronization (CPRE, *concurrency precondition*) for **Dec**, that ensures the invariant after the execution of any of the two actions.

To the right, we see an implementation of the resource specification as a Java class using locks and conditions. Class `Counter` encapsulates an integer variable `c` to represent the state of the counter. Also, a lock object `mutex` is used to ensure execution of the actions in mutual exclusion. Finally, a condition object `strictlyPos` is used to implement the conditional synchronization of threads invoking `dec()`, i.e., to put them to sleep when the action is invoked with the counter set to 0.

The code has been written following the principle that only 0 or 1 `await()` calls are executed during an action, and that the same is true for `signal()`. This coding guideline really pays off when applied to concurrency problems of greater significance. Basically, it reduces the context switching points inside the code for an action to (at most) one (the execution of `await()`), dividing it into two sequential fragments, which facilitates formal reasoning.

It is the responsibility of the signalling thread (in this case, any thread executing `Inc`) to ensure that the signalled thread wakes up in a safe state. In this case, there is nothing to check, as `strictlyPos.signal()` is placed right after incrementing the counter and thus, if the signalled thread resumes execution right after the signalling one executes `mutex.unlock()` it is safe to decrement the counter. That justifies the comment above the line containing "`c--;`".

Unfortunately, this code is unsafe. Why? The reason is that Java's implementation is an example of *no-priority* signalling [2]. That means that threads signalled after blocking on a *condition* queue do not proceed immediately, but they have to wait until both the signalling thread and also other threads that might have requested the lock release it. In other words, when the decrementer thread resumes execution it could be the case that other decrementer thread was already queued on `mutex` and set the counter to 0 right before. Moreover, given that the execution of the `inc()` and `dec()` methods takes very little time, the probability that this scenario happens is relatively low, what makes this the kind of error that can go unnoticed for thousands of runs.

The official documentation for the `Condition` class leaves considerable room for the operational interpretation of the `await()` and `signal()` methods in future implementations and, in fact, the no-priority behaviour is not really implied by the natural language specification of the API. There is, however, a generic recommendation to enclose the calls to `await()` inside invariant-checking loops, motivated by the possibility of *spurious* wakeups. Although the use of such loops can be seen as a defensive coding technique that can avoid hazards in general, we think that forcing the application programmer to use them can be quite unsatisfactory in many situations as testing the concurrency precondition repeatedly can increase contention of certain concurrent algorithms intolerably.

The use of the 0-1 coding scheme allows for low-contention, deterministic implementations of shared resources. Moreover, other works like the aforementioned paper by Buhr et al. [2], which contains an exhaustive classification and comparison of existing and theoretical flavours of monitors, argue that, priority implementations are, in general, less error-prone. With this in mind, we decided to reuse as much from the existing reference implementation of condition objects and turn them into a concise, priority implementation of monitors. Our proposal is described in the following section.

2 Implementing Priority Monitors on Java's no-Priority Monitors

Figures 2 and 3 show a stripped-down version of the source code for `Monitor.java`, our implementation of priority monitors. Due to space limitations, comments and exception managers have been omitted. A full version, including comments and a couple of extra operations can be found at http://babel.ls.fi.upm.es/software/cclib.

In addition to the `Monitor` class, `Monitor.java` defines the `Cond` class. Both classes are intended to be a replacement for the original `Lock` and `Condition` classes. Figure 3 contains the code for class `Cond`, and the rest of `Monitor.java` is shown in Figure 2. However, this separation is a mere convenience for displaying the code: the implementation of class `Cond` requires access to the state variables in class `Monitor`.

The functionality provided by the class is similar to that of locks and conditions. Method `enter()` provides exclusive access to the monitor, like method `lock()` in class `Lock`. If one thread got exclusive access to a monitor object m, susbsequent calls to m`.enter()` from other threads will force these to wait in m's queue. The monitor is released by invoking m`.leave()` (analogous to `unlock()`). If there are threads waiting at m's entry queue, executing m`.leave()` will give control to the first thread in the queue.

Condition queues associated with m are created by invoking m`.newCond()` rather than calling the constructor of class `Cond`. This is similar to the behaviour of `newCondition()` in class `Lock`. Instances $c, c' \ldots$ of class `Cond` associated with a given monitor object m are managed like condition variables. A thread that invokes c`.await()` after m`.enter()` is blocked unconditionally and put to wait in a queue associated with c. Also, executing c`.await()` releases m, so that other threads can get exclusive access to m by executing m`.enter()`. As in the case of `leave()`, if there are threads waiting at m's entry queue, executing m`.await()` will give control to the first thread in queue.

A thread that executes c`.signal()` after getting exclusive access to m will continue executing (signal-and-continue policy) but it is guaranteed that if there is a thread waiting on c, it will be awakened *before* any process waiting on m's entry queue. This is where the behavior differs from that of signals in the standard locks and conditions package, where signalled threads are requeued at the tail of the lock's entry queue.

The implementation is done on top of the existing implementation of locks and conditions, without reimplementing their functionality say, by using low-level concurrency mechanisms such as semaphores, nor accessing the Java runtime in order to "magically move" threads from one queue to another.

```
package es.upm.babel.cclib;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;

public class Monitor {
   private Lock mutex = new ReentrantLock();
   private Condition purgatory = mutex.newCondition();
   private int inPurgatory = 0;
   private int pendingSignals = 0;

   public Monitor() {
   }

   public void enter() {
      mutex.lock();
      if (pendingSignals > 0 || inPurgatory > 0) {
         inPurgatory++;
         try { purgatory.await(); }
         catch (InterruptedException e) { }
         inPurgatory--;
      }
   }

   public void leave() {
      if (pendingSignals == 0 && inPurgatory > 0) {
         purgatory.signal();
      }
      mutex.unlock();
   }

   public Cond newCond() {
      return new Cond();
   }

   inner class Cond goes here.
}
```

Fig. 2. Java source for priority monitors (class `Monitor`).

```
public class Cond {
   private Condition condition;
   private int waiting;
   private Cond() {
      condition = mutex.newCondition();
      waiting = 0;
   }

   public void await() {
      waiting++;
      if (pendingSignals == 0 && inPurgatory > 0 ) {
         purgatory.signal();
      }
      try { condition.await(); }
      catch (InterruptedException e) { }
      pendingSignals--;
   }

   public void signal() {
      if (waiting > 0) {
         pendingSignals++;
         waiting--;
         condition.signal();
      }
   }
}
```

Fig. 3. Java source for priority condition variables (class `Cond`).

The technique used to simulate the effect of moving signalled threads from the head of the condition queue to the head (rather than the tail) of the monitor's entry queue is to *flush* the contents of this until the signalled thread becomes the first in queue. This is achieved by letting some threads get in the monitor, but only to make them await in a special condition queue devised for that purpose, and which we call "the purgatory". Of course, these threads will have to be eventually awakened and allowed to get access to the monitor in the right sequence. As they have been moved to a condition variable, signalling them will bring them back to the lock's entry queue, so a little care is needed to ensure that the whole thing progresses appropriately. A couple of global counters are responsible for this.

Variable `inPurgatory` counts the number of threads that have been sent to the `purgatory` condition queue and have not been given access to the monitor yet – even if they have already been signalled. Variable `pendingSignals` counts how many threads that have been signalled (in regular condition variables, not `purgatory`) are yet waiting for re-entry.

Method `enter()` starts executing `lock()` on the monitor's internal lock object, called `mutex` (line 16). However, if `pendingSignals` is nonnull, that means that this thread should give way to some signalled thread that was waiting for monitor re-entry later in the queue associated with `mutex`, and thus has to execute a `purgatory.await()`, updating counter `inPurgatory` before and after (lines 17–22). The same is done if other threads have been sent to the purgatory earlier and have re-entered yet (`inPurgatory` > `0`).

The implementation of method `leave()` is quite symmetric. It ends by invoking `mutex.unlock()` (line 29), but before, a chance to re-gain access to the monitor must be given to threads moved to the purgatory (line 27), only if there are no signalled threads pending monitor re-entry (line 26).

Method `newCond` is implemented just by invoking the standard constructor of class `Cond` (line 33). Association to a given monitor object is just implicit in the visibility of the monitor's state variables that the `Cond` object has. A `Cond` object is basically a pair of a `Condition` object associated with `mutex` and a counter `waiting` that keeps track of the number of threads waiting on it (lines 38–43).

The core of method `await()` is the corresponding call on its condition object (line 50) but, before, counter `waiting` is incremented (line 46) and the right to execute inside the monitor is given to threads in the purgatory only if there are no pending signals (lines 47–49). Finally, if the thread that has invoked `await()` reaches line 52, that means that the whole signalling procedure has been successful including monitor re-entry, so `pendingSignals` must be decremented accordingly.

Finally, method `signal()` checks whether the queue size is nonzero (line 56) and if so increments `pendingSignals` (line 57), as monitor re-entry for the signalled thread will be incomplete, decrements `waiting` (line 58) and executes `signal()` on the condition object (line 59). Executing `signal()` on an empty queue has no effect.

Although the implementation of the `Monitor` and `Cond` classes is quite concise, the interplay between the different threads and synchronization barriers, governed by the different counters is, admittedly, complex and hard to follow. After testing the class on a number of applications specifically devised to force errors related to incoming threads getting in the way of signalled ones, we still felt that a more convincing proof of the correctness of our implementation was necessary.

3 A Formal Model of Java Locks and Conditions

As a first step towards a formal argumentation of the correctness of our implementation of priority monitors, we decided to define a transition model for the underlying implementation of locks and condition variables.

The model has been defined as a network of (parametric) automata in Uppaal [1]: one automaton for modelling the concurrency mechanism, i.e., the lock object and its associated conditions, and one automaton per thread that makes use of it. While we give a complete formalization of the concurrency mechanism, unrelated thread code is abstracted away by just showing the interaction with the lock and the conditions.

Each thread has its own thread identifier, *pid*, and so do condition variables (*cid*). All interaction between the different automata takes place via communication channels:

- `lock[`*pid*`]` is the channel the thread *pid* uses when it needs to lock the lock.
- `await[`*cid*`][`*pid*`]` is the channel the thread *pid* uses when it needs to wait in condition *cid*.
- `signal[`*cid*`][`*pid*`]` is the channel a thread uses to signal condition *cid*.
- `lock_granted[`*pid*`]` is the channel the concurrency mechanism uses to grant a lock to the thread *pid*.

Let's start with the abstract model for the threads, since, in our view, it is the most didactic way to present the whole model.

3.1 Modelling the Threads

Each thread has its own thread identifier, *pid*, and the model of a thread is the automaton (actually an Uppaal template parametrised by thread identifiers) shown in Figure 4.

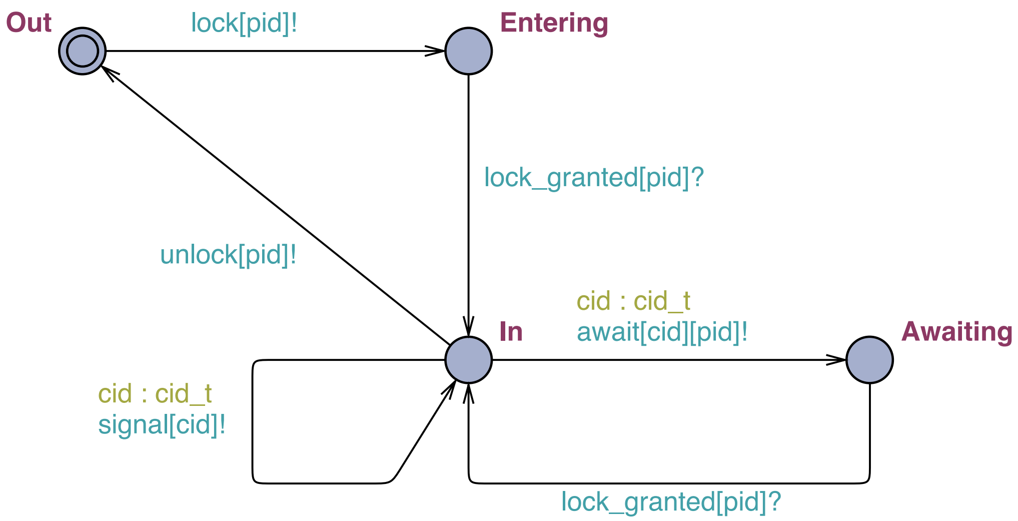


Fig. 4. State model for a Java thread invoking operations on locks.

The automaton has four locations that represent the state of a thread with respect to a lock:

- Out is the location that represents that the thread neither has got the lock nor is waiting in a condition. It is an abstract representation of any program point of any arbitrary thread *outside* of the monitor.

- In is the location that represents that the thread has the lock and it is not blocked neither trying to lock nor waiting in a condition. It is another abstract representation, this time, of any program point of any arbitrary thread *in* the monitor.
- Entering is the location that represents that the thread is blocked while waiting for the lock to be granted to it. From the monitor perspective, it is trying to enter the monitor.
- Awaiting is the location that represents that the thread is blocked in a condition. From the monitor perspective, it is waiting for a signal that allows it to re-enter the monitor.

Edges are extremely simple. The idea behind this simplicity is trying to *contaminate* the thread models as little as possible. They do not involve access to variables, neither global nor local so adapting the model to any particular thread is immediate. Let us see the intended meaning of the actions at the edges:

- Out–Entering with action `lock[`*pid*`]!`, a send-action that indicates that the thread needs the lock. As we will see, the co-action (`lock[`*pid*`]?`) is continuously enabled in the model of the mechanism.
- Entering–In with action `lock_granted[`*pid*`]?`, a receive-action that indicates the thread is allowed to get the lock. The co-action will occur when the mechanism decides that the thread *pid* has the *top* priority to lock the monitor.
- In–Awaiting with action `await[`*cid*`][`*pid*`]!`, a send-action that indicates that the thread wants to wait in the condition variable *cid*.[1]
- Awaiting–In with action `lock_granted[`*pid*`]?`, a receive-action that indicates the thread is allowed to re-take the lock (re-enter the monitor).
- In–Out with action `unlock[`*pid*`]!`, a send-action that indicates that the thread wants to release the lock (leave the monitor).
- In–In with action `signal[`*cid*`]!`, a send-action that indicates that the thread wants to signal a thread on the condition variable *cid*.

3.2 Modelling Java Locks and Conditions

The model of a lock object and its associated condition variables is shown in Figure 5. The main task of the automaton is to keep track of threads by *listening* for their actions (on channels `locks`, `awaits` and `signals`) and responding appropriately (on channel `lock_granted`).

To keep track of the threads, the automaton defines several local variables:

- A thread identifier, `locking_pid`, to represent which is the locking thread, if any.
- A bounded queue of thread identifiers, `waiting`, that represent the queue for getting access to the monitor.

[1] In Uppaal, expressions such as `cid : cid_t` non-deterministically bind the identifier *cid* to a value in the range `cid_t`. It can be understood as an edge *replicator*.

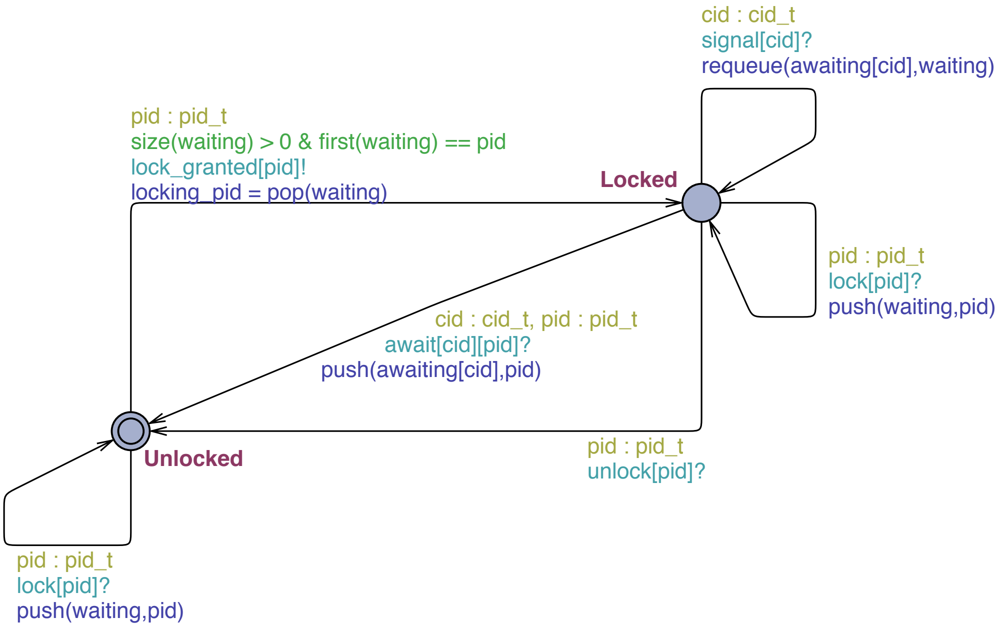


Fig. 5. State model for a Java lock.

– A bounded queue per condition *cid* of thread identifiers, `awaiting[`*cid*`]`, that represent the condition variable queues.

The automaton has two locations, Locked and Unlocked, that represent that the lock is locked by a thread or unlocked, respectively.

Most logic is encoded in the edges. To explain this logic we will explore the actions that fire them. We have to take into account that edges are *indexed* by thread identifiers (*pid*) and condition variable identifiers (*cid*).

– Edges with the receive-action `lock[`*pid*`]?` do not change locations, are always enabled and their assignments just push the thread identifier index *pid* in the access queue `waiting`.
– The edge Locked–Unlocked with the receive-action `await[`*cid*`][`*pid*`]?` is always enabled and the assignment just pushes the thread identifier index *pid* in the condition variable queue `awaiting[`*cid*`]`.
– The edge Locked–Locked with the receive-action `signal[`*cid*`][`*pid*`]?` is always enabled and the assignment just moves the first thread identifier index *pid* from the condition queue `awaiting[`*cid*`]` to the access queue `waiting`.
– The edge Unlocked–Locked with index *pid* is just enabled when the access queue `waiting` is not empty and its first thread identifier is *pid*. The send-action of this edge is `lock_granted[`*pid*`]!`, granting the access to the thread with identifier *pid*.
– The edge Locked–Unlocked with the receive-action `unlock[`*pid*`]!` is just enabled when the identifier of the *asking* thread coincides with the identifier in the variable `locking_pid` and has no assignment.

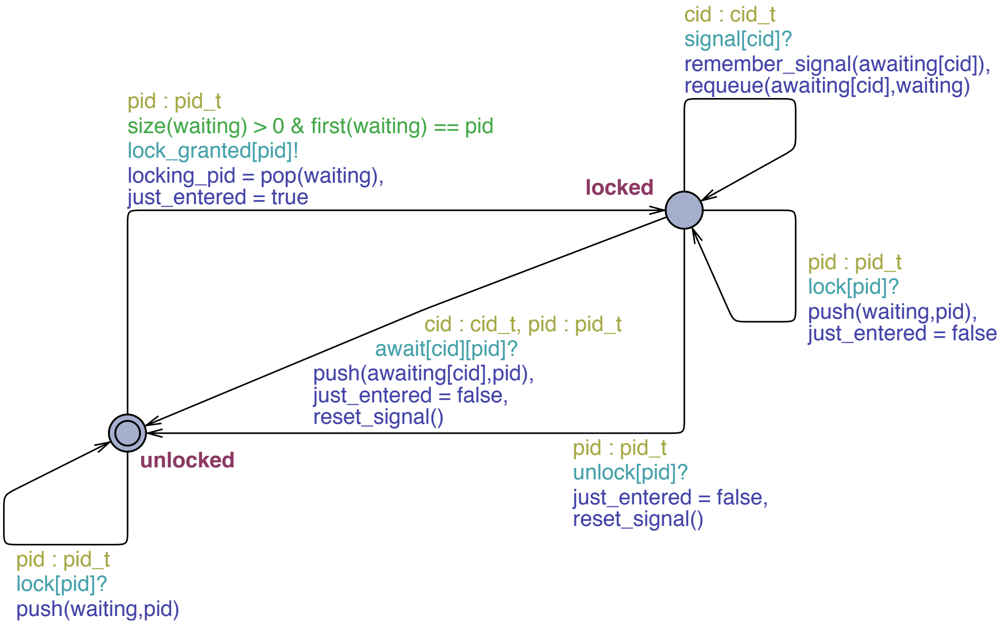


Fig. 6. An instrumented version of the Lock automaton.

The full code for the declarations and auxiliary method definitions in this model can be found under http://babel.ls.fi.upm.es/software/cclib.

3.3 Instrumenting the Model

The natural step after defining this model is to check some properties on it. For us, the most relevant property (correctness aside) is whether signalled threads resume their execution inside the monitor immediately after the signalling thread leaves or not. We will call this the *characteristic* property. As a sanity check, we woud like to check that this property does *not* hold for the model just presented.

However, stating this property in the query language supported by Uppaal is not possible. Basically, the query language is a subset of CTL that only allows temporal modalities as the topmost symbol of formulae, but a query representing the characteristic property for our model would require to nest modal operators.

A workaround for this limitation consists in instrumenting the model so that part of its recent history is somehow *remembered*. This way, we can encode certain temporal properties as state predicates, thus reducing the number of temporal operators required to express the characteristic property.

Again, we avoid contaminating the thread model and Figure 6 shows the resulting lock automaton. Basically, the system has new global variables to store:

- whether the nth thread that got access to the lock did execute an effective signal on some condition, and
- the identifier for the thread that received the signal (if any).

The first is done thanks to a new variable just_signalled, and the second with variable thread_just_awakened. Both of them are set by operation remember_signal while operation reset_signal resets just_signalled once the thread leaving the monitor coincides with thread_just_awakened. Also, the automata for threads are slightly constrained so that at most one signal is allowed per lock.

With these changes, the characteristic property is violated iff the $(n+1)$th thread in gaining access to the lock is different from the thread signalled by the nth thread. In order to avoid having generation counters in the model, we have added yet another boolean variable just_entered to represent that the last transition to take place in the lock automaton is precisely the one that follows a lock_granted message. Violation of the characteristic property can then be encoded in the Uppaal query

$$E\Diamond\,(\mathit{just\_entered} \wedge \mathit{just\_signalled} \wedge \mathit{locking\_pid} \neq \mathit{thread\_just\_awakened})\,.$$

That is, whether there is some path (E path quantifier) that may lead ($\Diamond$ modality) to a state in which the aforementioned proposition holds. The tool finds an example for 3 threads almost immediately, as expected.

4 Verifying our Implementation

A transition model for our priority monitors takes the previous one as starting point. This is quite natural, as our implementation is based on existing locks and conditions. Figure 7 shows the automaton that describes the transitions of the lock (mutex) and condition objects used in the implementation of class Monitor. This is basically the automaton of the previous section (Fig. 5). The only difference is that the special condition purgatory is distinguished from the rest, resulting in the extra edge from locked to itself (upper right corner) and from locked to unlocked. This explicit naming is just necessary for the thread model to refer to the purgatory when necessary.

The model that describes the locations visited by the client threads is shown in Figure 8. The idea is similar to the model for threads in the previous section, but here we have tried to constrain the model a little in order to enforce the intended usage of the Monitor class. That is, threads are assumed to gain exclusive access to the mutex lock first, then they execute 0 or 1 await() calls, then 0 or 1 signal() calls, and finally they give up the monitor by executing leave().

To make the graph easier to follow, most of the locations have been named according to fragments of the source code in Monitor.java. For example, execution of method enter() spans the subgraph from location out to in0, that includes the possibility of a short excursion to the purgatory. Global counters pendingSignals and inPurgatory are faithfully managed in the transitions.

Execution of await() is considered in the locations from in0 to in1. The possibility of not invoking await() is in the following modelled the skip transition that links in0 to in1 directly. Methods signal() and leave() are modelled analogously.

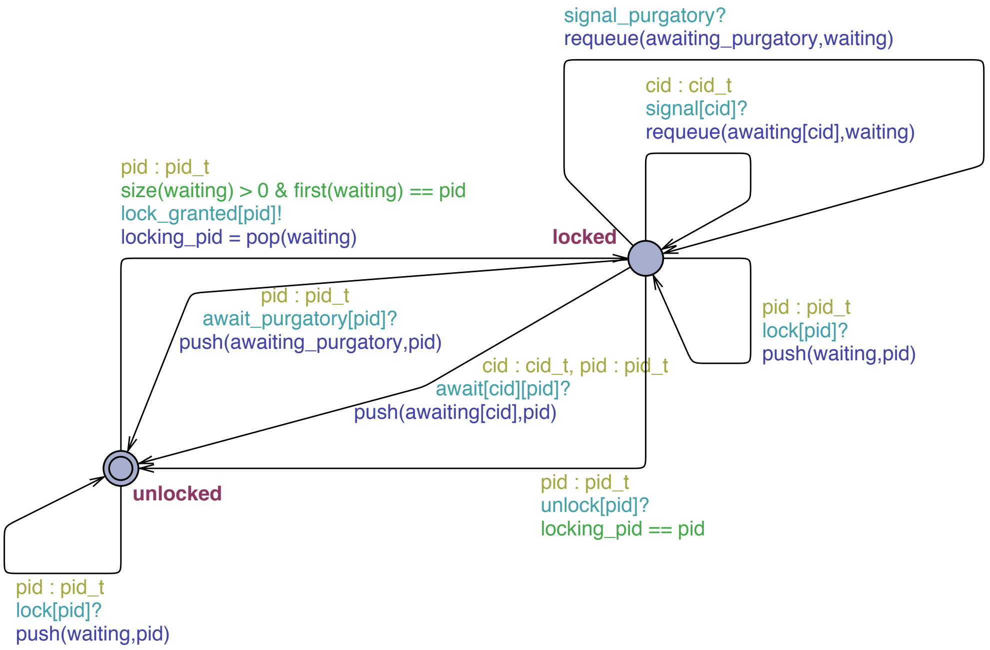


Fig. 7. State model for locks and conditions used in the `Monitor` class implementation.

no. of threads ⇓	1	2	3	4 ⇐ no. of conditions
1	0.01	0.03	0.53	61.0
2	0.02	0.08	138.03	
3	0,02	0.36	501.58	
4	0.02	1.44		
5	0.02	4.64		

Table 1. Times spent in checking the characteristic property of the monitor transition model, for different numbers of threads and condition variables (in seconds).

4.1 Experimental Results

The state model for priority monitors has been instrumented following the ideas in Sec. 3.3. Table 1 shows execution times for checking the characteristic property in the model, given different numbers of client threads and condition queues.[2]

5 Conclusion

We have presented an implementation of nonblocking (signal-and-continue) priority monitors in Java, implemented on top of the existing nonblocking, no-priority implementation in the standard locks & conditions package. Moreover,

[2] Figures obtained on Intel Core2 Duo CPU U9600 1.60GHz, RAM 4Gb, running the Academic version of Uppaal 4.0.11 on a Ubuntu box with kernel Linux 2.6.32-34-generic-pae.

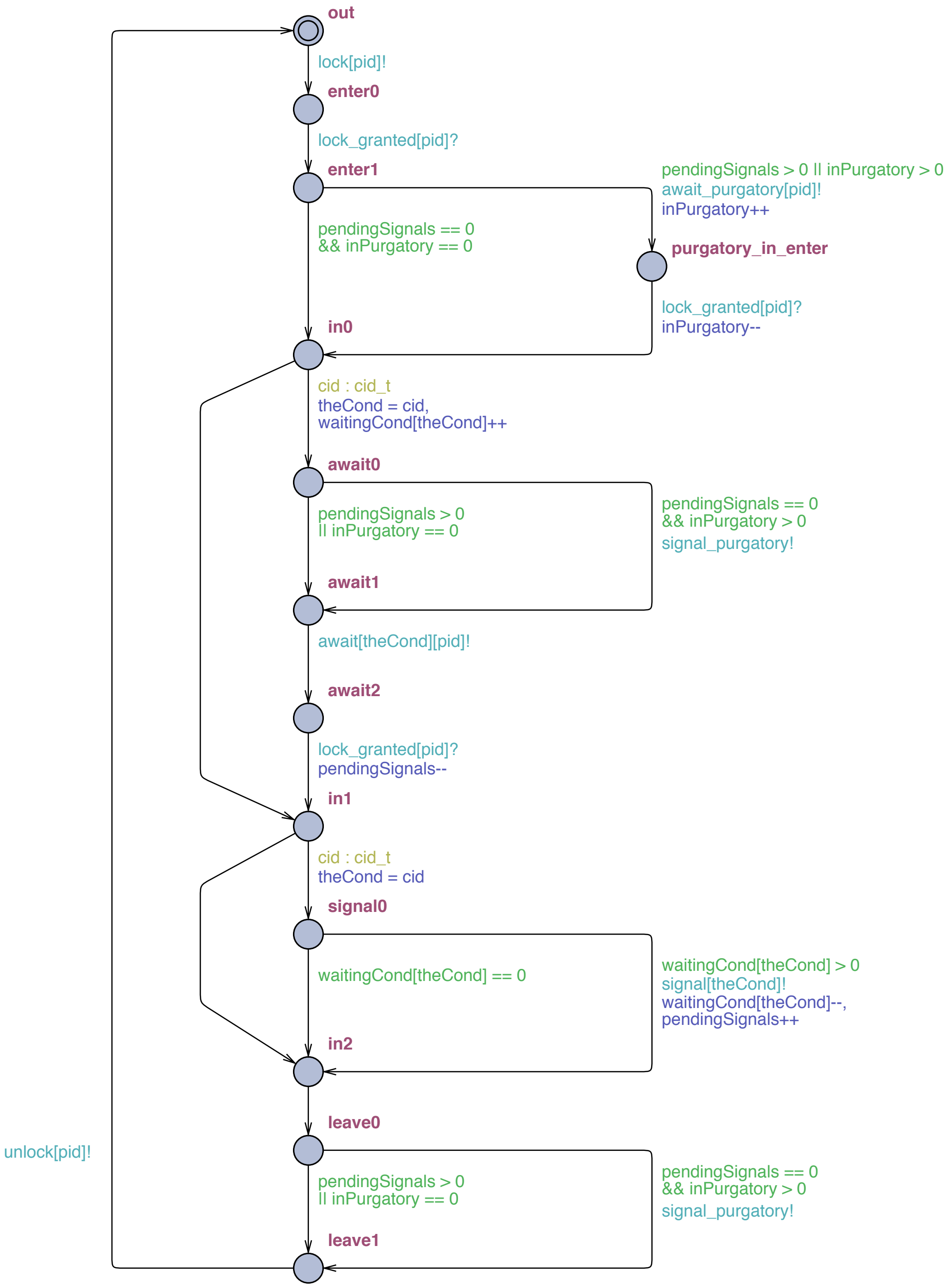


Fig. 8. State model for a thread using the `Monitor` class operations.

we have provided a state model for our solution (extending a model of the existing Java mechanism) that gives *formal evidence* that the priority mechanism is actually implemented by our algorithm.

Our `Monitor` class encourages the use of certain coding styles that cannot be used with the standard Java implementation (i.e., the 0-1 await/signal coding idiom) and that result in a cleaner, safer code, and we are actually using it in the classroom as a replacement of standard locks and conditions.

Also, we think that the techniques used to implement the priority signalling can be interesting *per se* and, possibly, be adapted to other programming languages in case a slightly different behaviour for some concurrency mechanism is needed.

To our knowledge, there are just two other publicly available implementations of priority monitors in Java. The first one, by Chiao, Wu and Yuan [4], is really a language extension devised to overcome the limitations of synchronized methods and *per-object* waitsets – the paper is from 1999 and locks and conditions had not been added to Java yet. The implementation is based on a preprocessor which translates extended Java programs into standard Java code that invokes the methods of some `EMonitor` class in the right sequence. In a more recent work, T.S. Norvell [7] already considers the limitations of Java monitors in the light of the provided locks and conditions library. However, his approach is different from ours, reimplementing the monitor functionality on top of lower level concurrency mechanisms (semaphores) and using explicit queues of threads. The fact that our code is conciser and based on higher-level methods has facilitated the use of model-checking as a validation tool, while his implementation is not verified.[3]

The model-checking approach was chosen to provide a quick feedback on our solution in spite that we were aware of the double parametricity (no. of threads and no. of condition queues) of the problem and, to be honest, the experience has been harder than expected.

On one hand, there is one intrinsic issue with the use of model-checking versus, say, traditional program verification. The need of human intervention to abstract away the code and convert it into a transition system (often referred to as “cooking”) always leaves a feeling that there might be some detail lost in translation, even if the model tries to mimic the code as much as possible. Besides, in this case, the limitations in the query language added an additional instrumentation stage to the process. Also, the experiments show an exponential state explosion that happens much earlier than expected, although we are working to improve the figures.

For the sake of conciseness, some details of the reference implementation of Java locks have been ommitted in our model, namely the possibility of spurious locks and reentrant locking. Regarding the first, it is fairly reasonable to assume the implementation of class `ReentrantLock` free of spurious wakeups as, in the most liberal interpretation of the API specification, spurious wakeups would make any attempt at formal reasoning useless. Regarding the second, it can

[3] To be fair, Norvell’s implementation is longer also because he extends the functionality of Java monitors in other ways.

be seen as a benefit of the model-driven approach: we do not consider features outside the intended idioms.

Our intention is to work on a proof of the characteristic property of our algorithm using deductive verification as early as possible in order to compare the benefits and problems of both approaches.

Acknowledgments. We are very grateful to the anonymous reviewers for their comments on an earlier version of this paper.

References

1. Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–237. Springer Verlag, 2004.
2. Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor classification. *ACM Computing Surveys*, 27:63–107, 1995.
3. Manuel Carro, Julio Mariño, Ángel Herranz, and Juan José Moreno-Navarro. Teaching how to derive correct concurrent programs (from state-based specifications and code patterns). In C.N. Dean and R.T. Boute, editors, *Teaching Formal Methods, CoLogNET/FME Symposium, TFM 2004, Ghent, Belgium*, volume 3294 of *LNCS*, pages 85–106. Springer, 2004. ISBN 3-540-23611-2.
4. Hsin-Ta Chiao, Chi-Houng Wu, and Shyan-Ming Yuan. A more expressive monitor for concurrent Java programming. In Arndt Bode, Thomas Ludwig, Wolfgang Karl, and Roland Wismüller, editors, *Euro-Par 2000 Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 1053–1060. Springer Berlin / Heidelberg, 2000.
5. Per Brinch Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34:38–45, 1999.
6. Ángel Herranz, Julio Mariño, Manuel Carro, and Juan José Moreno-Navarro. Modeling concurrent systems with shared resources. In *Formal Methods for Industrial Critical Systems, 14th International Workshop, FMICS 2009, Eindhoven, The Netherlands, November 2-3, 2009. Proceedings*, volume 5825 of *Lecture Notes in Computer Science*, pages 102–116, 2009.
7. Theodore S. Norvell. Better monitors for Java, October 2007. http://world.com/javaworld/jw-10-2007/jw-10-monitors.html.

Scheduler-Specific Confidentiality for Multi-Threaded Programs and Its Logic-Based Verification

Marieke Huisman and Tri Minh Ngo

University of Twente, Netherlands
Marieke.Huisman@ewi.utwente.nl
tringominh@gmail.com

Abstract. Observational determinism has been proposed in the literature as a way to ensure confidentiality for multi-threaded programs. Intuitively, a program is observationally deterministic if the behavior of the public variables is deterministic, i.e., independent of the private variables and the scheduling policy. Several formal definitions of observational determinism exist, but all of them have shortcomings; for example they accept insecure programs or they reject too many innocuous programs. Besides, the role of schedulers was ignored in all the proposed definitions. A program that is secure under one kind of scheduler might not be secure when executed with a different scheduler. The existing definitions do not ensure that an accepted program behaves securely under the scheduler that is used to deploy the program.

Therefore, this paper proposes a new formalization of scheduler-specific observational determinism. It accepts programs that are secure when executed under a specific scheduler. Moreover, it is less restrictive on harmless programs under a particular scheduling policy. We discuss the properties of our definition and argue why it better approximates the intuitive understanding of observational determinism.

In addition, we discuss how compliance with our definition can be verified, using model checking. We use the idea of self-composition and we rephrase the observational determinism property for a single program C as a temporal logic formula over the program C executed in parallel with an independent copy of itself. Thus two states reachable during the execution of C are combined into a reachable program state of the self-composed program. This allows to compare two program executions in a single temporal logic formula. The actual characterization is done in two steps. First we discuss how stuttering equivalence can be characterized as a temporal logic formula. Observational determinism is then expressed in terms of the stuttering equivalence characterization. This results in a conjunction of an LTL and a CTL formula, that are amenable to model checking.

1 Introduction

The success of applications, such as e.g. Internet banking and mobile code, depends for a large part on the kind of confidentiality guarantees that can be given to clients. Using formal means to establish confidentiality properties of such applications is a promising approach. Of course, there are many challenges related to this. Many systems for which confidentiality is important are implemented in a multi-threaded fashion. Thus, the outcome of such programs depends on the scheduling policy. Moreover, because of the interactions between threads and the exchange of intermediate results, also intermediate states can be observed. Therefore, to guarantee confidentiality for multi-threaded programs, one should consider the whole execution traces, i.e., the sequences of states that occur during program execution.

In the literature, different definitions of confidentiality are proposed for multi-threaded programs. This paper follows the approach advocated by Roscoe [11] that the behavior that can be observed by an attacker should be deterministic. To capture this formally, the notion of *observational determinism* has been introduced. Intuitively, observational determinism expresses that a multi-threaded program is secure when its *publicly observable traces* are independent of its confidential data, and independent of the scheduling policy [16]. Several formal definitions are proposed [16, 7, 14], but none of these capture exactly this intuitive definition.

The first formal definition of observational determinism was proposed by Zdancewic and Myers [16]. It states that a program is observationally deterministic iff given any two initial stores s_1 and s_2 that are indistinguishable *w.r.t.* the low variables[1], any two low location traces are equivalent upto stuttering and prefixing, where a low location trace is the projection of a trace into a single low variable location. Zdancewic and Myers consider the trace of each low variable separately. Zdancewic and Myers also argue that prefixing is sufficiently strong equivalence relation, as this only causes external termination leaks of one bit of information [16].

In 2006, Huisman, Worah and Sunesen showed that allowing prefixing of low location traces can reveal more secret information — instead of just one bit of information — even for sequential programs. They strengthened the definition of observational determinism by requiring that low location traces must be stuttering equivalent [7]. In 2008, Terauchi showed that an attacker can observe the relative order of two updates of the low variables in traces, and derive secret information from this [14]. Therefore, he proposed another variant of observational determinism, requiring that all low store traces — which are the projection of traces into a store containing only all low variables — should be stuttering and prefixing equivalent, thus not considering the variables independently.

However, Terauchi's definition is also not satisfactory. This is for several reasons: first of all, the definition still allows an accepted program to reveal secret information, and second, it rejects too many innocuous programs because it requires the complete low store to evolve in a deterministic way.

[1] For simplicity, we consider a simple two-point security lattice, where the data is divided into two disjoint subsets H and L, containing the variables with high (private) and low (public) security level, respectively.

In addition, the fact that a program is secure under a particular scheduler does not imply that it is secure under another scheduler. All definitions of observational determinism proposed so far implicitly assume a non-deterministic scheduler, and might accept programs that are not secure when executed with a different scheduler. Therefore, in this paper, we propose a definition of scheduler-specific observational determinism that overcomes these shortcomings. This definition accepts only secure programs and rejects fewer secure programs under a particular scheduling policy. It essentially combines the previous definitions: it requires that for any low variable, the low location traces from initial stores s_1 and s_2 are stuttering equivalent. However, it also requires that for any low store trace starting in s_1, there *exists* a stuttering equivalent low store trace starting in s_2. Thus, any difference in the relative order of updates is coincidental, and no information can be deduced from it. This existential condition strongly depends on the scheduler used when the program is actually deployed, because traces model possible runs of a program under that scheduling policy. In addition, we also discuss the properties of our formalization. Based on the properties, we argue that our definition better approximates the intuitive understanding of observational determinism, which unfortunately cannot be formalized directly.

Of course, we also need a way to verify adherence to our new definition. A common way to do this for information flow properties is to use a type system. However, such a type-based approach is insensitive to control flow, and rejects many secure programs. Therefore, recently, self-composition has been advocated as a way to transform the verification of information-flow properties into a standard program verification problem [3, 1]. We exploit this idea in a similar way as in our earlier work [7, 5] and translate the verification problem into a model checking problem over a model that executes the program to be verified twice, in parallel with itself. We show that our definition can be characterized by a conjunction of an LTL [8] and a CTL [8] formula. For both logics, good model checkers exist that we can use to verify the information flow property. The characterization is done in two steps: first we characterize stuttering equivalence, and prove correctness of this characterization, and second we use this to characterize our definition of observational determinism.

The rest of this paper is organized as follows. After the preliminaries in Section 2, Section 3 formally discusses the existing definitions of observational determinism and illustrates their shortcomings on several examples. Section 4 gives our new formal definition of scheduler-specific observational determinism, and discusses its properties. The two following sections discuss verification of this new definition. Finally, Section 7 draws conclusions, and discusses related and future work.

2 Preliminaries

This section presents the formal background for this paper. It describes syntax and semantics of a simple programming language, and formally defines equivalence upto stuttering and prefixing.

2.1 Programs and Traces

We present a simple while-language, extended with parallel composition $\|$, i.e., $C\|C'$ where C and C' are two *threads* which can contain other parallel compositions. A thread is a unit of commands that can be scheduled by an scheduler. The program syntax is not used in subsequent definitions, but we need it to formulate our examples. Programs are defined as follows, where v denotes a variable, E a side-effect free expression involving numbers, variables and binary operators, b a Boolean expression, and ϵ the empty (terminated) program.

$$\begin{aligned} C \ ::= &\ \texttt{skip} \mid v := E \mid C; C \mid \texttt{while}\,(b)\,\texttt{do}\ C \mid \\ &\ \texttt{if}\ (b)\ \texttt{then}\ C\ \texttt{else}\ C \mid C\|C \mid \epsilon \end{aligned}$$

Parallel programs communicate via shared variables in a global store. For simplicity, we assume that assignments and lookups are atomic, thus data races (where two variable accesses can occur simultaneously) cannot happen, and we can assume an interleaving semantics (cf. [4]). We also do not consider procedure calls, local memory or locks. These could be added to the language but this would not essentially change the technical results.

Let *Conf*, *Com*, and *Store* denote the sets of *configurations*, *programs*, and *stores*, respectively. A configuration $c = \langle C, s\rangle \in \mathit{Conf}$ consists of a program $C \in \mathit{Com}$ and a store $s \in \mathit{Store}$, where C denotes the program that remains to be executed and s denotes the current program store. A store is the current state of the program memory, which is a map from program variables to values. Let L be a set of low variables. Given a store s, we use $s_{|L}$ to denote the restriction of the store where only the variables in L are defined. We say stores s_1 and s_2 are *low-equivalent*, denoted $s_1 =_L s_2$, iff $s_{1\,|L} = s_{2\,|L}$, i.e., the values of all variables in L in s_1 and s_2 are the same.

The small step operational semantics of our program language is standard. Individual transitions of the operational semantics are assumed to be atomic. As an example, we have the following rules for parallel composition (with their usual counterparts for C_2):

$$\frac{\langle C_1, s_1\rangle \to \langle \epsilon, s_1'\rangle}{\langle C_1 \parallel C_2, s_1\rangle \to \langle C_2, s_1'\rangle} \quad \frac{\langle C_1, s_1\rangle \to \langle C_1', s_1'\rangle \quad C_1' \neq \epsilon}{\langle C_1 \parallel C_2, s_1\rangle \to \langle C_1' \parallel C_2, s_1'\rangle}$$

We also have a special transition step for terminated programs, i.e., $\langle \epsilon, s\rangle \to \langle \epsilon, s\rangle$, ensuring that all traces are infinite. Thus, we assume that the attacker cannot detect termination.

A multi-threaded program executes threads from the set of live threads, i.e., the set of not-yet terminated threads. During the execution, a scheduling policy repeatedly decides which threads can be picked to proceed next with the computation. Different scheduling policies differ in how they make this decision, e.g., a nondeterministic scheduler chooses threads randomly and hence all possible interleavings of threads are potentially enabled; and a *round-robin* scheduler assigns equal time slices to each thread in circular order. Given scheduling policy δ, and configuration $\langle C, s\rangle$, an infinite list of configurations $T = c_0c_1c_2...$

$(T : \mathbb{N}_0 \to \mathit{Conf})$ is a *trace* of the execution of C from s under the control of δ, denoted $\langle C, s\rangle \Downarrow_\delta T$, iff $c_0 = \langle C, s\rangle$ and $\forall i \in \mathbb{N}_0.\ c_i \to c_{i+1}$ under δ. We simply write $\langle C, s\rangle \Downarrow T$ when the scheduler is nondeterministic.

Let T_i, for $i \in \mathbb{N}$, denote the i^{th} element in the trace, i.e., $T_i = c_i$. We use $T_{\ll i}$ to denote the *prefix* of T upto the index i, i.e., $T_{\ll i} = T_0 T_1 \ldots, T_i$. When appropriate, $T_{\ll i}$ can be considered as an infinite trace stuttering in T_i forever. Further, we use $T_{|_L}$ to denote the projection of a trace to a store containing only the variables in L. Formally: $T_{|_L} = \mathit{map}(_{-|_L} \circ \mathit{store})(T)$, where map is the standard higher-order function that applies $(_{-|_L} \circ \mathit{store})$ to all elements in T. When L is a singleton set $\{l\}$, we simply write $T_{|_l}$. Finally, in the examples below, when writing an infinite trace that stutters forever from state T_i onwards, we just write this as a finite trace $T = [T_0, T_1, \ldots, T_{i-1}, T_i]$.

2.2 Stuttering and Prefixing Equivalences

The key ingredient in the different definitions of observational determinism is the equivalence of traces upto stuttering or upto stuttering and prefixing. The definition of stuttering equivalence is based on [10, 7]. It uses an auxiliary notion of *stuttering equivalence upto indexes i and j*.

Definition 1 (Stuttering equivalence). *Traces T and T' are* stuttering equivalent upto i and j*, written $T \sim_{i,j} T'$, iff we can partition $T_{\ll i}$ and $T'_{\ll j}$ into n blocks such that elements in the p^{th} block of $T_{\ll i}$ are equal to each other and also equal to elements in the p^{th} block of $T'_{\ll j}$ (for all $p \leq n$). Corresponding blocks may have different lengths.*

Formally, $T \sim_{i,j} T'$ iff there are sequences $0 = k_0 < k_1 < k_2 < \ldots < k_n = i + 1$ and $0 = g_0 < g_1 < g_2 < \ldots < g_n = j + 1$ such that for each $0 \leq p < n$ holds: $T_{k_p} = T_{k_p+1} = \cdots = T_{k_{p+1}-1} = T'_{g_p} = T'_{g_p+1} = \cdots = T'_{g_{p+1}-1}$.

T and T' are stuttering equivalent*, denoted $T \sim T'$, iff $\forall i.\ \exists j.\ T \sim_{i,j} T' \wedge \forall j.\ \exists i.\ T \sim_{i,j} T'$.*

Stuttering equivalence defines an equivalence relation, i.e., it is reflexive, symmetric and transitive.

Equivalence upto stuttering and prefixing is defined as one trace being stuttering equivalent to a prefix of the other trace.

Definition 2 (Prefixing and stuttering equivalence). *Traces T and T' are* prefixing and stuttering equivalent*, written $T \sim_p T'$, iff $\exists i. T \sim T'_{\ll i} \vee T_{\ll i} \sim T'$.*

3 Observational Determinism in the Literature

This section presents the existing definitions of observational determinism formally, and discusses their shortcomings. The next section presents our improved definition.

3.1 Existing Definitions of Observational Determinism

Given any two initial low equivalent stores, $s_1 =_L s_2$, a program C is *observationally deterministic*, according to

- Zdancewic and Myers [16]: iff any two low location traces are equivalent upto stuttering and prefixing, i.e., $\forall T, T'.\ \langle C, s_1 \rangle \Downarrow T \wedge \langle C, s_2 \rangle \Downarrow T' \Rightarrow\ \forall l \in L.\ T_{|_l} \sim_p T'_{|_l}$.
- Huisman et al. [7]: iff any two low location traces are equivalent upto stuttering, i.e., $\forall T, T'.\ \langle C, s_1 \rangle \Downarrow T \wedge \langle C, s_2 \rangle \Downarrow T' \Rightarrow\ \forall l \in L.\ T_{|_l} \sim T'_{|_l}$.
- Terauchi [14]: iff any two low store traces are equivalent upto stuttering and prefixing, i.e., $\forall T, T'.\ \langle C, s_1 \rangle \Downarrow T \wedge \langle C, s_2 \rangle \Downarrow T' \Rightarrow\ T_{|_L} \sim_p T'_{|_L}$.

Notice that the existing definitions all have implicitly assumed a nondeterministic scheduler, without mentioning this explicitly.

Zdancewic and Myers, followed by Terauchi, allow equivalence upto prefixing. This has as an advantage that it removes the obligation to consider program termination. The definition of Huisman et al. is stronger than the one of Zdancewic and Myers, as it only allows stuttering equivalence. Both definitions of Zdancewic and Myers, and Huisman et al. only specify equivalence of traces on each single low location separately, they do not consider the relative order of variable updates in traces, while Terauchi does. In particular, Terauchi's definition is stronger than Zdancewic and Myers' definition as it requires equivalence upto stuttering and prefixing on low store traces instead of on low location traces.

3.2 Shortcomings of These Definitions

Unfortunately, all these definitions have shortcomings. Huisman et al. showed that allowing prefixing of low location traces, as in the definition of Zdancewic and Myers, can reveal secret information, see [7]. Further, as observed by Terauchi, attackers can derive secret information from the relative order of updates, see [14]. It is not sufficient to require that only the low location traces are deterministic for a program to be secure. Therefore, Terauchi required that all low store traces should be stuttering and prefixing equivalent. However, allowing prefixing of full low store traces still can reveal secret information. Besides, the requirement that traces have to agree on updates to all the low locations as a whole, as in Terauchi's definition, is overly restrictive. In addition, all these definitions accept programs that behave insecurely under some specific schedulers. These shortcomings are illustrated below by several examples. In all examples, we assume an observational model is where attackers can access the full code of the program, observe the traces of public data, and limit the set of possible program traces by choosing a scheduler.

How prefixing equivalences can reveal information Consider the following program. Suppose $\mathtt{h} \in H$ and $\mathtt{l1}, \mathtt{l2} \in L$, $\mathtt{h}$ is a Boolean.

Example 1.

```
l1 := 0; l2 := 0;
{if (l1 == 1) then (l2 := h) else skip} || l1 := 1
```

For notational convenience, let C_1 and C_2 denote the left and right operands of the parallel composition operator in all examples. A low store trace is denoted by a sequence of low stores, containing the values of the low variables in order, i.e., (`l1`, `l2`). If we execute this program from several low equivalent stores for different values of `h`, we obtain the following low store traces.

$$\text{Case } \mathtt{h} = 0 : T_{|_L} = \begin{cases} [(0,0),(1,0)] & \text{execute } C_1 \text{ first} \\ [(0,0),(1,0),(1,0)] & \text{execute } C_2 \text{ first} \end{cases}$$
$$\text{Case } \mathtt{h} = 1 : T_{|_L} = \begin{cases} [(0,0),(1,0)] & \text{execute } C_1 \text{ first} \\ [(0,0),(1,0),(1,1)] & \text{execute } C_2 \text{ first} \end{cases}$$

According to Zdancewic and Myers, and Terauchi, this program is observationally deterministic. However, when $\mathtt{h} = \mathtt{1}$, we can terminate in a state where $\mathtt{l2} = \mathtt{1}$. It means that when the value of `l2` changes, an attacker can conclude that surely $\mathtt{h} = \mathtt{1}$; partial information still can be leaked because of prefixing.

How too strong conditions reject too many programs The restrictiveness of Terauchi's definition arises from the fact that no variation in the relative order of updates is allowed at all. This rejects many harmless programs, such as for example,

Example 2.

$$\mathtt{l1} := \mathtt{0};\ \mathtt{l2} := \mathtt{0};\ \ \{\mathtt{l1} := \mathtt{3} \parallel \mathtt{l2} := \mathtt{4}\}$$

If C_1 is executed first, we get the following traces, $T_{|_L} = [(0,0),(3,0),(3,4)]$; otherwise, $T_{|_L} = [(0,0),(0,4),(3,4)]$. This program is rejected by Terauchi, because not all low store traces are equivalent upto stuttering and prefixing.

How scheduling policies can be exploited by attackers In all examples given so far, a nondeterministic scheduler is assumed. However, in practice, the scheduler may vary from execution to execution. The security of a program depends strongly on the scheduler's behavior. Under a specific scheduling policy, some traces *cannot occur*. Due to the fact that an attacker knows the full code of the program, when he chooses an appropriate scheduler, secret information can be revealed from the limited set of possible traces. This sort of attack is often called a refinement attack [13, 2], because the choice of scheduling policy refines the set of possible program traces. Consider the following example,

Example 3.

$$\mathtt{l} := \mathtt{0};\ \Big\{\{\{\mathtt{if}\ (\mathtt{h} > \mathtt{0})\ \mathtt{then}\ \mathtt{sleep(n)}\};\ \mathtt{l} := \mathtt{1}\} \parallel \mathtt{l} := \mathtt{0}\Big\}$$

where `sleep(n)` abbreviates `n` consecutive *skip* commands. Under a nondeterministic scheduler, the initial value of `h` cannot be derived; this program is accepted by the definitions of Zdancewic and Myers, and Terauchi.

However, suppose we execute this program using a *round-robin* scheduling policy, i.e., the scheduler picks a thread and then proceeds to run that thread for m steps, before giving control to the next thread. If $m < n$ we obtain store traces of the following shapes.

$$\text{Case } \mathtt{h} \leq 0: \quad T_{|_L} = \begin{cases} [(0),(1),(0)] \text{ execute } C_1 \text{ first} \\ [(0),(0),(1)] \text{ execute } C_2 \text{ first} \end{cases}$$
$$\text{Case } \mathtt{h} > 0: \quad T_{|_L} = \begin{cases} [(0),(0),\ldots,(0),(1)] \text{ execute } C_1 \text{ first} \\ [(0),(0),\ldots,(0),(1)] \text{ execute } C_2 \text{ first} \end{cases}$$

With this scheduling policy, this program is still accepted by Zdancewic and Myers, and Terauchi. However, when $\mathtt{h} \leq 0$, we can terminate in a state where $\mathtt{l} = 0$. Thus, the final value of `l` may reveal whether `h` is positive or not.

Example 4.

```
l1 := 0; l2 := 0;
{if (h > 0) then l1 := 1 else l2 := 1}||{l1 := 1; l2 := 1}||{l2 := 1; l1 := 1}
```

This program is secure under a nondeterministic scheduler, and it is accepted by the definitions of Zdancewic and Myers, and Huisman et al. However, when an attacker chooses a scheduler which always executes the leftmost thread first, he gets only two different kinds of traces, corresponding to the values of `h`: when $\mathtt{h} > 0$, $T_{|_L} = [(0,0),(1,0),(1,1),\ldots]$; otherwise, $T_{|_L} = [(0,0),(0,1),(1,1),\ldots]$.

In this case, this program is still accepted by the definitions of Zdancewic and Myers, and Huisman et al. but this program is not secure anymore. Attackers can learn information about `h` by observing whether `l1` is updated before `l2`. Notice that the problem of relative order of updates was shown in [14].

To conclude, the examples above show that all the existing definitions of observational determinism allow programs to reveal private data because they allow equivalence upto prefixing, as in the definitions of Zdancewic and Myers, and Terauchi, or do not consider the relative order of updates, as in the definitions of Zdancewic and Myers, and Huisman et al. The definition of Terauchi is also overly restrictive, rejecting many secure programs. Moreover, all these definitions are not scheduler-specific. They accept programs behaving insecurely under a specific scheduling policy. This is our motivation to propose a new definition of scheduler-specific observational determinism. This definition on one hand only accepts secure programs, and on the other hand is less restrictive on innocuous programs w.r.t. a particular scheduler.

4 Scheduler-Specific Observational Determinism

To overcome the problems discussed above, we say that a program is observationally deterministic under a particular scheduler if any two low location traces are stuttering equivalent *and* for any low store trace produced from one initial store, there exists a low store trace produced from the other initial low equivalent store such that these two traces are stuttering equivalent. Our definition does not allow information to be leaked because of prefixing equivalence. Notice that Zdancewic and Myers, and Terauchi allow prefixing equivalence because it removes the obligation to prove program termination in their proposed type systems.

Scheduler-specific observational determinism is defined formally as follows.

Definition 3 (δ-specific observational determinism).

*Given a scheduling policy δ, a program C is δ-*specific observationally deterministic w.r.t. *L iff for all initial low equivalent stores s_1, s_2, $s_1 =_L s_2$, the following conditions (1) and (2) are satisfied.*

$$\text{-}\,\forall T, T'.\ \langle C, s_1\rangle \Downarrow_\delta T \wedge \langle C, s_2\rangle \Downarrow_\delta T' \Rightarrow \forall l \in L.\ T_{|_l} \sim T'_{|_l}. \tag{1}$$

$$\text{-}\,\forall T.\ \langle C, s_1\rangle \Downarrow_\delta T. \exists T'.\ \langle C, s_2\rangle \Downarrow_\delta T' \wedge T_{|_L} \sim T'_{|_L}. \tag{2}$$

We require that the low locations individually behave deterministically because in the literature it has been shown how nondeterminism of a low variable can be exploited to make other programs reveal confidential information. Even the simple program "`l := 0` ‖ `l := 1`" can be used to violate confidentiality of another program. If public variables are shared between programs, there exists a channel between them [15]. Suppose that the public variable `l` is shared, i.e., this data is used by another apparently secure program, and access to this data is conditioned on confidential information, then this assignment is more likely to happen last. Therefore, there is a timing channel between two programs and it can be used to derive information about the confidential data, see [16, 15]. Therefore, to be considered secure, a program must enforce an ordering on the accesses to a single low location, i.e., the sequence of operations performed at a single low location is deterministic [16].

However, notice that the program "`l1 := 3` ‖ `l2 := 4`" in Example 2 is considered secure because it writes to two different locations.

Besides, this definition also releases the requirement that all low store traces have to agree on the relative order of updates. Our definition differs from the previous definitions of observational determinism in one important aspect: the existential condition. This condition depends strongly on the scheduling policy used to deploy the program because traces model possible runs of a program and refinements of the set of traces, when the scheduling policy changes, cannot guarantee this condition.

Notice that the execution of a program under a nondeterministic scheduler means that we consider all possible interleavings of threads. Given any scheduling policy δ, the set of possible program traces under δ is a subset of the set of possible program traces under a nondeterministic scheduler. If we quantify Definition 3 over all possible schedulers, it requires that each low store trace produced from one initial store under a nondeterministic scheduler must be matched with every low store trace produced from the other initial store. It means that for any two initial low equivalent stores, if any two low store traces obtained from the execution of a program under a nondeterministic scheduler are stuttering equivalent, this program is secure under any scheduling policy δ. Thus, this gives a truly scheduler-independent definition of observational determinism.

4.1 Properties of Scheduler-Specific Observational Determinism

To illustrate that Definition 3 captures the intended meaning of observational determinism best, we discuss different properties of the definition.
Property 1 (Deterministic low location traces). If a program is accepted by Definition 3, no secret information can be derived from the publicly observable location traces. It is required that the low locations individually evolve deterministically, and thus, the values of private variables may not affect the values of low variables.

Property 2 (Deterministic relative order of updates). If a program is accepted by Definition 3, no information can be derived from the relative order of updates because there is always a matching low store trace.

Notice that the insecure programs in Examples 1 and 3 are rejected by our definition under a nondeterministic scheduling policy. The program in Example 4 is secure under a nondeterministic scheduler and it is accepted by our definition instantiated accordingly. However, it is insecure under the scheduler that always chooses the leftmost thread to execute first; and hence, it is rejected if we instantiate the definition with this scheduler. Thus, given a scheduling policy δ, if a program is accepted by our definition, instantiated for this scheduler, we can conclude that the program is secure under δ.

Property 3 (Less restrictive on harmless programs). Compared with Terauchi's definition, Definition 3 is more permissive: it allows some freedom in the order of individual updates, as long as a matching execution exists.
For example, Example 2 and 4, which are secure, are accepted by our definition instantiated with a nondeterministic scheduler, but rejected by Terauchi.

After having presented an improved definition of observational determinism, the next sections discuss how to verify it formally.

5 A Temporal Logic Characterization of Stuttering Equivalence

5.1 Self-Composition to Verify Information Flow Properties

A common approach to check information flow properties is to use a type system. However, the type-based approach is not suitable to verify Definition 3. First, type systems for multi-threaded programs often aim to prevent secret information from affecting the thread timing behavior of a program, e.g., secret information can be derived from observing the internal timing of actions [16]. For to this reason, the type systems proposed to enforce confidentiality for multi-threaded programs are often very restrictive. This restrictiveness makes the application programming become impractical and many intuitively secure programs are rejected by type systems. Besides, it also seems difficult to enforce stuttering equivalence via type-based methods without being overly restrictive [14]. In addition, type systems are not suitable to verify existential properties, as the one in our definition. This can be understood as follows. If the program C is well-typed, then for any two configurations $c_1 = \langle C, s_1 \rangle$ and $c_2 = \langle C, s_2 \rangle$ such that $s_1 =_L s_2$, there exists a configuration, e.g., c', that simulates both [16]. This means that for any two traces T and T' starting in c_1 and c_2 respectively, the low-deterministic properties of T and T' can be simulated by the same trace starting in c'. In other words, if C is well-typed, two sets of traces starting in c_1 and c_2 have the same low-security behavior. Therefore, our definition, which contains an existential quantification, cannot be verified via type-based methods.

Instead, we use self-composition. This is a recently developed technique [3, 1] that transforms the verification of information flow properties into a verification problem. Self-composition means that we compose a program C with its copy, denoted C', i.e., we execute C and C' in parallel, and consider $C \parallel C'$ as a single program. Notice that C' is program C, but with all variables renamed to make them distinguishable from the variables in C [1]. In this model, the original two programs still can be distinguished, and then we express the information flow property as a property over the executions of the self-composed program.

Concretely, in this paper we characterize observational determinism with a temporal logic formula. The essence of observational determinism is stuttering equivalence of execution traces. Therefore, we first investigate the characteristics of stuttering equivalence and discuss which extra information is needed to characterize this in temporal logic. Based on the idea of self-composition and the extra information, we define a model over which we want the temporal logic formula to hold. After that, a temporal logic formula that characterizes stuttering equivalence is defined. This formula can be instantiated in different ways, depending on the equivalence relation that is used in the stuttering equivalence. Observational determinism is expressed in terms of the stuttering equivalence characterization. This results in a conjunction of an LTL and a CTL formula (for the syntax and semantics definitions of LTL and CTL, see [8]). Both formulas are evaluated over a single execution of the self-composed program. We show that the logic formulas are equivalent to the original definitions, thus the characterization as a model checking problem is sound and complete.

5.2 Characteristics of Stuttering Equivalence

We let symbols $\mathbf{a}, \mathbf{b}, \mathbf{c}$, etc. represent states in traces. Given $T \sim T'$ as follows,

$$\begin{array}{rl}
\text{index:} & {\scriptstyle 0\ 1\ 2\ 3\ 4\ 5\ \ldots} \\
T = & \mathbf{a\,b\,c\,d\,d\,d}\ldots \\
\text{nr of state changes in } T\text{:} & {\scriptstyle 0\ 1\ 2\ 3\ 3\ 3} \\
T' = & \mathbf{a\,a\,b\,b\,c\,d}\ldots \\
\text{nr of state changes in } T'\text{:} & {\scriptstyle 0\ 0\ 1\ 1\ 2\ 3}
\end{array}$$

The top row indicates the indexes of states. The row below each trace indicates the total numbers of state changes, counted from the first state, that happened in the trace. Based on this example, we can make some general observations about stuttering equivalence that form the basis for our temporal logic characterization.

- Any state change that occurs *first* in trace T at index i, i.e., T_i, will also occur later in trace T' at some index $j \geq i$.
- For any index r between such a *first* and *second* occurrence of a state change, i.e., $i \leq r < j$, at state T'_r, the total number of state changes is *strictly smaller* than the total number of state changes at T_r.
- Similarly for any change that occurs *first* in trace T'.

Notice that these properties are sound and complete to characterize stuttering equivalence, see Appendix A.2 of [6].

5.3 Extra Information

To characterize stuttering equivalence in temporal logic, we have to come up with a temporal logic formula over a combined trace. As a convention, we use T^1 and T^2 to denote the two component traces. Thus, the i^{th} state of the combined trace contains both T^1_i and T^2_i. The essence of stuttering equivalence is that any state change occurring in one trace also has to occur in the other trace. Therefore, we have to extend the state with extra information that allows to determine for a particular state (1) whether the current state is different from the previous one, (2) whether a change occurs first or second, and (3) how many state changes have already happened.

How to characterize state change? To determine whether a state change occurred, we need to know the previous state. Therefore, we define a *memorizing transition relation*, remembering the previous state of each transition.

Definition 4 (Memorizing transition relation). *Let* $\rightarrow \subseteq (\mathit{State} \times \mathit{State})$ *be a* transition relation. *The memorizing transition relation* $\rightarrow_m \subseteq (\mathit{State} \times \mathit{State}) \times (\mathit{State} \times \mathit{State})$ *is defined as:* $(c_1, c'_1) \rightarrow_m (c_2, c'_2) \ \Leftrightarrow\ c_1 \rightarrow c_2 \ \wedge\ c'_2 = c_1$.

Thus, (c_1, c'_1) makes a memorizing transition to (c_2, c'_2) if (1) c_1 makes a transition to c_2 in the original system, and (2) c'_2 remembers the old state c_1. We use accessor functions *current* and *old* to access the components of the memorized state, such that $\mathit{current}(c_1, c'_1) = c_1 \ \wedge\ \mathit{old}(c_1, c'_1) = c'_1$.

A state change can now be observed by comparing old and current components of a single state.

How to characterize the order of state changes? To determine whether a state change occurs for the first time or has already occurred in the other trace, we use a queue of states, denoted q. Its contents represents the *difference* between the two traces. We have the following operations and queries on a queue: *add*, adds an element to the end of the queue, *remove*, removes the first element of the queue, and *first*, returns the first element of the queue. In addition, we use an extra state component *lead*, that indicates which component trace added the last state in q, i.e., $\mathit{lead} = m$ $(m = 1, 2)$ if the last element in q was added from T^m. Initially, the queue is empty (denoted ε), and *lead* is 0.

The rules to add/remove a state to/from the queue are the following. Whenever a state change occurs for the first time in T^m, the current state is added to the queue and *lead* becomes m. When this state change occurs later in the other trace, the element will be removed from the queue. When a state change in one trace does not match with the change in the other trace, both q and *lead* become undefined, denoted $\perp$, indicating a blocked queue. If $q = \perp$ (and $\mathit{lead} = \perp$), the component traces are not stuttering equivalent, and therefore we do not have to check the remainders of the traces. Therefore, operations *add* and *remove* are not defined when q and *lead* are $\perp$.

Formally, these rules for adding and removing are defined as follows. Initially, q is ε and *lead* is 0. Whenever $q \neq \perp$ and $T_i^m \neq T_{i-1}^m$ $(m = 1, 2)$,

- if $\mathit{lead} = 3 - m$ and $T_i^m = \mathit{first}(q)$, then $\mathit{remove}(q)$. If $q = \varepsilon$, set $\mathit{lead} = 0$.
- if $\mathit{lead} = m$ or $\mathit{lead} = 0$, then execute $\mathit{add}(q, T_i^m)$ and set $\mathit{lead} = m$.
- otherwise, set $q = \perp$ and $\mathit{lead} = \perp$.

How to characterize the number of state changes? To determine the number of state changes that have happened, we extend the state with counters $\mathit{nr\_ch}^1$ and $\mathit{nr\_ch}^2$. Initially, both $\mathit{nr\_ch}^1$ and $\mathit{nr\_ch}^2$ are 0, and whenever a state change occurs, i.e., $T_i^m \neq T_{i-1}^m$ $(m = 1, 2)$, then $\mathit{nr\_ch}^m$ increases by one. Thus, the number of state changes at T_i^1 and T_i^2 can be determined via the values of $\mathit{nr\_ch}^1$ and $\mathit{nr\_ch}^2$, respectively.

5.4 Program Model

Next we define a model over which a temporal logic formula should hold. Given program C and two initial stores s, s', we take the parallel composition of C and its copy, denoted C', and consider $C \parallel C'$ as a single program. In this model, the store of $C \parallel C'$ can be considered as the product of the two separate stores s and s', ensuring that the variables from the two program copies are disjoint, and thus that updates are done locally, i.e., not affecting the store of the other program copy.

First, we define the elements of the program model.

States: A state of a composed trace is of the form $(\langle C_1 \parallel C_2, (s_1, s_2)\rangle, \langle C_3 \parallel C_4, (s_3, s_4)\rangle, \chi)$, where $\langle C_3 \parallel C_4, (s_3, s_4)\rangle$ remembers the old configuration (via the memorizing transition relation of Definition 4), and χ is extra information, as discussed above, of the form $(\mathit{nr\_ch}^1, \mathit{nr\_ch}^2, q, \mathit{lead})$. We define accessor functions conf_1, conf_2, and *extra* to extract $(\langle C_1, s_1\rangle, \langle C_3, s_3\rangle)$, $(\langle C_2, s_2\rangle, \langle C_4, s_4\rangle)$, and χ, respectively.

Thus, in our model, the original two program copies still can be distinguished and the updates of program copies are done locally. Therefore, if $\mathcal{T}$ is a trace of the composed model, then we can decompose it into two individual traces by functions Π_1 and Π_2, respectively, defined as $\Pi_m = \mathit{map}(\mathit{conf}_m)$. Thus, given a state $\mathcal{T}_i = (\langle C_1 \parallel C_2, (s_1, s_2)\rangle, \langle C_3 \parallel C_4, (s_3, s_4)\rangle, \chi)$ of the composed trace, then $(\Pi_1(\mathcal{T}))_i = (\langle C_1, s_1\rangle, \langle C_3, s_3\rangle)$ and $(\Pi_2(\mathcal{T}))_i = (\langle C_2, s_2\rangle, \langle C_4, s_4\rangle)$. The current configuration of program copy m can be extracted by function Γ_m, defined as $\Gamma_m = \mathit{map}(\mathit{current}) \circ \Pi_m$. Thus, $(\Gamma_1(\mathcal{T}))_i = \langle C_1, s_1\rangle$ and $(\Gamma_2(\mathcal{T}))_i = \langle C_2, s_2\rangle$.

Finally, $\mathit{extra}(\mathcal{T}_i)(x)$ denotes the value of the extra information x at $\mathcal{T}_i$, for $x \in \{\mathit{nr\_ch}^1, \mathit{nr\_ch}^2, q, \mathit{lead}\}$.

Transition Relation: Let $\to$ be the translation relation induced by the operational semantics of programs, and $\to_m$ the memorizing transition relation derived from $\to$ (cf. Definition 4). The transition relation of the program model $\to_\chi$ is defined using $\to_m$, and a relation $\to \subseteq \chi \times \mathit{Conf} \times \chi$ that describes how the extra information evolves, following the rules below (with a similar rule for when C_1 terminates, i.e., $\langle C_1, s_1\rangle \to \langle \epsilon, s_1\rangle$, and the symmetric counterparts for C_2).

$$\frac{(\langle C_1 \parallel C_2, (s_1, s_2)\rangle, c_2) \to_m (\langle C_1' \parallel C_2, (s_1', s_2)\rangle, c_4) \quad \chi \overset{\langle C_1', s_1'\rangle}{\to} \chi'}{(\langle C_1 \parallel C_2, (s_1, s_2)\rangle, c_2, \chi) \to_\chi (\langle C_1' \parallel C_2, (s_1', s_2)\rangle, c_4, \chi')}$$

where $c_4 = \langle C_1 \parallel C_2, (s_1, s_2)\rangle$ and $\chi \overset{c}{\to} \chi'$ is defined as follows (notice that this relation is parametric on the concrete equality relation used).

$$\frac{\mathit{lead} = 2 \quad c = \mathit{first}(q) \quad \mathit{nr\_ch}^{1'} = \mathit{nr\_ch}^1 + 1 \quad q' = \mathit{remove}(q) \quad \mathit{lead}' = 1}{(\mathit{nr\_ch}^1, \mathit{nr\_ch}^2, q, \mathit{lead}) \overset{c}{\to} (\mathit{nr\_ch}^{1'}, \mathit{nr\_ch}^{2'}, q', \mathit{lead}')}$$

$$\frac{\mathit{lead} \in \{0, 1\} \quad \mathit{lead}' = 1 \quad \mathit{nr\_ch}^{1'} = \mathit{nr\_ch}^1 + 1 \quad q' = \mathit{add}(q, c)}{(\mathit{nr\_ch}^1, \mathit{nr\_ch}^2, q, \mathit{lead}) \overset{c}{\to} (\mathit{nr\_ch}^{1'}, \mathit{nr\_ch}^{2'}, q', \mathit{lead}')}$$

$$\frac{\mathit{lead} \notin \{0, 1\} \quad c \neq \mathit{first}(q) \quad \mathit{nr\_ch}^{1'} = \mathit{nr\_ch}^1 + 1 \quad q' = \bot \quad \mathit{lead}' = \bot}{(\mathit{nr\_ch}^1, \mathit{nr\_ch}^2, q, \mathit{lead}) \overset{c}{\to} (\mathit{nr\_ch}^{1'}, \mathit{nr\_ch}^{2'}, q', \mathit{lead}')}$$

Notice that above we studied stuttering equivalence in a generic way, where two traces could make a state change simultaneously. However, in the self-composed program model, the operational semantics of parallel composition ensures that in every step, either C_1 or C_2, but not both, make a transition. Therefore, for any trace $\mathcal{T}$, state changes do not happen simultaneously in both $\Pi_1(\mathcal{T})$ and $\Pi_2(\mathcal{T})$. This also means that it can never happen that in one step, both *add* and *remove* are applied simultaneously on the queue.

Atomic Propositions: Next we define the atomic propositions of our program model, together with their valuation. Notice that their valuation is parametric on the concrete equality relation used. Below, when characterizing observational determinism, we instantiate this in different ways, to define stuttering equivalence on a low location trace, and on a low store trace, respectively.

For $m = 1, 2$,

- $\mathit{fst\_ch}^m$ denotes that a state change occurs for the first time in the program copy m.
- $\mathit{snd\_ch}^m$ denotes that a state change occurs in the program copy m, while the program copy $3 - m$ has already made this change.
- $\mathit{nr\_ch}^m < \mathit{nr\_ch}^{3-m}$ denotes that the number of state changes made by the program copy m is less than the total number of state changes made by the program copy $3 - m$.

The valuation function λ for these atomic propositions is defined as follows. Let $\mathfrak{c}$ denote a state of the composed trace.

$$\begin{aligned}
\mathit{fst\_ch}^m \in \lambda(\mathfrak{c}) &\Leftrightarrow \mathit{current}(\mathit{conf}_m(\mathfrak{c})) \neq \mathit{old}(\mathit{conf}_m(\mathfrak{c})) \text{ and} \\
&\quad\ \mathit{extra}(\mathfrak{c})(\mathit{lead}) = m \text{ or } \mathit{extra}(\mathfrak{c})(\mathit{lead}) = 0. \\
\mathit{snd\_ch}^m \in \lambda(\mathfrak{c}) &\Leftrightarrow \mathit{current}(\mathit{conf}_m(\mathfrak{c})) \neq \mathit{old}(\mathit{conf}_m(\mathfrak{c})) \text{ and} \\
&\quad\ \mathit{extra}(\mathfrak{c})(\mathit{lead}) = 3 - m \text{ and} \\
&\quad\ \mathit{current}(\mathit{conf}_m(\mathfrak{c})) = \mathit{first}(\mathit{extra}(\mathfrak{c})(q)). \\
\mathit{nr\_ch}^m < \mathit{nr\_ch}^{3-m} \in \lambda(\mathfrak{c}) &\Leftrightarrow \mathit{extra}(\mathfrak{c})(\mathit{nr\_ch}^m) < \mathit{extra}(\mathfrak{c})(\mathit{nr\_ch}^{3-m}).
\end{aligned}$$

Program Model: Using the definitions of state, transition relation and atomic propositions above, we can now define a program model, encoding the behavior of a self-composed program under a scheduler δ. The characterizations are expressed over this model.

Definition 5 (Program model). *Given a scheduler δ, let C be a program, and s_1 and s_2 be stores. The* program model $\mathcal{M}^{\delta}_{C,s,s'}$ *is defined as* $(\Sigma, \ \rightarrow_{\chi}\ , AP, \lambda, I)$ *where*

- *Σ denotes the set of all configurations, obtained by executing from the initial configuration under δ, including the extra information, as defined above;*
- *AP is the set of atomic propositions defined above, and λ is their valuation;*
- *$I = \{\langle C \parallel C', (s, s')\rangle\}$ is the initial configuration of the composed trace.*

5.5 Characterization of Stuttering Equivalence

Based on the observations and program model above, we characterize stuttering equivalence by an LTL formula ϕ.

$$\phi = \ G\Big(\bigwedge_{m\in\{1,2\}} \mathit{fst\_ch}^m \Rightarrow \ \mathit{nr\_ch}^{3-m} < \mathit{nr\_ch}^m \ \ U \ \mathit{snd\_ch}^{3-m}\Big).$$

Intuitively, this formula expresses the characteristics of stuttering equivalence: any state change occurring in one component trace will occur later in the other component trace; and in between these changes the number of state changes at the intermediate states in the latter is strictly smaller than in the first.

We prove formally that ϕ characterizes stuttering equivalence.

Theorem 1. *Let $\mathcal{T}$ be a composed trace that can be decomposed into T^1 and T^2 with $T^1_0 = T^2_0$, then $T^1 \sim T^2 \Leftrightarrow \mathcal{T} \models \phi$.*

Proof. See Appendix A.2 of [6].

6 Temporal Logic Characterization of Scheduler-Specific Observational Determinism

Based on the results from the previous section, a temporal logic formula characterizing scheduler-specific observational determinism can be established. The formula consists of two parts: one that expresses stuttering equivalence of low location traces, and one that expresses stuttering equivalence of low store traces. Both are instantiations of the formula characterizing stuttering equivalence defined above.

6.1 Definitions of Atomic Propositions

We define atomic propositions that are used to instantiate the characterization of stuttering equivalence in different ways, so that we can characterize stuttering equivalence over low location traces, and over low store traces. For each $l \in L$, $\mathit{fst\_ch}_l^m$, $\mathit{snd\_ch}_l^m$, and $\mathit{nr\_ch_l}^m < \mathit{nr\_ch}_l^{3-m}$ relate to each low variable, and $\mathit{fst\_ch}_L^m$, $\mathit{snd\_ch}_L^m$, and $\mathit{nr\_ch_L}^m < \mathit{nr\_ch}_L^{3-m}$ relate to the set of low variables L, where $m = 1$ or 2.

The formal definitions are defined as in the previous section, where equality is instantiated as $=_l$ (for $l \in L$) and $=_L$, respectively.

6.2 Characterization of Scheduler-Specific Observational Determinism

Now we can give a temporal logic formula characterizing the properties of traces of a program that is observationally deterministic under a scheduler δ. A program C is observationally deterministic under δ iff for any two low equivalent stores s_1 and s_2, the following formula holds on the traces of $\mathcal{M}^\delta_{C,s_1,s_2}$.

$$\left(\bigwedge_{l\in L} \phi_l\right) \wedge \ \phi_L, \ \text{where}$$

$$\phi_l = G\Big(\bigwedge_{m\in\{1,2\}} \mathit{fst\_ch}_l^m \ \Rightarrow \mathit{nr\_ch_l}^{3-m} < \mathit{nr\_ch}_l^m \ \ U \ \mathit{snd\_ch}_l^{3-m}\Big)$$

$$\phi_L = AG\Big(\bigwedge_{m\in\{1,2\}} \mathit{fst\_ch}_L^m \ \Rightarrow E(\mathit{nr\_ch_L}^{3-m} < \mathit{nr\_ch}_L^m \ \ U \ \mathit{snd\_ch}_L^{3-m})\Big)$$

Notice that ϕ_l is an LTL and ϕ_L a CTL formula.

Thus, if the program has n low variables, we have $n + 1$ verification tasks, where n tasks relate to low location traces and one task relates to low store traces. For each task, we instantiate the extra information χ and the equality relation differently.

Theorem 2. *Given program C and initial stores s_1 and s_2 such that $s_1 =_L s_2$, C is observationally deterministic under δ iff*

$$\mathcal{M}^\delta_{C,s_1,s_2} \models \left(\bigwedge_{l\in L} \phi_l\right) \wedge \ \phi_L.$$

Proof. See Appendix A.3 of [6].

7 Conclusion

This paper presents a new formal definition of observational determinism that approximates the intuitive definition of observational determinism well. If a program is accepted under a specific scheduler, no secret information can be derived from the publicly observable location traces and the relative order of updates.

Compliance with our definition can be verified via a characterization as a temporal logic formula. The characterization is developed in two steps: first we characterize stuttering equivalence, which is the basis of the definition of scheduler-specific observational determinism, and then we characterize our definition itself. The characterization is an important step towards model checking observational determinism properties.

Related Work: The idea of observational determinism originates from the notion of noninterference, which only considers input and output of programs. We refer to [13, 7] for a more detailed description of noninterference, its verification, and a discussion why it is not appropriate for multi-threaded programs.

Roscoe [11] was the first to state the importance of determinism to ensure secure information flow of multi-threaded programs. The work of Zdancewic and Myers, Huisman et al., and Terauchi [16, 7, 14] has been mentioned above. They all propose different formal definitions of observational determinism, with a corresponding verification method. Zdancewic and Myers propose a type system that requires that the type checked program must be confluent in order to be verified [14]. Terauchi also proposes a type system to verify observational determinism, but this one does not enforce confluence. Huisman et al. characterize observational determinism in CTL*, using a special non-standard synchronous composition operator, and also in the polyadic modal μ-calculus (a variation of the modal μ-calculus) [7]. The idea of using self-composition was first proposed by Barthe et al. and Darvas et al. [1, 3]. The way self-composition is worked out here, with a temporal logic characterization also bears resemblance with temporal logic characterizations of strong bisimulation [9].

Finally, Russo and Sabelfeld take a different approach to ensure security of a multi-threaded program. They propose to restrict the allowed interactions between threads and scheduler [12]. This allows them to present a compositional security type system which guarantees confidentiality for a wide class of schedulers. However, the proposed security specification is similar to noninterference, just considering input and output of a program.

Future Work: As future work, we will encode the characterization in one (or more) model checkers. An important challenge is to model the queue, as this can have a strong effect on the state space that has to be examined. An additional challenge is to make the program model parametric, so that properties can be expressed for varying initial values. This step will be necessary to scale to large applications.

Notice that observational determinism is a *possibilistic* secure information flow property: it only considers the nondeterminism that is possible in an execution, but it does not consider the probability that an execution will happen. In a separate line of work, we will also study how probability can be used to guarantee secure information flow.

Acknowledgment: The authors would like to thank Jaco van de Pol for his useful comments and the anonymous reviewers for useful feedback of an earlier version of this paper. Our work is supported by the Netherlands Organization for Scientific Research.

References

1. G. Barthe, P. D'Argenio, and T. Rezk. Secure information flow by self-composition. In R. Foccardi, editor, *Computer Security Foundations Workshop*, pages 100–114. IEEE Press, 2004.
2. G. Barthe and L.P. Nieto. Formally verifying information flow type systems for concurrent and thread systems. In *Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, FMSE '04, pages 13–22, New York, NY, USA, 2004. ACM.
3. A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Security in Pervasive Computing*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer-Verlag, 2005.
4. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java language specification, third edition.* Addison Wesley, 2005.
5. M. Huisman and H.-C. Blondeel. Model-checking secure information flow for multi-threaded programs. In *Theory of Security and Applications (Tosca)*, Lecture Notes in Computer Science. Springer-Verlag, 2011. To appear.
6. M. Huisman and T.M. Ngo. Scheduler-specific confidentiality for multi-threaded programs and its logic-based verification. Available via `http://www.homeewi.utwente.nl/∼ngominhtri/`, full version.
7. M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterization of observation determinism. In *Computer Security Foundations Workshop*. IEEE Computer Society, 2006.
8. M. Huth and M. Ryan. *Logic in computer science: modeling and reasoning about the system.* Cambridge University Press, second edition, 2004.
9. A. Parma and R. Segala. Logical characterizations of bisimulations for discrete probabilistic systems. In *Proceedings of the 10th international conference on Foundations of software science and computational structures*, FOSSACS'07, pages 287–301. Springer-Verlag, 2007.
10. D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. In *Inf. Processing Letters*, volume 63, pages 243–246, 1997.
11. A.W. Roscoe. Csp and determinism in security modeling. In *IEEE Symposium on Security and Privacy*, page 114. IEEE Computer Society, 1995.
12. A. Russo and A. Sabelfeld. Security interaction between threads and the scheduler. In *Computer Security Foundations Symposium*, pages 177–189, 2006.
13. A. Sabelfeld and A. Myers. Language-based information flow security. In *IEEE Journal on Selected Areas in Communications*, volume 21, pages 5–19, 2003.
14. T. Terauchi. A type system for observational determinism. In *Computer Science Foundations*, 2008.
15. T.V. Vleck. Timing channels. In *Poster session at IEEE TCSP conference*, 1990. Available via `http://www.multicians.org/timing-chn.html`.
16. S. Zdancewic and A.C. Myers. Observational determinism for concurrent program security. In *Computer Security Foundations Workshop*, pages 29–43. IEEE Press, June 2003.

Quantifying Information Flow in Programs using Program Logics

Vladimir Klebanov*

Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5, 76131 Karlsruhe, Germany
klebanov@kit.edu

Abstract. Recently, a method for detecting and quantifying information flow in programs using model checking has been proposed by Backes, Köpf and Rybalchenko [1]. We recast this method in terms of program logics and a deductive verification calculus. Our goal is to create a deductive quantitative analysis of information flow in object-oriented programs, and we use the KeY theorem prover for Java Dynamic Logic as our implementation platform. This is work in progress.

1 Introduction

Recently, there has been a surge in research on quantitative information flow analysis. The research is motivated by the observation that it is not feasible to completely prevent information leaks (i.e., the flow of confidential information to public ports) in realistic programs. Instead, practical security analysis demands a measure of leaked information in order to decide what leaks are tolerable.

A remarkable paper in this field is [1]. There, the authors present a two-stage approach to precise measurement of information flow. The first stage uses an off-the-shelf model checker to compute a summary of information flow in the program being analyzed. This summary takes the form of an equivalence relation describing which confidential inputs are indistinguishable by public program outputs in a given attack scenario. The second stage transforms this relation into a variety of information-theoretical metrics.

In this paper, we investigate what happens when in the first stage of quantitative analysis, model checking is replaced by deductive verification based on a program logic. In particular, we are using the KeY [5] verification system for Java and its symbolic execution calculus for first-order Java Dynamic Logic (which we introduce in Section 3). The advantages we expect from using a deductive system like KeY are:

- ability to check object-oriented programs and a high coverage of OO programming language features
- ability to detect and quantify information leaks via termination (Section 7.2)
- ability to analyze programs with a high number of code paths or unbounded loops (Section 7.1)

* This work was supported by the German National Science Foundation (DFG) under the priority programme 1496 "Reliably Secure Software Systems – RS3."

2 Related Work

There is a large body of work on demonstrating absence of information leaks in programs, which we cannot survey here.

In the field of precise quantitative information flow analysis, the most relevant related work for us is [1] and [8]. We have already discussed the former in the introduction. The latter work is concerned with checking quantitative leakage bounds via bounded model checking.

A theoretical account of the hardness of precisely quantifying information flow in programs is given in [13].

Another inspiration for our work is [6], showing different approaches to formalize and prove both program security (absence of leaks) and insecurity (presence of leaks) in Java DL and the KeY prover.

The self-composition technique was first presented in a workshop version of [6] and received further theoretical treatment and its name in [3]; it was also studied from the point of view of verification in [10].

3 Java Dynamic Logic

Java Dynamic Logic (Java DL) is the instance of first-order Dynamic Logic used in the KeY system. In this section, we explain the basics of Java DL. For an exhaustive account we refer to [5].

Syntax. The signature of Java DL includes predicate and function symbols as well as two kinds of variables: program variables (written in typewriter font: $\mathtt{v}$) and logical variables (written in math font: v). Program variables appear in programs as well as in assertions about them, and their value may differ from state to state. Logical variables are used for quantification; they may not appear in programs, and their value does not depend on a state. For succinctness, we will denote several related variables or terms as $\bar{\mathtt{v}}$, $\bar{v}$, $\bar{t}$, etc. and assume that all operations happen component-wise.

The set of formulas of Java DL is defined as common in first-order Dynamic Logic. That is, they are built using the connectives $\land, \lor, \rightarrow, \neg$ and the quantifiers $\forall, \exists$ (first-order part). Furthermore, for every Java program p and every formula ϕ, $\langle p \rangle \phi$ (the "diamond" modality) and $[p]\phi$ (the "box" modality, which is a shorthand for $\neg\langle p \rangle \neg\phi$) are (modal) formulas.

Java DL has a third, unique to it, modality: the update. An update has the form $\{\bar{\mathtt{v}} := \bar{t}\}$ resp. $\{\mathtt{v}_1 := t_1 \| \dots \| \mathtt{v}_n := t_n\}$ and describes a state transition where the program variables in $\bar{\mathtt{v}}$ are assigned the values of the terms in $\bar{t}$ in parallel. Updates serve several purposes. They help formalize symbolic execution, make for efficient aliasing treatment and allow relating program and logical variables (and thus quantification over program inputs).

Semantics. The semantics of Java DL is based on the notion of a (program) state, i.e., a first-order structure assigning (among other things) values to program variables. We presume an appropriate signature and refer to the set of all possible states that are based on it as S.

The transition relation $\rho_p \subseteq S \times S$ gives meaning to a program p as a relation between its initial and final states. The definition of the programming language fixes ρ_p for every syntactically valid program p. In this paper, we only consider deterministic programs, so all relations ρ_p are actually partial functions: for every initial state, there is at most one final state.

Furthermore, for any given state $s \in S$:

- Terms and formulas without modalities have the meaning as usual in first-order logic.
- The diamond formula $\langle p \rangle \phi$ is true in s, if the program p started in s terminates and the formula ϕ is true in the state $\rho_p(s)$ reached upon termination.
- The meaning of a box formula is the same, but termination is not required: $[p]\phi$ is true in s, if either p does not terminate when started in s, or $\langle p \rangle \phi$ is true in s. The formula $\psi \to [p]\phi$ has the same intuitive meaning as the triple $\{\psi\}p\{\phi\}$ in Hoare Logic.
- The meaning of a formula with an update $\{\bar{\mathrm{v}} := \bar{t}\}\phi$ in the state s is the same as the meaning of ϕ in the state s' that is identical to s except as follows: s' assigns the variables $\bar{\mathrm{v}}$ the values that the terms $\bar{t}$ have in s.

A formula is *logically valid* if it is true in every state.[1]

Calculus. To prove validity of formulas of Java DL, KeY implements a sequent calculus. A *sequent* is of the form $\Gamma \Longrightarrow \Delta$, where the *antecedent* Γ and the *succedent* Δ are sets of formulas. The meaning of a sequent is that of the formula $\bigwedge_{\phi \in \Gamma} \phi \;\to\; \bigvee_{\psi \in \Delta} \psi$. We call the latter formula the *meaning formula* of the sequent and will refer to it as $M(\Gamma \Longrightarrow \Delta)$.

A *rule schema* is of the form

$$\frac{L_1 \quad L_2 \quad \cdots \quad L_k}{C} \qquad (k \geq 0)$$

where $L_1, \ldots, L_k$ and C are schematic sequents, i.e., sequents containing schema variables. As common in sequent calculus, the direction of entailment in the rules is from premisses (sequents above the bar) to the conclusion (sequent below), while reasoning in practice happens the other way round: by matching the conclusion to the goal.

A *proof tree* is a finite tree, where each node is annotated with a sequent, and each inner node is additionally annotated with one of the calculus rules relating the node's sequent to the sequents of its descendants. A leaf node may or may not be annotated with a rule. If it is, then the rule must be a closing rule (i.e., have no premisses). If it is not, then we call such leaf node an *open*

[1] Thus, there is implicit universal quantification over program variables.

$$\textsf{hideLeft}\ \frac{\Gamma, \quad \Longrightarrow \Delta}{\Gamma, \phi \Longrightarrow \Delta}$$

$$\textsf{assignment}\ \frac{\Gamma \Longrightarrow \{\texttt{loc} := \texttt{val}\}\langle \pi\ \ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \pi\ \texttt{loc = val;}\ \omega\rangle\phi, \Delta}$$

$$\textsf{ifElseSplit}\ \frac{\begin{array}{c}\Gamma, \texttt{v} = \mathrm{TRUE} \Longrightarrow \langle \pi\ \ p\ \ \omega\rangle\phi, \Delta \\ \Gamma, \texttt{v} = \mathrm{FALSE} \Longrightarrow \langle \pi\ \ q\ \ \omega\rangle\phi, \Delta\end{array}}{\Gamma \Longrightarrow \langle \pi\ \texttt{if (v)}\ p\ \texttt{else}\ q\ \ \omega\rangle\phi, \Delta}$$

$$\textsf{invariant}\ \frac{\begin{array}{c}\Gamma \Longrightarrow \mathcal{U}I,\ \Delta \\ I, \quad \texttt{v} \Longrightarrow [p]I \\ I,\ \neg\texttt{v} \Longrightarrow \phi\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\texttt{while (v) \{}\ p\ \texttt{\}}]\phi,\ \Delta}$$

The following is a small illustration of KeY calculus rules. The rules have been simplified. Many more (and more complicated) rules are necessary to deal with a real-life language like Java.
ϕ is a formula of Java DL. The prefix π stands for a sequence of opening braces `{`, labels, `try{`, etc. The postfix ω denotes the "rest" of the program. `v` is a local variable. $\mathcal{U}$ is a sequence of updates. I is the user-provided loop invariant.

Fig. 1. An illustration of the KeY calculus rules

goal. If all leafs of a proof tree for the sequent $\Gamma \Longrightarrow \Delta$ are closed, then the formula $M(\Gamma \Longrightarrow \Delta)$ is valid.

The KeY calculus contains rules for symbolic execution of Java programs, update simplification, induction, first-order and theory reasoning. A few (simplified) examples of the rules are given in Figure 1, but we will not discuss them in detail here. The automated proof search strategy of KeY applies rules to reduce a proof obligation containing programs to purely first-order goals (and then prove these). This process is completely automatic, if for each program loop, either a (reasonably small) bound is known (and the loop can be unrolled), or a loop invariant is available.

4 Basics of Information Flow

We extend the standard Java DL for reasoning about information flow as follows:

- The signature marks each program variable either as *high* (confidential) or as *low* (publicly observable/changeable).[2]

[2] This definition forces us to mark local variables as high, but this restriction has no practical consequence.

- According to the above distinction, we define projection functions $\cdot_{hi}$ and $\cdot_{lo}$. Each state $s \in S$ is a pair of its high component s_{hi} and its low component s_{lo}, and $S = S_{hi} \times S_{lo}$.

The attacker. Our attacker model is as follows. Assume a run of a program p, with an initial state $s = (s_{hi}, s_{lo})$, and the final state $s' = \rho_p(s) = (s'_{hi}, s'_{lo})$. The attacker knows p, s_{lo}, and s'_{lo} (but not s_{hi}, s'_{hi}, or any intermediate states). The goal of the attacker is to learn something about s_{hi}.

The amount of information leaked by the program (and thus the success of the attacker) depends on the number of program runs that the attacker can study. Each such run is called in terminology of [1] an *experiment*, and it is uniquely characterized by the low component s_{lo} of the initial state. We assume that the attacker can freely choose s_{lo}.

Our whole analysis is parameterized by a set of experiments E. This parameter is chosen by the security analyst. Different E describe different attack scenarios. A singleton set E corresponds to a single guess (a password checker, for instance, is relatively secure in this scenario), while $E = S_{lo}$ models an exhaustive attack (against which a password checker is helpless).

Describing information leaks. The canonical way to describe information leakage of a program is by grouping confidential inputs that lead to the same public output in a given attack scenario. As far as we know, this relation has no name in literature; we like to call it the *cover-up relation.*

Definition 1 (Cover-up relation). *For a given program p and a set of experiments E, the* cover-up relation $\approx_p^E \subseteq S_{hi} \times S_{hi}$ *is*

$$\approx_p^E = \{(s_{1hi}, s_{2hi}) \mid \textit{for all } e \in E : \big(\rho_p((s_{1hi}, e))\big)_{lo} = \big(\rho_p((s_{2hi}, e))\big)_{lo}\} \ .$$

Intuitively, $\approx_p^E$ is an "equivalence relation on the set of possible secret inputs. Two inputs are in the same equivalence class whenever the program produces the same result on both inputs. By observing the output of the program, the attacker can then only deduce the secret input up to its [...] equivalence class." [1]

More precisely, for a given p and E, $\approx_p^E$ is an equivalence relation on high state components. Taking any two states

- with their high components in the same $\approx_p^E$-equivalence class
- and their low components identical and in E

as initial states for running p will lead to the same observable (i.e., low) final state. We will consider the case that programs may not terminate in Section 7.2.

Secure programs have a coarse cover-up relation, while insecure a fine one. If the cover-up relation is identity (very fine), then all equivalence classes are singleton sets, and each low final state corresponds uniquely to a high initial state: the attacker has perfect knowledge. Conversely, the coarsest cover-up relation $\approx_p^E = S_{hi} \times S_{hi}$ with only one equivalence class means that the attacker learns nothing about the high inputs observing the low outputs.

5 Detecting Information Flow with KeY

We use KeY to compute (a logical description of) $\approx_p^E$. We use self-composition, i.e., employ two copies of the program $p(\bar{\mathtt{h}}, \bar{\mathtt{l}})$ with renamed variables: $p_1 := p(\bar{\mathtt{h}}_1, \bar{\mathtt{l}}_1)$ and $p_2 := p(\bar{\mathtt{h}}_2, \bar{\mathtt{l}}_2)$. Our goal is to determine a program-free formula $\Phi(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2)$ such that

$$s_{1hi} \approx_p^E s_{2hi} \text{ iff } \Phi(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2) \text{ is true in a state } (s_{1hi} \oplus s_{2hi}, s_{lo}) \ ,$$

where s_{lo} is some low state component and $s_{1hi} \oplus s_{2hi}$ is a high state component where the values of $\bar{\mathtt{h}}_1$ are the same as the values of $\bar{\mathtt{h}}$ in s_{1hi} and the values of $\bar{\mathtt{h}}_2$ are the same as the values of $\bar{\mathtt{h}}$ in s_{2hi}.

Computing $\approx_p^E$. To determine $\Phi(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2)$, we use the automated proof search of KeY to construct a proof tree for the sequent

$$\Longrightarrow \exists \bar{x}. \Big(E(\bar{x}) \wedge \{\bar{\mathtt{l}}_1 := \bar{x} \,||\, \bar{\mathtt{l}}_2 := \bar{x}\} \langle p_1 \rangle \langle p_2 \rangle \neg \bar{\mathtt{l}}_1 = \bar{\mathtt{l}}_2 \Big) \ . \tag{1}$$

The formula in the succedent is the negated definition of $\approx_p^E$. Thus, open proof goals[3] correspond to models of $\approx_p^E$. The predicate $E(x)$ describes the set of experiments E.

Formally, we assume a proof tree with Sequent (1) at the root (we'll call it $\Longrightarrow \neg A$ in the following) and the open goals $L_1, \ldots, L_n$ as leafs. Given the soundness of the calculus, we know that the meaning of the root is implied by the conjunction of the leaf meanings: $M(\Longrightarrow \neg A) \leftarrow M(L_1) \wedge \ldots \wedge M(L_n)$. If all loops are bounded and unrolled and all method calls are inlined (essentially as in [1]), then we can assume equivalence between the root sequent and the conjunction of open goals [9]. We will deal with unbounded loops and other potential sources of weakening in Section 7.1. Under this equivalence, the models of A (which defines $\approx_p^E$) are exactly those of $\Phi(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2) := \neg M(L_1) \vee \ldots \vee \neg M(L_n)$.

Example 1. We illustrate the computation of $\approx_p^E$ with the popular password checker example, implemented as follows:

```
if (candidate==pass) { res=true;} else { res=false; }
```

Low variables are `candidate` (the entered password) and `res` (the result of the check). The only high variable is `pass` (the correct password). Instantiating (1) gives us the sequent

$$\begin{aligned}\Longrightarrow \exists x. \Big(& E(x) \wedge \{\mathtt{candidate1} := x \,||\, \mathtt{candidate2} := x\} \\ & \neg\langle \mathtt{if\ (candidate1{=}{=}pass1)\ res1{=}true;\ else\ res1{=}false;} \rangle \\ & \ \langle \mathtt{if\ (candidate2{=}{=}pass2)\ res2{=}true;\ else\ res2{=}false;} \rangle \\ & \qquad\qquad\qquad\qquad\qquad\qquad \mathtt{res1} = \mathtt{res2} \Big)\end{aligned}$$

[3] The sequent is indeed not provable, since we can choose the same initial state for both programs (i.e., $\bar{\mathtt{h}}_1 = \bar{\mathtt{h}}_2$), which will lead to $\bar{\mathtt{l}}_1 = \bar{\mathtt{l}}_2$ in the final state.

which we will try to prove with KeY for different attack scenarios $E(x)$.

Choosing the predicate $E(x)$ as $x = 5$, we model a single attempt by the attacker to guess if the password is five. After the KeY proof search is exhausted, we are left with two open goals:

$$\texttt{pass1} = 5, \texttt{pass2} = 5 \Longrightarrow \quad \text{and} \quad \Longrightarrow \texttt{pass1} = 5, \texttt{pass2} = 5 \enspace .$$

The logical description $\Phi(\texttt{pass1}, \texttt{pass2})$ of $\approx_p^E$ is thus:

$$(\texttt{pass1} = 5 \land \texttt{pass2} = 5) \lor (\texttt{pass1} \neq 5 \land \texttt{pass2} \neq 5) \enspace .$$

We will give a systematic way to compute the equivalence classes of this relation in Section 6.1, but it is clear that there are two of them in this case: one very small class $\{[\![5]\!]\}$[4] and one very large class $\{[\![i]\!] \mid i \in Z \setminus \{5\}\}$. Intuitively, the program is quite secure in this attack scenario.

Setting $E(x)$ to $x = 5 \lor x = 7$, we model a two-guess attack. The result is three equivalence classes: two very small ones $\{[\![5]\!]\}$ and $\{[\![7]\!]\}$ and one still very large class $\{[\![i]\!] \mid i \in Z \setminus \{5, 7\}\}$. In this scenario, the program is just slightly less secure than in the previous one.

Setting $E(x)$ to *true*, we model an exhaustive attack. The result is one open goal $\texttt{pass1} = \texttt{pass2} \Longrightarrow$, which makes $\approx_p^E$ the identity relationship with many singleton classes. The program is completely insecure in this attack scenario.

6 Quantifying Information Flow

Stage two of the quantitative information flow analysis is concerned with computing the individual equivalence classes of the cover-up relation $\approx_p^E$ and summarizing their number and/or size in a single quantitative measure. The technique that we use for this is—modulo some implementation details—that of [1].

6.1 Computing the Equivalence Classes of $\approx_p^E$

In Section 5, we have obtained $\Phi(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2)$ as a logical description of the cover-up relation $\approx_p^E$. It is now our goal to obtain logical descriptions of individual equivalence classes of $\approx_p^E$. To reach this goal, we compute a representative system

$$\{[\![\bar{r_i}]\!] \mid \; S_{hi} = \uplus_i \left[[\![\bar{r_i}]\!]\right]\}$$

where $[\![\bar{r_i}]\!]$ is the high state component, in which the high variables $\bar{\mathtt{h}}$ have the same value as the terms $\bar{r_i}$, and $\left[[\![\bar{r_i}]\!]\right]$ is its equivalence class. Then, $\Phi(\bar{\mathtt{h}}, \bar{r_i})$ is the logical description of the ith equivalence class of $\approx_p^E$.

The representative system is computed by the following iterative algorithm, where we are repeatedly asking for a representative term that is not equivalent to any of the previously computed representatives:

[4] With $[\![5]\!]$ we denote the high state component where pass has the value 5.

```
1 input $\Phi(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2)$
2 $\Phi(\bar{x}, \bar{y}) := \Phi(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2)$
3 $Repr := \emptyset$
4 while ($\Phi(\bar{x}, \bar{y})$ is satisfiable) do
5     $\bar{r}$ := value of $\bar{x}$ in a satisfying assignment of $\Phi(\bar{x}, \bar{y})$
6     $Repr := Repr \cup \{\bar{r}\}$
7     $\Phi(\bar{x}, \bar{y}) := \Phi(\bar{x}, \bar{y}) \wedge \neg\Phi(\bar{x}, \bar{r})$
8 od
9 output $Repr$
```

In this algorithm, we delegate both the satisfiability check in line 4 and the generation of a satisfying assignment in line 5 to the SMT solver Z3 [7]. This arrangement works as long as the formula $\Phi(\bar{x}, \bar{y})$ falls into a logical fragment that is decidable and for which an SMT solver can generate models. Indeed, for a large class of programs, $\Phi(\bar{x}, \bar{y})$ is a quantifier-free formula over linear arithmetic. For non-integer data types (such as strings), we plan to investigate mappings into the integer domain. On the implementation side, we profit from the KeY system's interface to SMT solvers.

6.2 Quantifying Information Flow

Having computed the equivalence classes of $\approx_p^E$ in the previous step, it is time to determine their size. This can be achieved by an implementation of Barvinok's algorithm [4] for counting the integer points in polytopes. While [1] uses the LattE framework for this purpose, we have chosen the Barvinok tool [12, 11], which is easier to handle technically.

The Barvinok tool takes the formula $\Phi(\bar{\mathtt{h}}, \bar{r}_i)$ (in the syntax largely compatible with that of KeY) and returns the number of integer assignments for $\bar{\mathtt{h}}$ satisfying the formula, which is the size of the $\bar{r}_i$-equivalence class. It is, of course, necessary to add range restrictions for variables. The tool also works with parametric polytopes, although it is not yet clear whether this capability allows analyzing a larger class of programs.

Given the number and sizes of equivalence classes, "it is possible to compute different security measures, such as the average uncertainty about the secret in bits (Shannon entropy), the average number of guesses that are needed to identify secrets (conditional and minimal guessing entropy), and the maximal rate at which information can be transmitted using the program as a communication channel (channel capacity)." [1] For the moment, we do not include these calculations here, but refer to the intuitive argument of Example 1.

7 Extensions

7.1 Treating Unbounded Loops and the Role of Weakening

In this section we investigate how invariants can be used to deal with loops where unrolling is either impossible (due to the unknown loop bound) or infeasible (due to the high number of iterations and/or code paths through the loop body). The main question is how the use of the invariant rule affects the soundness of computing the cover-up relation.

The computation is sound if all rules applied in the proof attempt are "reversible", i.e., their conclusion C is equivalent to the conjunction of the premisses $C_1 \wedge \ldots \wedge C_n$ (cf. Section 5). This is not necessarily the case with so-called *weakening rules*, to which the invariant rule belongs. The premisses of a weakening rule are strictly stronger than its conclusion: $C \leftarrow C_1 \wedge \ldots \wedge C_n$. Weakening rules are sound for proving validity (and thus functional verification), but not for computing the cover-up relation. Using weakening in the computation may produce a relation that is too coarse and thus deem the program more secure than it actually is. We now discuss the sources of weakening in proofs and how to remedy this situation.

There are three potential sources of weakening: the explicit weakening rules (e.g., hideLeft in Figure 1), the invariant rule (invariant in Figure 1) and the method contract rule (also known as the modular method call rule, not shown). While the explicit weakenings can be eliminated from a proof without detriment, the invariant rule is necessary for completeness in the presence of unbounded loops and the contract rule for practical scalability. The latter two rules become reversible, if one uses strongest specifications (invariants or contracts). A strongest specification implies any other specification that is still satisfied by the loop or method. It fully captures the behavior of the corresponding code.

Finding strongest specifications is at least as hard as finding specifications in general, but certainly more annoying in practice. In functional verification, one typically wants specifications that are merely strong enough to show the desired postcondition. Everything else just makes the proof harder. On the other hand, while being demanding, the use of specifications allows analyzing security of strictly more programs than previous approaches based on model checking.

We propose the following algorithm for checking if unsoundness was introduced through the use of specifications:

1. Compute $\Phi(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2)$ as described in Section 5. We assume (without loss of generality) exactly one use of an invariant or contract rule with the specification I.
2. Compute the representative system $\{[\![\bar{r}_i]\!]\}$ as described in Section 6.1. Each of the equivalence classes of $\approx_p^E$ can be described syntactically with the formula $\Phi(\bar{\mathtt{h}}, \bar{r}_i)$.
3. Check, whether $\Phi(\bar{\mathtt{h}}_1, \bar{\mathtt{h}}_2)$ is unsoundly coarse by attempting to prove noninterference within each equivalence class: for each equivalence class i, prove
$$\Longrightarrow \forall \bar{x}. \big(E(\bar{x}) \wedge \Phi(\bar{\mathtt{h}}_1, \bar{r}_i) \wedge \Phi(\bar{\mathtt{h}}_2, \bar{r}_i) \rightarrow \{\bar{\mathtt{l}}_1 := \bar{x} \,||\, \bar{\mathtt{l}}_2 := \bar{x}\}\langle p_1 \rangle \langle p_2 \rangle \bar{\mathtt{l}}_1 = \bar{\mathtt{l}}_2\big) \tag{2}$$
using the specification I. Since I is known, the proof either succeeds automatically or fails. A successful proof demonstrates directly that the computed characterization of the equivalence class i is sufficiently fine to comply with the definition of $\approx_p^E$. If the proof fails, then I must be strengthened (by the user) and the process repeated from Step 1.

4. Check that no equivalence classes have been missed. Prove:

$$\Longrightarrow \Phi(\bar{\mathtt{h}}, \bar{r}_1) \lor \ldots \lor \Phi(\bar{\mathtt{h}}, \bar{r}_n) \ . \tag{3}$$

If the proof succeeds, then the computed characterization of $\approx_p^E$ is sound. If it fails, then I must be strengthened (by the user) and the process repeated from Step 1.

7.2 Treating Leaks by Termination

A *leak via termination* occurs when an attacker learns something about the secret inputs by observing that a program run terminates. Observing termination, of course, only leaks information if there is a possibility of non-terminating program runs. Due to the finite execution requirement of model checking, the analysis of [1] is limited to terminating programs (where such leaks do not occur). In contrast, we can extend the proof obligation (1) to measure information leaking via termination (as done for proving insecurity in [6]):

$$\Longrightarrow \exists \bar{x}. \Big(E(\bar{x}) \land \{\bar{\mathtt{l}}_1 := \bar{x} \,||\, \bar{\mathtt{l}}_2 := \bar{x}\}\big(\\ (\langle p_1\rangle\langle p_2\rangle\neg\bar{\mathtt{l}}_1 = \bar{\mathtt{l}}_2) \bigvee (\langle p_1\rangle \mathit{true} \land [p_2]\mathit{false})\big)\Big) \tag{4}$$

This way, we take into account that the attacker can gain knowledge not only from the varying low output ($\neg\bar{\mathtt{l}}_1 = \bar{\mathtt{l}}_2$), but also from the fact that one program run terminates ($\langle p_1\rangle \mathit{true}$ holds) while another doesn't ($[p_2]\mathit{false}$ holds).

8 Conclusion and Current/Future Work

So far, we have carried out just a few initial experiments, combining KeY and the other tools manually. We are currently automating this process and are looking forward to more experiments and evaluating the technique in more detail. We plan to extend the analysis to non-integer inputs/outputs by investigating mappings into the integer domain.

We would like to investigate the role of specifications in more detail. It would also be interesting to extend the analysis to nondeterministic (e.g., multi-threaded) programs.

Regarding the generality of our encoding, we assume that the analysis could be implemented in any program logic that can express self-composition and has a calculus that computes first-order verification conditions. In view of the recent results on program products [2], we speculate that the self-composition requirement can also be dropped.

References

1. Michael Backes, Boris Köpf, and Andrey Rybalchenko. Automatic discovery and quantification of information leaks. In *Proceedings, 30th IEEE Symposium on Security and Privacy (S&P 2009)*, pages 141–153. IEEE Computer Society, 2009.

2. Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In Michael Butler and Wolfram Schulte, editors, *Proceedings, 17th International Symposium on Formal Methods (FM)*, volume 6664 of *LNCS*, pages 200–214. Springer, 2011.
3. Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, CSFW-17, Pacific Grove, CA, USA*, pages 100–114. IEEE Computer Society, 2004.
4. Alexander I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Math. Oper. Res.*, 19:769–779, November 1994.
5. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
6. Ádám Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors, *Proceedings, Security in Pervasive Computing*, volume 3450 of *LNCS*, pages 193–209. Springer, 2005.
7. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
8. Jonathan Heusser and Pasquale Malacaria. Quantifying information leaks in software. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 261–269. ACM, 2010.
9. André Platzer. Using a program verification calculus for constructing specifications from implementations. Minor Thesis, University of Karlsruhe, Department of Computer Science. Institute for Logic, Complexity and Deduction Systems, February 2004. Available at `http://symbolaris.com/logic/Minoranthe.pdf`.
10. Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In Chris Hankin and Igor Siveroni, editors, *Proceedings, Symposium on Static Analysis*, volume 3672 of *LNCS*, pages 352–367. Springer, 2005.
11. Sven Verdoolaege. The Barvinok tool. Website at `www.kotnet.org/~skimo/barvinok/`.
12. Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48(1):37–66, June 2007.
13. Hirotoshi Yasuoka and Tachio Terauchi. Quantitative information flow – verification hardness and possibilities. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF '10, pages 15–27. IEEE Computer Society, 2010.

Verification of Actor Systems Needs Specification Techniques for Strong Causality and Hierarchical Reasoning [⋆]

— Position Paper —

Arnd Poetzsch-Heffter, Ilham W. Kurnia, Christoph Feller

University of Kaiserslautern, Germany
{poetzsch,ilham,c_feller}@cs.uni-kl.de

Abstract. Actor languages become increasingly popular for modeling and programming concurrent and distributed system. Although several foundational proof systems are available for actor systems, we claim that there is a need for higher-level specification and verification techniques. To clarify and substantiate this position, we introduce a simple actor language together with a non-trivial example and show how language-based specification techniques, generalizing ideas from object-oriented specifications, can provide more structure and modularity. We explain the semantics of the specification constructs based on strong causality relations of incoming and outgoing messages and illustrate their use. Furthermore, we discuss how verification techniques can be constructed for such specifications. The paper finishes with a discussion of related and future work.

1 Introduction

For analysis, checking, and verification, many modern programming languages are complemented by specification languages that allow the formulation of program-specific properties (e.g., Java and JML [22], C# and Spec# [5], Ada and Anna [25]). Whereas these specification languages mainly focus on sequential programming, researchers more and more investigate multi-threading and other concurrency aspects of such programming languages. On the other hand, there is a trend to develop modeling and programming languages directly based on concurrency primitives, e.g., based on actors or processes (e.g. [3, 18, 31, 7, 20, 19]). A strong argument for these approaches is that their concurrency constructs usually come with a well-understood theory and often foundational reasoning technique. However, the development of specification and high-level reasoning techniques for these languages is less advanced than for the classical programming languages.

In this paper, we focus on actor languages in which a system or component is described as a dynamically varying set of actors. An actor can create other actors and communicate with other actors via asynchronous messages. When used for

[⋆] This work is partially supported by the EU project *FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models* and the RS3 project *MoVeSPAcI: Modular Verification of Security Properties in Actor Implementations*.

modeling purposes, actor languages need specification languages to state and verify properties of the modeled systems. We claim that the specification techniques and corresponding verification methods to develop such specification languages are not sufficiently understood. Formulated positively, we aim to convey that the development of specification techniques for actor languages is an interesting and fruitful research area.

More precisely, the *position of this paper* is that the following questions still wait for satisfying solutions:

1. What are the central specification constructs for actors?
2. How do we achieve scaling from single actors to actor systems?
3. How can we build higher-level reasoning techniques based on such actor specifications?

In the following, we explain these questions in turn by comparing them to the specification techniques for classical programming languages. In addition, we shortly introduce the approaches to these questions that we will further elaborate in the rest of the paper to clarify the stated position.

The central specification construct for programming languages centers around procedures/methods and is based on pre- and postconditions (see, e.g., [28]). A pre-/post-specification nicely abstracts from the execution sequence of a procedure by talking only about its initial and final states. In object-oriented (OO) languages, pre-/post-specifications of methods are complemented by object invariants (see, e.g., [24]). What are the corresponding central abstractions to specify actors living in a concurrent environment? Our approach is to specify the reaction to incoming messages.[1] As a new concept, we will propose to use so-called *strong causality relations* between incoming messages and the resulting outgoing messages as a generalization of pre-/post-specifications.

In sequential settings, procedures hierarchically structure the complete system runs. One call might lead to subcalls. The behavior of the main call can be derived from the behaviors of the subcalls. Similar to OO settings, the scaling in actor programs is less clear. Specification techniques for OO programs often use ownership concepts for scaling from single objects to object structures (see, e.g., [32, 5]). Our approach is similar in that we will propose to specify the behavior of components consisting of groups of actors. In particular, we aim to specify such components with the same technique as actors.

There are foundational theories how to reason about actor systems (see, e.g., [1, 14, 2]). These theories directly build on the traces generated by the actor systems. This is very flexible and powerful. However, for practical purposes, it can get quite complex and elaborate. Are there higher-level, easier to handle verification methods for reasoning based on a specification technique, maybe even at the expense of losing relative completeness? In this paper, we will only discuss the topic in relation to the presented specification techniques.

[1] How this approach can be extended to spontaneous actions will be discussed below.

```
interface Client { void receive( Value );   }

interface Server { void serve( Client, CompTask ); }

interface Worker {
   void do( CompTask, Int );
   void propagateResult( Value, Client );
}

actor class AServer implements Server {
   void serve( Client c, CompTask t ) {
      Int    numberSubtasks = taskSize(t); // numberSubtasks >= 1
      Worker w = new AWorker();
      w.do( t, numberSubtasks );
      w.propagateResult( null, c );
   }
}
```

Fig. 1. Interfaces and the actor class `AServer`

Paper outline. To clarify and substantiate the sketched position, we discuss the questions mentioned above in a concrete setting. Sect. 2 describes a simple actor language that we call AJ and a non-trivial example written in AJ. In Sect. 3, we present and discuss a candidate for a specification technique and apply it to the running example. In Sect. 4, we discuss verification aspects. Sect. 5 contains summarizing discussions, further comparisons to related work, and our aims in future work.

2 Actor Language and Actor Systems

To have a sufficiently clear background for the following discussion on specification and verification, we informally introduce the core actor language AJ together with a client-server example. The server takes a computation task, splits it up into a number of subtasks that are concurrently executed, collects and merges the results, and sends the final result back to the client (this is a simplified version of the ring example treated in a case study by Arts and Dam [4]).

Actors share many properties with objects. In particular, they are declared using classes (we use the keyword **`actor class`**), can be dynamically created, implement interfaces, have an actor-local state expressed in terms of instance variables, and are addressed via a typed reference. Thus, for these aspects, we can use a syntax similar to OO programming languages. For example, Fig. 1 shows the interfaces `Client`, `Server`, and `Worker` together with an implementation of the `Server`-interface.

Reacting to messages. The central difference of actors and objects is the treatment of messages and methods. In AJ, a *message* consists of a name and typed parameters, but must not have a return type/value[2], and the semantics of message sending is different. A statement of the form `r.m(p1,p2)` is executed by sending the message `m` with actual parameters `p1` and `p2` to the receiver actor `r`. Such a send-operation is non-blocking; execution directly continues with the next statement. Thus, in general, a message send leads to concurrent behavior. For each of its messages, an actor has a *body* that describes how it reacts to a message. For example, an actor of class `AServer` (see Fig. 1) reacts on a message `serve` as follows: It determines the number of subtasks into which task `t` should be split, creates a worker actor, and sends first a `do`- and then a `propagateResult` message to the worker. For AJ, we make the following simplifying assumptions (we discuss the assumptions in Sect. 5):

A1 Messages sent from one sender to the same receiver are transmitted in order.
A2 Actors are single-threaded, i.e., they work on at most one message body at a time.
A3 Message bodies are executed without interruption.
A4 The execution of message bodies must terminate. This is an obligation of the programmer. (If an actor should be non-terminating, it can send itself a message at the end of the message body.)

Receiving and selecting messages. It remains to explain what happens on a message receive. We assume that actors are input enabled (cf. [27, p. 257]) and have an infinite input queue. Messages are *selected* from the queue essentially in a FIFO manner. However if they have a guard that evaluates to false, their selection is postponed. Thus, an actor has control over the execution of incoming messages. Message selection is fair for messages with true guards. In Fig. 2, the actor class `AWorker` uses a guard to select a `propagateResult`-message only if a result is available.

Further constructs. In addition to the actor-related aspects, we assume some basic language constructs usually present in functional programming languages. In the example, we use the types `CompTask` and `Value` and the functions:

```
  compute   :  CompTask ⟶ Value
  taskSize  :  CompTask ⟶ Int
  firstTask :  CompTask × Int ⟶ CompTask
  restTask  :  CompTask × Int ⟶ CompTask
  merge     :  Value × Value ⟶ Value
```

where `compute(t)` computes the result of `t`; `taskSize(t)` yields a number of subtask in which `t` could be reasonably partitioned; `firstTask(t,n)`, for $\mathtt{n} \geq 1$, chops off the last $\mathtt{n} - 1$-th part of `t`, `restTask(t,n)` returns the chopped part; and `merge` merges results. We assume in particular:

```
taskSize(t) ≥ 1
n > 1 → compute(t) = merge(compute(firstTask(t,n)),compute(restTask(t,n)))
compute(t) = compute(firstTask(t,n))
merge(v,null) = v
merge(v,merge(w,x)) = merge(merge(v,w),x)
```

[2] I.e., return types are always **`void`**.

```
actor class AWorker implements Worker {
   Value  myResult = null;
   Worker nextWorker = null;

   void do( CompTask t, Int n ) {
      if( n > 1 ) {
         nextWorker = new AWorker();
         nextWorker.do( restTask(t,n), n-1 );
      } else {
         nextWorker = null;
      }
      myResult = compute( firstTask(t,n) );
   }

   void propagateResult( Value v, Client c )
      guard  myResult != null
   {
      if( nextWorker == null ) {
          c.receive( merge(myResult,v) );
      } else {
          nextWorker.propagateResult( merge(myResult,v), c );
      }
   }
}
```

Fig. 2. Actor class `AWorker`

Actor systems are started by creating actors on computers and start their activities or connect them to activities in the environment, for example to user interfaces.

We designed AJ in such a way that it supports the core features of actor programming and is easy to understand for the object community in order to simplify the bridge to the OO specification community. A comparison of AJ in the context of related work is contained in Sect. 5.

3 Specifications of Functional Properties

Even if described by a simple language such as AJ, actor systems can become very complex with almost unmanageable behavior. Of course, in theory we can capture this behavior in form of trace sets and formalize properties as predicates on these sets. However, for practical purposes, we would like to have a more modular and more structured approach. Modularity would allow us for example to prove properties about our server actors of Sect. 2 without knowing all possible contexts in which the servers are used. The hierarchical structuring should help us to develop specifications for typical programs. Our position is that improvements w.r.t. modularity and hierarchical techniques are necessary to master actor systems.

To clarify and substantiate the above claim, we describe how such improvements could look like. We focus on the basic ideas and make several simplifying assumptions. Furthermore, we try to reuse OO specification technology where appropriate. Our central goal is

> *a specification technique that works for single actors and scales to groups and components consisting of many actors.*

The reason for looking at groups of actors is that the user of an actor is rarely interested in how the actor does its work. For example, a client using our server is not interested how the server splits the work and assigns it to several workers and how these do their work concurrently; the client is only interested in getting the correct result. In the following, we only talk about actor groups and not actor components[3], but we believe that the main aspects of what we develop also hold for components built from actors. Before we turn back to groups, we consider the specification of single actors.

3.1 Specifying Single Actors

Our basic specification construct is designed to express the behavior of an actor in reaction to a method selection. The construct has the following form:

$$\begin{array}{rl} \textit{input\_message} \Longrightarrow & \text{set of } \textit{actor\_creation}; \\ & \text{REGEXP}(\textit{output\_messages}) \\ \textbf{assume} & \textit{boolean\_expression} \\ \textbf{assert} & \textit{boolean\_expression} \end{array}$$

We call the specification construct a *reaction rule*, because it can be read as defining a transition in reaction to an input: When the actor receives the *input_message*, i.e., queued, and the assume clause evaluates to true, it can make the following transition. The actor creates the specified set of actors, sends the output messages as described by the regular expression over the *output_messages*, and guarantees that the assert clause holds in the poststate. Note that assumption A4 guarantees that a poststate exists. Using regular expressions to describe the sent output messages is just a convenience. In this paper we will only use finite sequences of output messages and assume, in particular, that the Kleene-star cannot be used.

The expression in the assume clause can refer to parameters of the input message and the instance variables of the actor. The default for missing assume clauses is **`true`**. The expression in the assert clause can refer to parameters of the input message and to the instance variables of the actor in the pre- and poststate. The value of an instance variable `v` in the prestate is denoted by `old(v)`. The default for missing assert clauses asserts that all instance variables `v` are unchanged, i.e., `v == old(v)`.

Before we discuss the construct further, let us consider specifications for our server and worker as examples.

The behavior of an actor `AServer` given in Fig. 1 is represented in the specification given in Fig. 3. On selection of a `serve` message, a server creates a new actor of

[3] mainly to stay out of a component discussion

```
actor spec AServer {
   this.serve( c, t ) ==>  w <- new AWorker;
                           w.do( t, taskSize(t) );
                           w.propagateResult( null, c )
   assumes c != null
}
```

Fig. 3. AServer actor specification

type `AWorker`, names it `w` and sends two messages to the newly created worker. This selection assumes that both message parameters are non-null. The workers are more interesting. It uses the instance variables of the actor class in the specification (according to JML terminology, they are `spec-public`; cf. [23]).

```
actor spec AWorker {
   Value  myResult = null;
   Worker nextWorker = null;

   this.do( t, n ) ==> empty
   assumes  n == 1
   asserts  myResult == compute(t) && nextWorker == null

   this.do( t, n ) ==> w <- new AWorker;
                       w.do( restTask(t,n), n-1 )
   assumes  n > 1
   asserts  myResult == compute( firstTask(t, n) ) && nextWorker == w


   this.propagateResult( v, c ) ==> c.receive( merge(myResult,v) )
   assumes  myResult != null && nextWorker == null

   this.propagateResult( v, c ) ==>
                nextWorker.propagateResult( merge(myResult,v), c )
   assumes  myResult != null && nextWorker != null
}
```

Fig. 4. `AWorker` actor specification

The figure above states he behavior of the `AWorker` actor in Fig. 2. On selection of a `do` message with actual parameter `n > 1`, a worker creates another worker and asks it to `do` the remaining $(\texttt{n} - 1)$ parts of the computation task. The worker itself computes the first part and updates its state accordingly. On selection of a `do` message with actual parameter `n == 1`, a worker computes the given task; it does not have any effect on the environment.

If the return from a procedure/method execution is considered as sending an outgoing message back to the original caller, our specification construct can be understood as a generalization of pre-/post-specifications. The central differences are that the number of outgoing messages can also be zero or several and that actor creation has to be explicitly specified as it affects the environment.

Semantical aspects. We assume that the semantics of actor programs is defined as sets of traces where a trace is a possibly infinite sequence alternating between configurations and events. A *configuration* captures the set of actors and their states. There are three types of *events*: output events for message sending, input events for message selection, and creation events for actor creation. The distinction between output and input events is necessary, because a message that never satisfies its guard is never selected. Events have an identity[4] in the sense that no events in a trace are equal.

The semantics of specifications is based on the semantics of actor systems. A reaction rule is interpreted as a mechanism expressing a causality relation between an input event e_i and a set $E = \{e_{c_1}, \dots e_{c_m}, e_{o_1}, \dots, e_{o_n}\}$ of creation and output events where causality means that

> *whenever we see an event e_i, eventually all events of E will appear in the subsequent trace*

In this weak form of causality, it might happen that for two input events e_i and e'_i the caused event sets E and E' have common events, i.e., that an output or creation event is "caused" by two different input events. In practice, this would lead to the undesirable situation that sending five times the same request to a server might result in only one response. Furthermore, this weak form is in contrast to our goal to generalize pre-/post-specification, as the actor specifications above guarantee that the number of calls always corresponds to the number of returns[5]. Most importantly, actor implementations never generate traces where an event is caused by two different input events. That is why we argue for a *strong causality* semantics in which

> *every event e either has a unique event that caused e or is an* external *event*

where output events are caused by the corresponding input event and external events are coming from outside the considered system (e.g., for starting up systems or for interacting with an unknown environment).

Discussion. From a view point of actor implementations, strong causality seems to be a natural basis for specifications. That is why it was a bit surprising to us to notice that strong causality cannot be expressed in classical temporal logic. To be more precise, when we project down the traces to the communicating actors (e.g., between a server and a client), the trace prefixes are non-regular. Specifying

[4] for example, the number of their position in the trace

[5] Recall that we assume termination

the strong causality needs counting facilities (see [21] for a temporal logic with counting).

The above specifications for `AServer` and `AWorker` slightly abstract from the behavior of the actor implementations and simplify the reasoning about these actors. However, from a client point of view, they are not of much interest. A client is not interested in how the server does its work, but wants to know that the server eventually returns the correct result for every request. That is, the client is interested in the behavior of the group of actors consisting of the server and the dynamically created workers. This is the topic of the next subsection.

3.2 Specifying Actor Groups

As illustrated above, there are typical scenarios in which a user of an actor is not interested in the work directly performed by the actor itself, but wants to have guarantees about the behavior that the actor achieves together with its helper actors. This is one reason to develop specifications for actor groups. Another reason is that group specification are important to formulate induction invariants for proofs about actor groups of varying sizes (in our example, the number of workers varies depending on the input). In summary, our position is that

> *specifications for actor groups are needed both as contracts for more complex actor groups and as important construct for reasoning.*

We illustrate both aspects. We like to stress that the aspects are closely related as specifications of single actors and actor groups are needed to prove specifications of larger actor groups in a hierarchical manner.

```
group spec AServer {
   this.serve(c,t) ==> c.receive( compute(t) )
   assumes  c != null
}
```

Fig. 5. AServer group specification

A central goal in the design of specifications of actor groups is that they are similar to specifications of single actors. For example, the specification of the group owned by an `AServer`-actor in Fig. 5 almost looks like an actor specification. It expresses exactly what a client wants to know. To achieve our design goal, we make two simplifying assumptions in this paper:

G1 Every actor a *owns* a group. The members of a's group are just the actors directly or indirectly created by a.
G2 Every group only exposes the reference of the owner to the group environment.

```
group spec AWorker {
   boolean  working = false;
   CompTask myTask = emptyTask;

   this.do( t, n ) ==> empty
   assumes  n >= 1
   asserts  myTask == t && working == true

   this.propagateResult( v, c ) ==> c.receive( merge(compute(mytask), v) )
   assumes  working
}
```

Fig. 6. `AWorker` group specification

The used notions of grouping are similar to simple ownership disciplines in OO programming (e.g., [33]). Because of assumption G1, we do not have to introduce constructs in AJ to define groups. Groups are defined implicitly. To be more realistic, one could distinguish at creation sites whether the new actor should be created in the local group or in the environment (ABS uses such a technique [19]). However, for illustrating our position, our primitive grouping mechanism is sufficient. Assumption G2 is a restriction on the actor programs we are allowed to write, and it cannot be checked automatically. In general, an actor group can expose references of several actors of the group to the environment. For example, the `receive`-message could pass a reference of a worker to the client. To eliminate G2, one would need to generalize the specification construct for actor groups.

Whereas the specification of an `AServer`-group illustrates a typical client-server contract, the specification of an `AWorker`-group in Fig. 6 describes the behavior of an `AWorker`-actor w together with the workers created by w. To express this behavior, the specification uses the variables `working` and `myTask`. The boolean flag `working` is a model variable that abstracts the instance variable `myResult`. It is true iff `myResult != null`. We use the model variable here to hide the implementation details of the worker. Variable `myTask` is a ghost variable storing the task worked on by the actor group. It is needed to formulate the reaction rule for message `propagateResult`. Model and ghost variables are well-investigated techniques in OO specification (see, e.g., [8]).

Semantical aspects. Group specifications look syntactically similar to single actor specifications. The semantics is also similar except that group internal events are hidden. That is why the specification of the `do` message does not list any caused events. More precisely, a reaction rule has to specify all output events with a receiver outside the group that are directly or indirectly caused by an input event. The reaction rule for message `propagateResult` is an example of an indirectly caused event, because the propagation might iterate through a number of other workers before the result is received by the client.

4 Modular Verification

In this section, we discuss how the developed specification techniques can be used for verification. The aim is to *illustrate* how a higher-level verification technique that does not work on the semantics or trace level can be conceived. To achieve this, we will leave many details undiscussed and do not claim that we present a formal proof. We describe how group specifications can be verified from other group and actor specifications. Thus, we have a modular, hierarchical verification technique in which we can use the verified specifications of actors and subgroups to prove enclosing groups. We look into the two essential cases:

- Verifying a group by composing the specifications of the subgroups and actors.
- Verifying a group by induction over its parameter values.

In this paper, we do not discuss the task of verifying that actor implementations satisfy their specifications. This is the classical problem that can be handled by adapted Hoare-style or dynamic logics.

To aid the presentation, the verification sketches will be accompanied with proof outlines. A proof outline has the form of a sequence whose elements are separated by the symbol ==>*. Each element consists of a list with state assertions, actor creations and output messages. State assertions are expressed as a predicate enclosed in curly brackets, representing the value of each variable of the actors and the message parameters. Messages have the same format as in the specifications. The symbol ==>* should be read as a combination of the transitive, reflexive closure of ==> and logical implication.

Compositional verification of group specification. In our server example, we have two groups that we need to verify: the AServer-group and the AWorker-group. Here, we verify the specification of the AServer-group (Fig. 5) to show compositional verification using the specifications of the AWorker-group (Fig. 6) and the AServer-actor (Fig. 3).

The property that we have to prove is that whenever an instance of the AServer-group receives a serve(c,t) request to compute the task t, the server sends back to the non-null client c the corresponding computation result. The proof outline for the AServer group is given in Fig. 7. The first step of the outline is directly obtained from the specification of message serve in Fig. 3. Using the group specification of the created worker and the property of taskSize (cf. Sect. 2) gives us the state expression. The next two steps are obtained by unfolding the do- and propagateResult-specifications of Fig. 6. Finally, we use a property of merge.

Inductive verification of group specification. Inductive verification of group specifications is in general more complex. In simple cases, the group specification is sufficient as an induction invariant. However, in more complex cases, structural invariants are needed in addition. For example, the proof of the worker group specification of Fig. 6 needs an invariant that links the construction of the actor structure with the result propagation.

```
    { c!= null }
    this.serve(c,t)
==>*
    w <- new AWorker
    w.do( t, taskSize(t) );
    w.propagateResult( null, c )
==>*
    { w.working==false && w.myTask==emptyTask && n==taskSize(t) && n>=1 }
    w.do( t, n );
    w.propagateResult( null, c )
==>*
    { w.myTask == t  &&  w.working == true }
    w.propagateResult( null, c )
==>*
    { w.myTask == t  &&  w.working == true }
    c.receive( merge(compute(w.myTask),null) )
==>*
    c.receive( compute(t) )
```

Fig. 7. Proof outline for the server verification

To discuss inductive proofs, we consider the actor group in Fig. 8 computing the factorial of a natural number. The basic proof outline of the induction step is as follows:

```
   { n>0 }
   this.fac(n,c)
==>*
   { n>0 && argument==n  &&  myCaller==c }
   f <- new AFactorial
   f.fac(n-1,this)
==>*
   { n>0 && argument==n  &&  myCaller==c }
   this.return(factorial(n-1))
==>*
   { n>0 && argument==n  &&  myCaller==c }
   c.return(factorial(n-1)*n)
==>*
   c.return(factorial(n))
```

In the first step, we use the property of the actor specification. Then, we use the group specification and finally the actor specification again. Of course, this form of reasoning has to be complemented with mechanisms controlling undesired outside interaction either by always working with new actor instances or by blocking incoming messages as long as critical computations are going on in a group.

```
interface Caller { void return( Int ); }
interface Factorial extends Caller { void fac( Int n, Caller c ); }

actor class AFactorial implements Factorial { ... }

actor spec AFactorial {
   Int argument;
   Caller myCaller = null;

   this.fac( n, c ) ==> c.return(1)
   assumes  n == 0

   this.fac( n, c ) ==> f <- new AFactorial
                        f.fac( n-1, this )
   assumes  n > 0
   asserts  argument == n && myCaller == c

   this.return(res) ==> myCaller.return( res*argument )
   assumes  myCaller != null
}

group spec AFactorial {
   this.fac( n, c ) ==> c.return(factorial(n))
   assumes n >= 0
}
```

Fig. 8. An actor group computing the factorial function

5 Discussion of Related and Future Work

Actors provide a popular model for distributed and concurrent systems. Although researchers have worked for over twenty years on actor languages and verification, there are still interesting and open questions. In particular, we claim that the specification and verification techniques for non-local actor properties in dynamically evolving actor systems are not sufficiently developed. Technically, we proposed

- to use specifications that are based on strong causality and not on classical temporal logic and
- to support reasoning over hierarchical actor groups or components.

Next, we broaden the discussion of related work and mention plans for future work.

5.1 Related Work

Actor languages. Many actor languages are available (to name only some: ABS [19], AmbientTalk [11], CoBoxes [36], Creol [20], E [29], Erlang [3]) and the family is still growing. Furthermore, there existed a variety of actor libraries for programming languages. AJ can be seen as a simplified version of Creol [20] and ABS [19], without futures and return values and without the cooperative scheduling mechanism. However, many of the features can be expressed using additional state and self-messages, i.e., messages that are sent from an actor to itself. As with Creol or CoBoxes, it can be proved that the type system guarantees that type correct AJ actors know the messages that are sent to them.

Specification. Whereas in the OO community there is a large corpus of work about specification languages and techniques (see [17]), this is not true in the actor community. In [16], a technique for specifying and dynamically checking the interaction behavior of single actors is described. The technique uses attribute grammars for specifying the possible protocols an actor can engage in.

History-based specification techniques as described in, e.g., [9, 14] enable one to specify safety properties based on trace prefixes in a modular way. As strong causality property is a liveness property, these techniques cannot specify it without any extensions. An approximation of the intended `AServer`-group behavior (Fig. 5) is as follows.

```
ok([]) = true
ok(h; m) = true, WHERE m != c.resp(_)
ok(h; c.resp(res(t))) = okcnt(h; c.resp(res(t)), c.resp(res(t)), 0) && ok(h)

okcnt([], _, n) = (n >= 0)
okcnt(h; m, c.resp(res(t)), n) = okcnt(h, c.resp(res(t)), n)
  WHERE m != c.resp(res(t)) && m != this.req(t, c)
okcnt(h; this.req(t, c), c.resp(res(t)), n) = okcnt(h, c.resp(res(t)), n+1)
okcnt(h; c.resp(res(t)), c.resp(res(t)), n) = okcnt(h, c.resp(res(t)), n-1)
```

The execution invariant predicate `ok` keeps track of the number of requests and responses with the same parameters, assuming one cannot distinguish them (i.e., the events lack identities).

To represent liveness properties, an extension in form of *ready set* is presented in [9]. This set represents the set of events an actor (group) is ready to accept next. More investigation is needed to compare strong causality with ready set.

The notion of strong causality also appears in form of session types [37] and has been applied to a similar client/server context as described above [15, 13]. For example, using the syntax from MOOSE ([12]) the interaction of the `AServer`-group using a session type as given below.

```
session ClientRequest = begin.!CompTask.?Value.end
```

A client request is represented as the emittance of a task and receiving a value. Communication between a client and a server is done over a channel, and channels can be passed around creating the possibility of higher-order session. Because an interaction between actors is represented within a session, the client reference does not need to be sent. However, the types are an overapproximation of the intended behavior. In the example above, there is no information that the session indeed occurs exclusively between a client and a server (because channels can be passed around), and whether the resulting value should have any correspondence to the task until the `ClientRequest` type is coupled to the implementation. Our aim is to provide a precise specification of how a system and its components should interact. Furthermore, strong causality can be used to specify the link between events that do not occur within one session.

Somewhat further than session types lies π-calculus [30], which provides hierarchical means to specify concurrent systems, a necessity stated in our position. An adaptation to the actor setting will require an introduction of identities.

Other works dealing with specification of system level functional properties covering data and control aspects include the following. Broy [6] proposes a stream-based approach; in the architecture description language (ADL) field, we have, for example, SOFA [35] and Rapide [26]; and the specification of open distributed system in Erlang by Dam, Fredlund and Gurov [10]. As stated in [10], the literature mostly covers aspects of static systems, with fixed amount of component instances interacting within the execution of a system, lacking the instance creation aspects.

Verification. Dovland, Johnsen and Owe [14] and Ahrendt and Dylla [2] developed a Hoare logic and a dynamic logic, respectively, for Creol [20]. Formulation of a property of a simple program already requires a large formula, making it hard to be applied directly to specify system-wide property. Closely related is the work of Dam, Fredlund, and Gurov on verification of dynamically created Erlang processes [10]. They used a logic based on the first-order μ-calculus, but did not consider additional specification techniques.

5.2 Future Work

This paper introduced AJ as a core language acting as the base for specification and verification. The core language needs to include just the essential features that affect the specification and verification techniques. It would be interesting to analyze how other features like cooperative multi-tasking or futures as in ABS [19] could be expressed in AJ. On the other hand, we plan to extend our approach to directly support the modeling language ABS that has all these features in the language.

As hierarchical system is a necessary part of verifying dynamic actor systems, the specification language needs to deal with actor groups/components. The cases we have dealt with so far use a single actor as the entry point to a component. In general, a component may contain more than one actor which readily interact with its user. It is harder to compose the causality relations of actors of such a component for verification purposes.

Handling the strong causality specification construct requires an expressive logic, such as a higher-order logic. To strengthen our verification approach, we have embedded some parts of our formalization in Isabelle/HOL [34]. We are continuing this work to obtain full coverage of our specification and verification technique.

Acknowledgement The authors thank Mads Dam for discussion on strong causality and verification of non-local actor properties. In particular, he suggested to use a simplified version of the problem described in [4] as a running example. The authors also thank the anonymous referees for their valuable input.

References

1. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. J. Funct. Program. 7(1), 1–72 (1997)
2. Ahrendt, W., Dylla, M.: A system for compositional verification of asynchronous objects. Science of Computer Programming (2011)
3. Armstrong, J.: Erlang. Commun. ACM 53, 68–75 (2010)
4. Arts, T., Dam, M.: Verifying a distributed database lookup manager written in Erlang. In: World Congress on Formal Methods. pp. 682–700 (1999)
5. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: CASSIS. pp. 49–69. Springer (2004)
6. Broy, M.: A logical basis for component-oriented software and systems engineering. Comput. J. 53(10), 1758–1782 (2010)
7. Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous and deterministic objects. In: POPL. pp. 123–134 (2004)
8. Cheon, Y., Leavens, G.T., Sitaraman, M., Edwards, S.H.: Model variables: cleanly supporting abstraction in design by contract. Softw., Pract. Exper. 35(6), 583–599 (2005)
9. Dahl, O.J., Owe, O.: Formal development with ABEL. In: VDM Europe (2). pp. 320–362 (1991)
10. Dam, M., Fredlund, L.Å., Gurov, D.: Toward parametric verification of open distributed systems. In: COMPOS. pp. 150–185 (1997)
11. Dedecker, J., Cutsem, T.V., Mostinckx, S., D'Hondt, T., Meuter, W.D.: Ambient-oriented programming in AmbientTalk. In: ECOOP. pp. 230–254 (2006)
12. Dezani-Ciancaglini, M., Drossopoulou, S., Mostrous, D., Yoshida, N.: Objects and session types. Inf. Comput. 207(5), 595–641 (2009)
13. Dezani-Ciancaglini, M., Yoshida, N., Ahern, A., Drossopoulou, S.: A distributed object-oriented language with session types. In: TGC. pp. 299–318 (2005)
14. Dovland, J., Johnsen, E.B., Owe, O.: Observable behavior of dynamic systems: Component reasoning for concurrent objects. ENTCS 203(3), 19–34 (2008)
15. Gay, S.J., Hole, M.: Types and subtypes for client-server interactions. In: ESOP. pp. 74–90 (1999)
16. de Gouw, S., de Boer, F., Vinju, J.: Prototyping a tool environment for run-time assertion checking in JML with communication histories. In: FTfJP (2010)
17. Hatcliff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M.: Behavioral interface specification languages. Tech. Rep. CS-TR-09-01a (2010)
18. Inmos Corp: Occam Programming Manual. Prentice Hall Trade (1984)
19. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: FMCO 2010. LNCS, Springer (2011), to appear
20. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: A type-safe object-oriented model for distributed concurrent systems. Theor. Comput. Sci. 365(1-2), 23–66 (2006)
21. Laroussinie, F., Meyer, A., Petonnet, E.: Counting LTL. In: TIME. pp. 51–58 (2010)
22. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. ACM SIGSOFT Software Engineering Notes 31(3), 1–38 (2006)
23. Leavens, G.T., Cheon, Y.: Design by contract with JML (2006)
24. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: ECOOP. pp. 491–516 (2004)
25. Luckham, D.C., von Henke, F.W., Krieg-Brückner, B., Owe, O.: ANNA - A Language for Annotating Ada Programs, Reference Manual, LNCS, vol. 260. Springer (1987)

26. Luckham, D.C., Kenney, J.J., Augustin, L.M., Vera, J., Bryan, D., Mann, W.: Specification and analysis of system architecture using rapide. IEEE Trans. Software Eng. 21(4), 336–355 (1995)
27. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)
28. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall (1997)
29. Miller, M.S., Tribble, E.D., Shapiro, J.S.: Concurrency among strangers. In: TGC. pp. 195–229 (2005)
30. Milner, R.: Communicating and mobile systems - the Pi-calculus. Cambridge University Press (1999)
31. Misra, J.: A discipline of multiprogramming. ACM Comput. Surv. 28 (1996)
32. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. Ph.D. thesis, FernUniversität Hagen (2001)
33. Müller, P., Poetzsch-Heffter, A.: Universes: A type system for alias and dependency control. Tech. Rep. 279, Fernuniversität Hagen (2001)
34. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
35. Plasil, F., Visnovsky, S.: Behavior protocols for software components. IEEE Trans. Software Eng. 28(11), 1056–1076 (2002)
36. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing active objects to concurrent components. In: ECOOP 2010. pp. 275–299. LNCS, Springer (2010)
37. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: PARLE. pp. 398–413 (1994)

A 𝕂-Based Formal Framework for Domain-Specific Modelling Languages

Vlad Rusu[1] and Dorel Lucanu[2]

[1] Inria Lille, France, Vlad.Rusu@inria.fr
[2] University of Iasi, Romania, dlucanu@info.uaic.ro

Abstract. We propose a formal approach for the definition of domain-specific modelling languages (DSMLs). The approach uses basic Model-Driven Engineering artifacts for defining a DSML's syntax (using metamodels) and its operational semantics (using model transformations). We give formal meanings to these artifacts by mapping them to the 𝕂 semantic framework. The mapping is implemented in the Rascal metaprogramming language. Since the resulting 𝕂 definitions are executable, one obtains an execution engine for DSMLs and gains acces to 𝕂's formal analysis tools. We illustrate the approach on xSPEM, a language for describing the execution of tasks constrained by time, precedence, and resources.

1 Introduction

Domain-Specific Modelling Languages (DSMLs) are languages dedicated to modelling in specific application areas. Recently, the design of DSMLs has become widely accessible to engineers trained in the basics of Model-Driven Engineering (MDE): one designs a *metamodel* for the language's abstract syntax; then, the language's operational semantics is expressed using *model transformations* over the metamodel. The democratisation of DSML design catalysed by MDE is likely to give birth to numerous languages, and one can also reasonably expect that there shall be numerous errors in those languages. Indeed, getting a language right (especially its operational semantics) is hard, regardless of whether the language is defined in the modern MDE framework or in more traditional ones.

Formal methods can help detect or avoid errors in DSML definitions. However, the history of formal methods offers many examples of valorous methods that could not be transferred outside a circle of specialised users, because software engineers do not have the time or the background required for learning them. The lesson learned from these failures is that, in order to be accepted by software engineers, formal approaches have to operate with notions familiar to them.

We propose here such an approach, which formalises the basic MDE ingredients used in DSML definitions. From the point of vue a user, the approach is a black box (Figure 1): users can define their DSMLs using familiar MDE ingredients (metamodels for syntax, OCL [1] constraints for static semantics, model transformations for operational semantics). These inputs are parsed and processed by a Rascal [2] program (that we wrote) and are mapped to 𝕂 [3] code, together

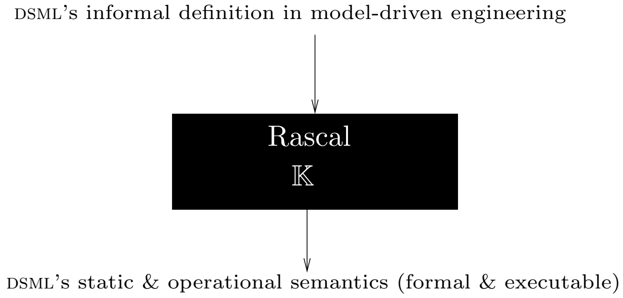

DSML's informal definition in model-driven engineering
Rascal
𝕂
DSML's static & operational semantics (formal & executable)

Fig. 1. Our approach, from the point of vue of a user defining a DSML.

with additional 𝕂 code (that we also wrote). In this way, users benefit from 𝕂's execution engine and formal analysis tools for free - without having to write code unfamiliar to them or unrelated to their task - which allows them to experiment with and to perform formal analyses on their languages in a transparent way.

We illustrate the approach on xSPEM [4], a DSML based on the OMG standard [5] for describing the execution of activities constrained by time, resources, and precedence relations. We show how users can automatically: check model-to-metamodel conformance (including OCL constraints); execute a DSML's semantics; and model check for reachability properties over the DSML's executions.

Contributions Our main contributions is providing a formal semantics to the MDE notions employed in DSML definitions, using the 𝕂 semantical framework:

- metamodels, together with well-formedness OCL constraints, and models;
- model transformations for operational semantics. We have designed for this a language called KMRL (𝕂-based Model-Rewrite Language) by building on basic MDE notions. KMRL is composed of model-rewrite rules, where each rule consists of a model pattern similar to MDE-models, an optional condition written in OCL, and an optional piece of imperative code also based on OCL. KMRL should therefore look and feel familiar to our target users: software engineers familiar with such basic MDE notions as model and OCL constraints.

Note that in our approach, the user *does not* define the semantics of her/his DSML directly in 𝕂. We do not advocate this, since 𝕂 is unlikely to be accepted by software engineers, and we have stated that the main motivation for this work is to gain their acceptance (and, ultimately, to further the cause of formal methods in practice). We supply instead automated means for that.

Organisation Section 2 provides preliminaries: it describes the 𝕂 semantical framework and the Rascal metaprogramming language, and illustrates the MDE-based definition of the xSPEM DSML. In Section 3 we present our approach and illustrate it on xSPEM. Section 4 concludes and presents related and future work.

2 Preliminaries

2.1 The $\mathbb{K}$ Framework and the Rascal Metaprogramming Language

$\mathbb{K}$ [3] is a framework mainly intended for defining and analysing semantics of programming languages[3]. The main features of $\mathbb{K}$ include:

- *executability*: the definitions are directly executable in order to be experimented with and analysed;
- *unique definition*: there is only one definition for a language, and several analysis tools that are sound with respect to this definition;
- *program logic*: the framework serve as a program logic with which the programs can be verified and analysed (see, e.g., [6]).

A $\mathbb{K}$ definition has three main ingredients: a *configuration*, which is a structure of nested cells abstracting the state structure of the machine on which the programs are executed; *computations*, which are sequences of tasks derived from the annotated syntax; and $\mathbb{K}$ *rules*, which describe the computation steps using minimal information (only what is needed for matching and rewriting). These concepts will be illustrated in Section 3, where we show how the DSML definitions can be formalized in $\mathbb{K}$. Then, the $\mathbb{K}$ execution engine can be used for executing the definition, and its model checker, for checking reachability properties.

Rascal [2] is a recent metaprogramming language for source code analysis and transformation. We use it to implement the mapping of the MDE concepts used in DSML definitions (metamodels, OCL, models, model transformations) to $\mathbb{K}$.

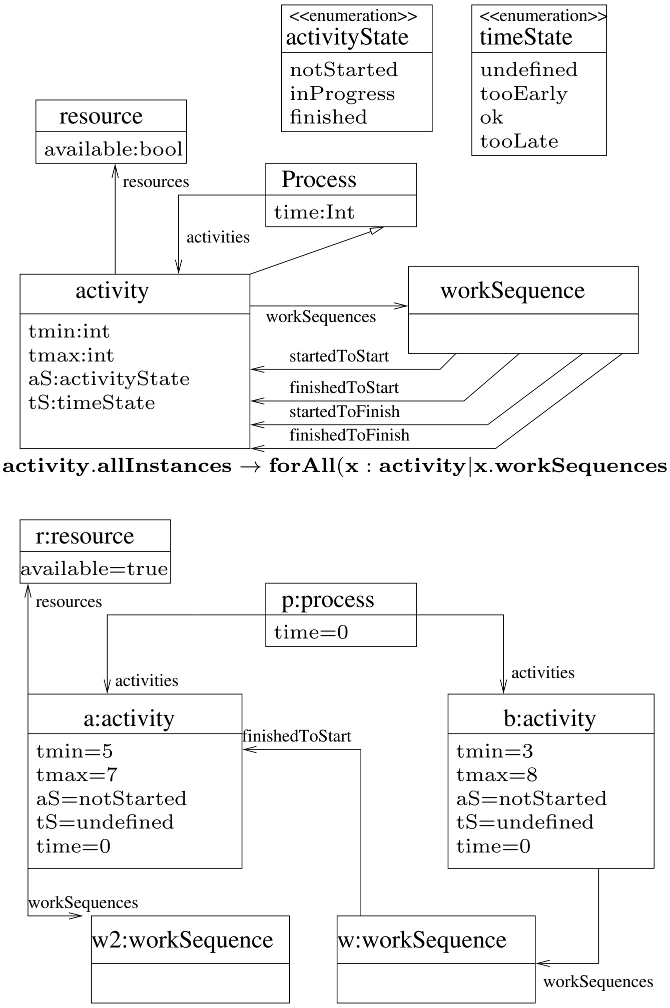
2.2 Defining a DSML using MDE: xSPEM

We illustrate our approach on a DSML called xSPEM [4], which is an executable version of the SPEM language standard [5]. The language describes the execution of *activities* constrained by time, resources, and precedence relations.

We first describe the syntax and static semantics of (a simplified version of) xSPEM by showing its metamodel as well as a sample model. Then we describe the language's operational semantics using a mixture of graphical and textual notations. These notations will become formal when we represent them in $\mathbb{K}$.

In the metamodel of Figure 2 (top), *activity* is the class of entities being executed. The *tmin* and *tmax* attributes of the *activity* class denote the minimum and maximum expected duration of activities. The *aS* attribute takes its values in the *activityState* enumeration: *notStarted*, *inProgress*, or *finished*; and the *tS* attribute takes its values in the *timeState* eumeration: *undefined*, *tooEarly*, *ok*, or *tooLate*. An activity may also have *resources*, which are reserved by the activity (and become unavailable to others) while the activity is running. In addition to the availability of resources (*resource* class) the execution of activities is also governed by explicit ordering constraints (*workSequence* class). Each activity also has exactly one *workSequence* instance, as indicated by the OCL invariant associated to the metamodel. The *workSequence* class has references to four possibly empty sets of activities, namely, the activities that must be

[3] e.g., a definition of C is available at `http://code.google.com/p/c-semantics/`.



INV: $\mathbf{activity.allInstances} \rightarrow \mathbf{forAll(x : activity|x.workSequences.size() = 1)}$

Fig. 2. (top): A simplified metamodel for xSPEM, together with an OCL constraint (bottom) one xSPEM model that conforms to the xSPEM metamodel.

- started for the current activity to start (*startedToStart* reference);
- finished for the current activity to start (*finishedToStart* reference);
- started for the current activity to finish (*startedToFinish* reference);
- finished for the current activity to finish (*finishedToFinish* reference).

Time is measured by a clock, encoded by the *time* attribute of the *process* class. Activities inherit from processes (inheritance is denoted in class diagram by an open arrrow). Hence, activities also have the *time* and *activities* features.

In the model depicted at the bottom Figure 2, the activities a and b are linked by a *workSequence* via the reference *finishedToStart*, meaning that b is allowed to start only when a is finished; and a has the (available) resource r.

We now give the operational semantics of xSPEM using a semi-formal notation mixing graphical and textual rules. The notation is isomorphic to the (textual) KMRL language that we shall formalise in Section 3.5. The first rule is shown in

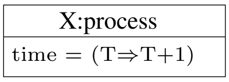
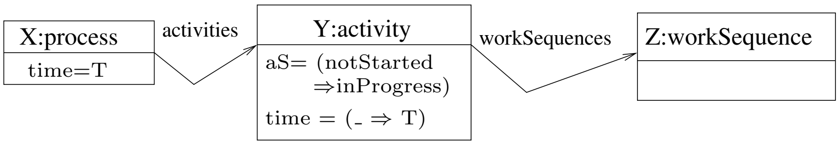


Fig. 3. Time-passing rule.



when $Z.\mathit{finishedToStart} \rightarrow \mathit{forAll}(u : \mathit{Activity}|u.aS = \mathit{finished})\wedge$
$\quad Z.\mathit{startedToStart} \rightarrow \mathit{forAll}(v : \mathit{Activity}|v.aS = \mathit{inProgress})\wedge$
$\quad Y.\mathit{resources} \rightarrow \mathit{forAll}(r : \mathit{Resource}|r.\mathit{available} = \mathit{true})$
then $\mathit{for}(r \leftarrow Y.\mathit{resources})\{r.\mathit{available} \leftarrow \mathit{false}\}$

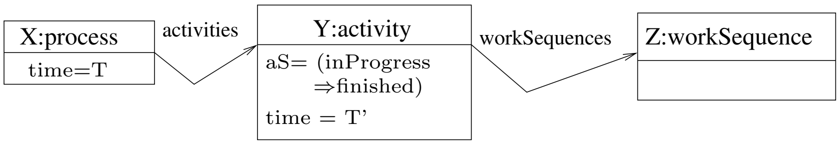
Fig. 4. Starting an activity.

Figure 3. It expresses the fact that processes "make time pass": the *time* attribute is increased by one. This is expressed by a local rewrite rule (a concept inspired from $\mathbb{K}$): $\mathit{time} = (T \Rightarrow T + 1)$, within a process X. Here, X and T are variables (of type process, respectively, integer) to be matched with correspondingly typed constants in a model; when this happens, the model is rewritten, just like in usual rewrite systems. For example, the execution of this rule on the model shown at the bottom of Figure 2 would produce the same model except that $p.\mathit{time} = 1$.

The next rule (Figure 4) expresses the starting of an activity. When an activity Y is started, the value of its *aS* attribute is rewritten from *notStarted* to *inProgress*, and the activity Y memorises in its *time* attribute the value of the homonymous attribute of the process, say X, which owns the activity. This is expressed by the local rewrite rule $\mathit{time} = (\_ \Rightarrow T)$, which says that the *time* attribute of X is rewritten, from whatever value it had, to T; where "whatever" is denoted by an underscore, and T equals the value of $X.\mathit{time}$. Moreover, the activity Y may only be started if the (optional) *when* condition holds; and, finally, some additional imperative code, in the (optional) *then* clause, is executed.

Here, the *when* clause says that the activities that have to be started (resp. finished) for the current activity Y to start are indeed in the expected states, and that all the resources of Y are available. The additional code in the *then* clause is here an imperative *for* loop, which is in charge of assigning all the activity's resources *available* attributes to *false*. In general, the additional code can include assignments, loops, and conditionals, and can declare local variables for storing intermediate values. The practical utility of the imperative code can be illustrated on the current example: when an activity is started, it needs to make all its resources un-available to other activities; but this cannot be expressed in a graphical rewrite pattern, because an activity may have any number of resources; whereas a graphical pattern "draws" a fixed number of model elements.

The semantics of xSPEM includes one other rule in addition to the two ones shown above. The third rule is in charge of finishing activities. It is shown in Figure 5. The main difference with the rule for starting an activity lies in the more complex imperative code, which is here used for updating the *tS* attribute



```
when $Z.finishedToFinish \to forAll(u : Activity | u.aS = finished) \wedge$
        $Z.startedToFinish \to forAll(v : Activity | v.aS = inProgress)$
then $var\ x;\ x \leftarrow T - T';$
      $if (x < Y.tmin)\ Y.tS \leftarrow tooEarly$
        $else\ if\ (x < Y.tmax)\ Y.tS \leftarrow ok$
          $else\ Y.tS \leftarrow tooLate$
        $endif$
      $endif;$
      $for(r \leftarrow Y.resources)\{r.available \leftarrow true\}$
```

Fig. 5. Finishing an activity.

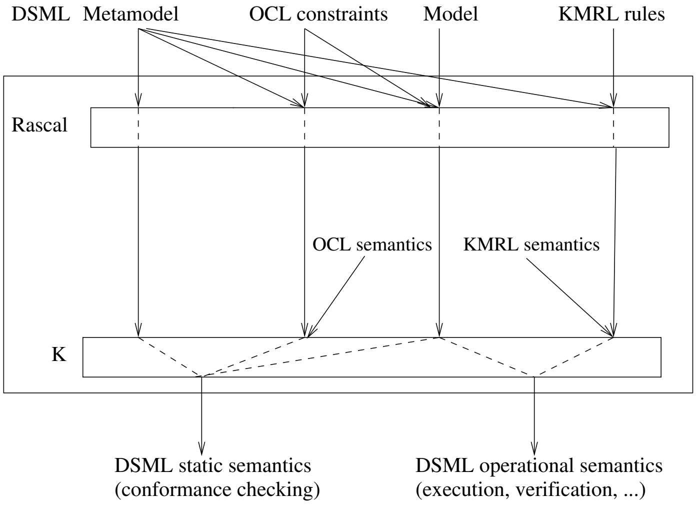


Fig. 6. Dataflow diagram of our framework.

of the activity Y being terminated, to *tooEarly*, *ok*, or *tooLate*, depending on whether its execution time is in $[0,tmin)$, $[tmin,tmax)$ or $[tmax,\infty)$, respectively. This is achieved using two nested *if-then-else-endif* conditionals. In order to avoid recomputing the execution time $T - T'$, we store it in the local variable x.

Hence, we have a flexible and expressive declarative/imperative language for describing operational semantics of DSMLs. In order to make it formal and executable we map it (together with languages for expressing metamodels and models for DSMLs) to the $\mathbb{K}$ framework, which we now briefly introduce.

3 A $\mathbb{K}$-Based Formal Framework for DSMLs

3.1 A General Overview

In this section we show how MDE concepts used in defining DSMLs can be mapped to $\mathbb{K}$. The mapping is implemented in the Rascal metaprogramming language [2].

A dataflow diagram of our approach is shown in Figure 6. It consists of:

```
metamodel xSPEM{
  enumeration activityState{notStarted; inProgress; finished}
  enumeration timeState{undef; tooEarly; ok; tooLate}
  class process{
    attribute time : int;
    attribute tS : timeState;
    reference activities : activity;
  }
  class activity extends {process}{
    attribute tmin : int;
    attribute tmax : int;
    attribute aS : activityState;
    reference workSequences : workSequence [1-1];
    reference resources : resource;
  }
  class workSequence{
    reference startedToStart : activity;
    reference startedToFinish : activity;
    reference finishedToStart : activity;
    reference finishedToFinish : activity;
  }
  class resource{
    attribute available : bool;
  }
}
```

Fig. 7. Textual representation of the xSPEM metamodel from Figure 2 (top).

- a Rascal program, which takes as input the ingredients of a DSML definition (metamodel, OCL constraints, models, KMRL rules) in a certain textual format, and produces intermediate representations suitable for $\mathbb{K}$;
- a $\mathbb{K}$ program, which takes the output produced by Rascal, together with $\mathbb{K}$ additional code defining the semantics of OCL and KMRL, and generates executable static semantics and operational semantics for the DSML.

Note that from the point of view of DSML designers, the outmost box is a black box: they do not need to know what is inside, but only need to provide the MDE artefacts for the definition of their DSML in a textual format; then, the static and operational semantics of the DSML is automatically generated for them.

We now describe the framework in detail by "scanning" the diagram in Figure 6 from left to right, and illustrate it on the xSPEM language from Section 2.2.

3.2 Metamodels and OCL constraints

We show in Figure 7 the textual syntax for the xSPEM metamodel from Figure 2. In order to generate input for $\mathbb{K}$, the metamodel in textual syntax is processed by a Rascal program. This includes parsing and some syntactical transformations, such as the replacement of multiplicity constraints and bidirectionality constraints by equivalent OCL invariants. For example, the `[1-1]` multiplicity of the `workSequences` reference is replaced by the equivalent OCL invariant

```
allInstances(activity)→forAll(x:activity|x.workSequences.size() = 1).
```

The metamodel is thus stripped of its multiplicity and bidirectionality constraints and the result is translated to a set of $\mathbb{K}$ function declarations together with rules to evaluate them. The functions encode all the (stripped) metamodel's information: for each enumeration, its set of values, and for each class, its children classes, its attributes with their types, and its references with their types. They are used for checking the syntactical correctness of models with respect to the given metamodel. This is discussed in more detail in Section 3.3.

Well-formedness OCL constraints In addition to the implicit constraints induced by multiplicities and bidirectionality of references, a metamodel may include other OCL constraints, which enforce well-formedness requirements that models conforming to the metamodel must satisfy. For example, we shall require that in any well-formed xSPEM model there is exactly one "proper" process:

```
(allInstances(process) \ allInstances(activity)).size() = 1.
```

3.3 Models

In DSML terminology, models can be seen as "DSML programs", by analogy with the programs of usual programming languages. In this section we show the syntax of models required as input by our Rascal module in charge of processing models, and the representation of the models as $\mathbb{K}$ configurations generated by the module in question. In the next section we outline of the $\mathbb{K}$ semantics for OCL, and show how model-to-metamodel conformance checking is done in $\mathbb{K}$.

We also use textual language for the model description. The essential information about a model consists of the name of the metamodel it must conform to and a collection of objects (class instances). Each object is described by its name, the class it belongs to, and the values of its attributes and references.

In Figure 8 we give the equivalent textual syntax for a fragment of the xSPEM model example given in Figure 2. Using this input together with the information obtained from a metamodel and OCL constraints, a Rascal module generates a $\mathbb{K}$ configuration described as follows.

$\mathbb{K}$ **Configuration for DSMLs** $\mathbb{K}$ configurations are structures consisting of nested cells. The generic $\mathbb{K}$ configuration for DSMLs is graphically represented in Figure 9. The configuration includes: a cell $\langle \_ \rangle_{\mathsf{model}}$ for models (described later in this section); a cell $\langle \_ \rangle_{\mathsf{oclConstraint}}$ containing OCL constraints to be checked on the model; a cell $\langle \_ \rangle_{\mathsf{k}}$ containing computation tasks to be performed on the model (conformance checking, model execution, model checking, ...); and a cell $\langle \_ \rangle_{\mathsf{result}}$ for storing the results of the computation tasks. Initially, the cells are empty (as denoted by periods and dashes within them, depending on their type). They are filled by Rascal modules: the $\langle \_ \rangle_{\mathsf{model}}$ cell is filled by the module in charge of models, and cell $\langle \_ \rangle_{\mathsf{oclConstraint}}$ cell is filled by the module in charge of OCL constraints of the metamodel to which the model is supposed to conform.

```
onexSPEM = new xSPEM {
  p = new process {
    time = 0;
    activities = {a b};
  }
  a = new activity {
    tmin = 5;  tmax = 7;
    aS = notStarted; tS = undef;
    resources = {r}; time = 0;
    linkToPredecessor = {w2}; activities = {};
  }
  b = new activity {... // similar to activity a
  }
  r = new resource {
    available = true;
  }
  w1 = new workSequence {
    startedToStart = {}; startedToFinish = {};
    finishedToStart = {a};  finishedToFinish = {};
  }
  w2 = new workSequence { ... // similar to workSequence w1
  }
}
```

Fig. 8. Textual representation of the model from Figure 2.

The structure of the cell $\langle \_ \rangle_{\mathsf{model}}$ is similar to that of the textual language we use for model description. It consists of a set of $\langle \_ \rangle_{\mathsf{instance}}$ cells, each of which contains the instance's name, its class, and its attribute/reference values.

3.4 $\mathbb{K}$ Semantics of OCL

We have defined in $\mathbb{K}$ a substantial fragment of OCL based on the standard [1]. Due to limited space, a complete description will be given in a separate paper.

The elementary types in our definition of OCL are integer, string, and Boolean with the usual elementary operations on them. We also allows for collection types, built using navigation through attributes/references, iterators (`select`, `collect`), quantifiers (`forAll`, `exists`), as well as the usual set operations. The allInstances() query returns all the instances of a given class. This provides us with a rich language for constraints, which we may enrich in the future.

Here is, for instance, the $\mathbb{K}$ rule giving semantics to the query allInstances():

$$\langle \frac{\texttt{allInstances(Cls)}}{\texttt{val(collectAllInstanceNames(Cls , children(Cls) , M))}} \cdots\rangle_{\mathsf{k}}$$

$$\langle \texttt{M} \rangle_{\mathsf{model}}$$

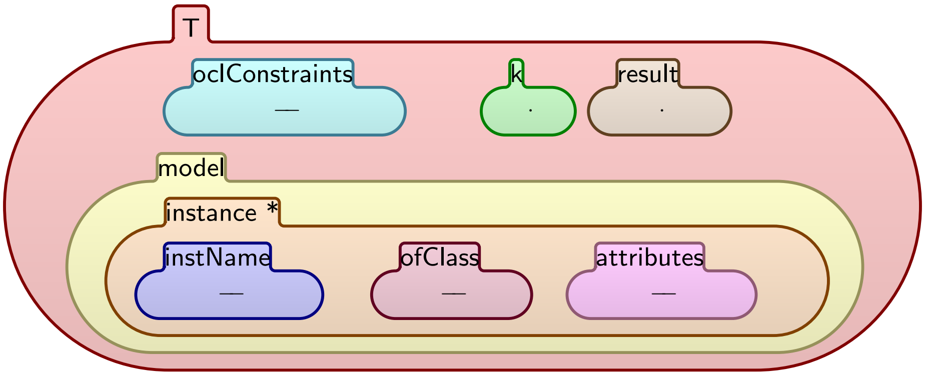


Fig. 9. $\mathbb{K}$ Configuration for DSMLs

The rule says that, in order to compute all the instances of a given class `Cls` in a model `M`, a helper function `collectAllInstanceNames()` must be called, with three parameters: the class `Cls`, its children `children(Cls)`, and the model `M`.

The fraction line denotes a local rewrite, here, at the the head of the computation cell $\langle\_\rangle_{\mathsf{k}}$. The numerator is what is usually written in the left-hand side of rewrite rules, and the denominator is the equivalent of the right-hand side. A $\mathbb{K}$ rule may have several local rewrites in various configuration cells, and it can use other cells just for providing context of the rewrite. For example, in the above rule, the cell $\langle\_\rangle_{\mathsf{model}}$ provides the model `M`. The operator `children()` is a part of the $\mathbb{K}$ description of the metamodel, and the function `collectAllInstanceNames()` traverses the $\langle\_\rangle_{\mathsf{instance}}$ cells of `M` and collects their names. The semantics of the later function is given by a set of similar $\mathbb{K}$ rules. Finally, the `val` operator wraps the result in order for $\mathbb{K}$ to interpret it as a fully eveluated OCL value (in our case, a collection of instance names).

As another axample, we give the $\mathbb{K}$ semantical rules for the `forAll` operation in order to illustrate some interesting features of $\mathbb{K}$. Syntactically, the operation is written `Col->forAll(Id | Exp)`, and its meaning is that it is *true* if and only if the third argument `Exp` evaluates to *true* on each element (denoted by the second argument `Id`) of the first argument `Col`. Its $\mathbb{K}$ semantical rules are

$$\langle \frac{\texttt{val(}\cdot\texttt{)->forAll(Var | Exp)}}{\texttt{true}} \cdots\rangle_{\mathsf{k}} \tag{1}$$

$$\langle \frac{\texttt{val(Hd, Tl)->forAll(Var | Exp)}}{\texttt{if Exp(Hd/Var) then val(Tl)->forAll(Var|Exp) else false}} \cdots\rangle_{\mathsf{k}} \tag{2}$$

The first rule describes the base case, when the first argument is an empty collection. The second rule describes the inductive step: when the first argument is a nonempty collection of the form `val(Hd, Tl)`, the result depends on the value of the expression `Exp` on the element `Hd`. This value is computed by applying the $\mathbb{K}$ visitor pattern [3] to perform substitution of `Var` by `Hd` in `Exp`. If the value is *true*, then the overall result is that of the `forAll` operation, recursively evaluated on the smaller collection `val(Tl)`; otherwise, the overall result is *false*.

This relatively simple definition is possible due to a powerful mechanism of $\mathbb{K}$, which automatically generates rewrite rules in order to evaluate arguments of operators declared to be "strict". The actual $\mathbb{K}$ grammar for `forAll` is

```
Exp ::= Exp ->forAll( Id  | Exp ) [strict (1)]
```

This means that the `forAll` operator is "strict" in its first argument; that is, this argument will be evaluated before the `forAll` expression is evaluated.

In order to do this, $\mathbb{K}$ automatically generates "heating rewrites" of the form

$$E_1 \texttt{ ->forAll(} V \mid E_2 \texttt{)} \quad \longrightarrow \quad E_1 \curvearrowright \square \texttt{ ->forAll(} V \mid E_2 \texttt{)}$$

by which the first computation task becomes the evaluation of the first argument E_1; this can generate further computation tasks using the same mechanism, depending on the structure of E_1 (as it is the case with the `if-then-else` expression in Rule 2). When E_1 is evaluated, "cooling rewrites" of the form:

$$\texttt{val(} L \texttt{)} \curvearrowright \square \texttt{ ->forAll(} V \mid E_2 \texttt{)} \quad \longrightarrow \quad \texttt{val(} L \texttt{)->forAll(} V \mid E_2 \texttt{)}$$

fill the "hole" left by E_1. Then, eventually, one of the rules (1–2) finishes the evaluation of the `forAll` operator.

Conformance Checking A model is well-formed if and only if it conforms to its metamodel, i.e., it is syntactically correct and satisfies the OCL constraints of the metamodel. Specifically, for a DSML, the class diagram of its metamodel defines its syntax, and the OCL constraints define its static semantics. The procedure for verifying that a model is well-formed is referred to as *conformance checking.*

In the rest of this section we briefly describe conformance checking in $\mathbb{K}$. Syntactical correctness is checked using the $\mathbb{K}$ functions specifying the metamodel. The OCL constraints are checked by first "loading" the content of the cell $\langle \_ \rangle_{\mathsf{oclConstr}}$ into the computation cell $\langle \_ \rangle_{\mathsf{k}}$. This is triggered by the following rule:

$$\langle \underset{e}{\underline{\texttt{checkConformance}}} \rangle_{\mathsf{k}} \ \langle e \rangle_{\mathsf{oclConstr}}$$

Then, the execution of the OCL semantics generates a computation of the form:

$$\begin{aligned} &\langle \langle e \rangle_{\mathsf{k}} \ \langle e \rangle_{\mathsf{oclConstr}} \ \langle \cdot \rangle_{\mathsf{mem}} \ \langle \cdot \rangle_{\mathsf{result}} \ \langle M \rangle_{\mathsf{model}} \ \cdots \ \rangle_{\mathsf{T}} \xrightarrow{*} \\ &\langle \langle \cdot \rangle_{\mathsf{k}} \ \langle e \rangle_{\mathsf{oclConstr}} \ \langle \cdot \rangle_{\mathsf{mem}} \ \langle v \rangle_{\mathsf{result}} \ \langle M \rangle_{\mathsf{model}} \ \cdots \ \rangle_{\mathsf{T}} \end{aligned}$$

in which the result v of the OCL constraints e is placed in the $\langle \_ \rangle_{\mathsf{result}}$ cell.

Finally, a model M satisfies the OCL constraints e if and only if

$$\begin{aligned} &\langle \langle \texttt{checkConformance} \rangle_{\mathsf{k}} \ \langle e \rangle_{\mathsf{oclConstr}} \ \langle \cdot \rangle_{\mathsf{result}} \ \langle M \rangle_{\mathsf{model}} \ \cdots \ \rangle_{\mathsf{T}} \xrightarrow{*} \\ &\langle \langle \cdot \rangle_{\mathsf{k}} \ \langle e \rangle_{\mathsf{oclConstr}} \ \langle \cdot \rangle_{\mathsf{mem}} \ \langle \mathit{true} \rangle_{\mathsf{result}} \ \langle M \rangle_{\mathsf{model}} \ \cdots \ \rangle_{\mathsf{T}} \end{aligned}$$

This provides us with a formal, executable definition for conformance checking.

```
rule start
forAll P Y Z timeVal
pattern {
  process P = {
    time = timeVal;
    activities = {Y _};
  }
  activity Y =  {
    aS = (notStarted => inProgress);
    time  = (0 => timeVal);
    linkToPredecessor  = {Z};
  }
}
when {
  Z.startedToStart->forAll(act| act.aS == inProgress);
  Z.finishedToStart->forAll(act| act.aS == finished);
  Y.resources->forAll(r| r.available == true)
}
then {
 var r;
 for (r <-Y.resources){(r.available) <- false} ;
 print("start"); print(Y)
}
```

Fig. 10. Textual version of the *start* from Figure 4.

3.5 Operational Semantics

The last ingredient in the definition of a DSML is the definition of its operational semantics. We propose for this the language KMRL ($\mathbb{K}$ Model-Rewrite Language), a mixed declarative/imperative language for model rewriting. A glimpse of KMRL has already been shown in Section 2.2, where we informally gave the semantics of xSPEM using (graphical) model-rewrite rules. To formalise the semantics of DSMLs, we need first to formalise the KMRL language itself. We provide it with a textual syntax, checked by a Rascal-generated parser, and with a formal semantics as a set of $\mathbb{K}$ rules. Those rules include the rules for the semantics of OCL since OCL is a sublanguage of KMRL.

We now illustrate the overall process with the rule *start* shown in Figure 4. The textual version of the rule is given in Figure 10. It is composed of four parts:

- global variable declaration (`forAll` keyword). Here, the notion of global variable[4] should be understood as variable in the rewrite-systems terminology: variables match terms, and are replaced by those terms when rewriting is performed; e.g., on our example, the variable `Y` matches one element of `acivities`'s value (which is a collection). The scope of global variables is the whole rule; they can occur everywhere in the rule.

[4] We shall call global variables just "variables" when no confusion with local variables, to be introduced later, can occur. We shall do the same for local variables.

- rewrite pattern (`pattern` keyword). Patterns are very much like models, discussed in the previous section. The main difference is that rewrite patterns need not be completely specified models; informally speaking, they may match any model that is a "superset" of the pattern. Also, unlike models, rewrite patterns may refer to variables (those declared in the global variable declarations) in addition to constants; and their attributes and references can be local rewrite rules[5]. For example, the `time` attribute of the activity `Y` is rewritten from `0` to `timeVal`, where the latter is a variable, which was chosen to be the value of the `time` attribute of the process `P` of our pattern. This means that when the rule is applied, the `time` attribute of the activity `Y` gets assigned the value of the `time` attribute of the process `P`. Note also the value of the `activities` reference of `P`: the meaning of {`Y_`} is a set containing at least the value `Y` and possibly more. This is in contrast to, e.g., the set {`Z`}, which means the set containing exactly the value(s) as given.
- OCL condition (`when` keyword). This is a condition in the sense of conditional rewrite systems: a rule is applied only when its condition holds after its (global) variables (here, `Y` and `Z`) are substituted with the matched (sub)terms.
- imperative code (`then` keyword). This is essentially a program composed of assignments, loops, and conditionals. The program starts with the declaration of a list of local variables, distinct from the global variables, which play the roles of usual variables in imperative programs: essentially, they serve for storage of intermediate computed results and as iterators of loops. In our example, the variable `r` serves as an iterator for the `for` loop. Regarding assignments, their left-hand side are OCL navigation expressions, i.e., expressions of the form $Variable.reference_1.\cdots.reference_n$, where *Variable* can be a global or a (previously evaluated) local variable (the latter case appears within the `for` loop of our example). Their right-hand sides can be arbitrary OCL expressions, including local variables. Finally, note the `print` statements: their arguments are arbitrary OCL expressions, and they are used to print output - useful for executing, debugging, or verifying KMRL code.

Parsing and processing KMRL with Rascal. The KMRL input defining the operational semantics of a DSML, together with the corresponding metamodel for the DSML's syntax, is processed by a Rascal program in three main steps.

The first step is to compute the types of the (global) variables and constants occurring in the rule: in our example, Rascal infers from the metamodel that `timeVal` is an integer, and that `inProgress` is an `activityState`. The second step is a form of "context transformation", also inspired from $\mathbb{K}$ but considerably simpler: since the rewriting pattern and its components are typically incomplete, we complete them with adequately typed variables; for example, the incomplete set {`Y,_`} is completed to {`Y, rest`}, where `rest` is a fresh variables that can match any number of set elements. And, finally, the third processing step consists in separating the rule's rewriting pattern into a left-hand and a right-hand side,

[5] This idea is borrowed from $\mathbb{K}$. The advantage is that produces simple/compact rules.

which has indeed the effect of turning the pattern into a proper rewrite rule. (Essentially, the right-hand side of a rewrite pattern is a copy of the pattern in which the left-hand sides of the local rewrite rules occurring in it are kept; and symetrically so for the right-hand sides. Those parts of the patttern that are not local rewrite rules appear in both sides, and serve as a rewriting context.)

Let r be a KMRL rule. We denote by `lhs(r)` and `rhs(r)` its left-hand and right-hand sides computed by Rascal (as described above), by `C(r)` its condition, and by `imp(r)` its imperative part. These are translated by Rascal to a $\mathbb{K}$ format which we do not present here since it is not essential for understanding. What is important is the the overall translation of the rule r, denoted by $\mathbb{K}(r)$:

$$\Big\langle \underset{\texttt{when(C(r))}\curvearrowright\texttt{update(rhs(r))}\curvearrowright\texttt{apply(imp(r))}\curvearrowright\texttt{run}}{\underline{\texttt{run}\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}}\ \cdots\Big\rangle_{\mathsf{k}} \quad \langle\texttt{lhs(r)}\rangle_{\mathsf{model}}$$

This means that whenever the keyword *run* (which is by convention the instruction for model execution) is at the top of the $\langle\_\rangle_{\mathsf{k}}$ cell, and the $\langle\_\rangle_{\mathsf{model}}$ cell matches `lhs(r)`, the *run* keyword is rewritten into a sequence that evaluates the condition `C(r)`, and, if the condition holds, then it updates the $\langle\_\rangle_{\mathsf{model}}$ cell by replacing its contents with `rhs(r)`, it applies the imperative program `imp(r)` to the result, and, finally, it recursively invokes model execution by reinserting `run` in the $\langle\_\rangle_{\mathsf{k}}$ cell.

This effect is obtained by the $\mathbb{K}$ semantics of KMRL we now briefly describe.

$\mathbb{K}$ semantics of KMRL First, we give the rules for the `when` clause of $\mathbb{K}(r)$.

$$\langle \underset{\cdot}{\underline{\texttt{when(true)}}}\ \cdots\rangle_{\mathsf{k}} \qquad\qquad \langle \underset{\texttt{run}}{\underline{\texttt{when(false)}\curvearrowright K \curvearrowright\texttt{run}}}\ \cdots\rangle_{\mathsf{k}}$$

That is, the `when` clause disappears (i.e., rewrites to $\cdot$) when its argument evaluates to `true`, and it discards the computational tasks corresponding to the current matched rule otherwise. This simplicity is due to the fact that the `when` operation is strict: its argument (an OCL expression) is a Boolean when the `when` clause itself is evaluated. The evaluation of the argument consisted in applying the $\mathbb{K}$ rules for the semantics of OCL.

Then comes the rule for the `update` instruction. It simply consists in removing the `update` keyword from the top of the $\langle\_\rangle_{\mathsf{k}}$ cell, and by replacing the content of the $\langle\_\rangle_{\mathsf{model}}$ cell by the argument of the (just removed) `update` instruction:

$$\langle \underset{\cdot}{\underline{\texttt{update(M)}}}\ \cdots\rangle_{\mathsf{k}}\ \langle \underset{\texttt{M}}{\underline{\ \_\ }} \rangle_{\mathsf{model}}$$

Finally, there are rules for applying the imperative part `imp(r)` of $\mathbb{K}(r)$. We do not give these rules due to lack of space, but note that, except for assignment (which is quite specific to our present model-based framework) they are standard semantical rules for imperative programs constructs (loops and conditionals).

```
rule observer
  pattern  {
    process P;
  }
  when {
    P.activities->forAll(a | a.aS == finished and a.tS = ok)
  }
  then {
    print("reached")
 }
```

Fig. 11. Observer rule for model checking.

3.6 Execution and Model Checking

The operational semantics that we provide for DSML's semantics is executable, hence, it can directly be used for execution, and, with some user input, for model checking of reachability properties. Execution is here the nondeterministic execution of the ($\mathbb{K}$ semantics of) KMRL rules using the $\mathbb{K}$ execution engine.

We now illustrate model checking on xSPEM, whose metamodel and operational semantics rules were shown in Section 2.2. We consider the folllowing reachability problem: *is it possible, from the model shown at the bottom of Figure 2, to reach a model where all activities of the model's proper process*[6] *are finished within their expected time limits*? A model where all activities of a process P are finished within their expected time limits is characterised by the following OCL expression: $\texttt{P.activities} \rightarrow \texttt{forAll}(\texttt{a} \mid \texttt{a.aS} = \texttt{finished} \wedge \texttt{a.tS} = \texttt{ok})$. To search for such states, we add the KMRL rule in Figure 11 to the set of rules of the semantics of xSPEM. The rule is not part of the semantics of xSPEM, but acts like an observer, which runs together with the operational semantics rules, and, when it observes that the expected OCL query holds, it prints (by convention) the string "reached" - actually, it adds this string at the end of the $\langle \_ \rangle_{\mathsf{result}}$ cell.

The problem of finding states satisfying OCL Boolean queries has thus been reduced to searching for $\mathbb{K}$ configurations denoting models, reachable from a given initial model-configuration, such that the $\langle \_ \rangle_{\mathsf{result}}$ cell contains a sequence ending with the string "reached". We have automated this process using a script[7], that takes the compiled $\mathbb{K}$ semantics of our DSML enriched with the observer rule, launches it in the Maude rewriting engine to search for the shortest solution, and returns the solution, filtered for better readability. Recall that a compiled $\mathbb{K}$ definition is a Maude rewrite specification. Hence, all what users need to write is their observer rule in KMRL: the rest of the process is fully automatic.

[6] Remember that we have imposed an OCL constraint stating that there is exactly one proper process - i.e., a process which is not an activity - in each xSPEM model.

[7] The script can be tested on-line using the Web interface from the address `https://fmse.info.uaic.ro/tools/`.

Here is, for example, the result of model checking for the property defined by the above observer, as produced by our script (whe have taken advantage of the fact that rules print what they do: starting, clock ticking, and finishing):

```
["start"] [a : activity]
["tick"] [1] ["tick"] [2] ["tick"] [3] ["tick"] [4] ["tick"] [5]
["finishOk"] [a : activity]
["start"] [b : activity]
["tick"] [6] ["tick"] [7] ["tick"] [8]
["finishOk"] [b : activity]
```

One thing that can be noticed is that the expected property is indeed satisfied: both activities are finished in time (cf. the `["finishOk"]` output printed by their "finishing" rules). Another thing that can be noticed is that `b` started only when `a` finished, satisfying the requirement illustrated in the xSPEM model in Figure 2 by the `finishedToStart` reference linking `b` to `a` via `w1`. If we replace this link by a e.g., `finishedToStart`, the shortest solution is a different one, in which the two activitie's starting and finishing events occur in a different order.

Model-checking reachability properties can straightforwardly be generalised to checking the feasability of *scenario executions*, which consists in checking whether certain partially specified sequences of actions are executable by a DSML.

We are currently working on a language for scenario definition, its mapping to KMRL rules, and on the scenario-feasability verification technique in $\mathbb{K}$.

4 Conclusion, Related, and Future Work

We have proposed a formal approach for the definition and analysis of DSMLs. The approach uses the $\mathbb{K}$ semantical framework, which has shown its efficiency at defining semantics of general progamming languages, and applies it to the MDE ingredients used in defining DSMLs: metamodels for syntax, OCL for static semantics, models for "programs", and a combined declarative/imperative model-transformation language for operational semantics of DSMLs, which we call KMRL.

The approach was illustrated on xSPEM, a DSML based on an OMG standard.

Metamodels, OCL, and models are standard MDE concepts, which can be assumed to be familiar to software engineers trained in the basics of MDE. We have designed KMRL to re-use as much as possibly MDE basics: OCL occurs in both conditions and imperative parts, and the declarative part of KMRL generalises the representation of MDE-models. Our hope is therefore that KMRL will be easy to learn by engineers familiar with the basics of MDE, which will enable them to formally define their DSMLs and to perform formal verifications on them.

We have much benefitted in this work from $\mathbb{K}$'s modularity. Each syntactical construct of OCL is defined semantically in terms of a few $\mathbb{K}$ rules, and adding new constructions does not alter the semantics of the existing ones. Once OCL was defined, it could be reused as such to define the semantics of conditions of KMRL rules, and, with a few more $\mathbb{K}$ rules, we defined the imperative part of KMRL. We have also much benefitted from the flexibility of the general Rascal context-free grammars for parsing, and also from Rascal's powerful primitives for navigating in and transforming abstract syntax trees on the fly.

Comparison with Related Work KERMETA [7] is a metamodelling language, which allows users to define the syntax of DSMLs using metamodels, and their operational semantics by means of imperative commands of the language (assignments, loops, ...). Compared to KERMETA, KMRL also has declarative features (model-rewrite rules), it is formally defined, and allows for formal verification.

The ATL language [8] is a mixed declarative/imperative model transformation language. A formal definition of ATL in Maude [9] has been given in [10]. We took inspiration from ATL in this work. Compared to ATL, the declarative features of KMRL are more developped: in ATL one can only match over one model element, whereas in KMRL we allow for matching over arbitrary model patterns. On the other hand, ATL's imperative features are more developped than KMRL's: in ATL rules can call each other, and can invoke methods of class diagrams. Another difference is that ATL can perform general model transformations (i.e., between different metamodels), whereas KMRL is currently limited to one metamodel.

Several other approaches [11–13] use the Maude algebraic and rewriting-based formal specification language [9]. In these approaches, model transformations (in particular, DSML operational semantics) can only be specified in a declarative manner, by mapping them to Maude equations/rewrite rules. Compared to these approaches, ours also includes imperative features, which are lower-level but allow for better control. The same comparison can be drawn with declarative model transformations based on graph rewriting [14, 15].

Finally, the so-called *translational* approach [16] consists in endowing in a DSML with a formal semantics by translating it to a target language that does have a formally defined semantics. For example, xSPEM has been defined by translation to timed Petri nets [16]. Our approach differs in that we define not individual DSMLs, but a DSML *definition framework* (here, the MDE-based one). Our approach is thus more general than the translational one, and is more likely to be accepted by nonexperts since it does not require from them specialised knowledge of a target language (for writing a translation from DSML to it). On the other hand, due to its generality, our approach is likely to be less efficient for execution/verification than specialised, "hard-coded" translational ones.

Future Work One can envisage a way to combine the benefits of the translational approach (efficiency) and of ours (generality). It would consist in first having the DSML specialists formally define their language as we propose; then, the definition can serve as reference for translation to specialised languages for, e.g., more efficient execution and verification. If the target language has a formally defined operational semantics, the translation between the formally defined DSML and the target language can even formally be proved correct if needed.

Regarding formal verification, we are now working on scenario verification, which constitutes a high-level validation technique, compatible with the high-level nature of our DSML definition framework. The framework itself is currently implemented as a loosely coupled set of tools, which requires some knowledge to operate with. We are working on an implementation under Eclipse that will present users a friendly interface for their DSML definitions and analyses.

References

1. The Objet Management Group. The object constraint language, version 2.2. Technical report, 2010. `http://www.omg.org/spec/OCL/2.2/`.
2. Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *SCAM*, pages 168–177. IEEE Computer Society, 2009.
3. G. Roşu and T.-F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
4. Reda Bendraou, Benoît Combemale, Xavier Crégut, and Marie-Pierre Gervais. Definition of an executable spem 2.0. In *APSEC*, pages 390–397. IEEE Computer Society, 2007.
5. Software & systems process engineering metamodel specification (SPEM). `http://www.omg.org/spec/SPEM/2.0/`.
6. Grigore Roşu, Chucky Ellison, and Wolfram Schulte. Matching logic: An alternative to Hoare/Floyd logic. In Michael Johnson and Dusko Pavlovic, editors, *Proceedings of the 13th International Conference on Algebraic Methodology And Software Technology (AMAST '10)*, volume 6486, pages 142–162. LNCS, 2010.
7. Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2005.
8. Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.
9. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
10. Javier Troya and Antonio Vallecillo. Towards a rewriting logic semantics for atl. In Laurence Tratt and Martin Gogolla, editors, *ICMT*, volume 6142 of *Lecture Notes in Computer Science*, pages 230–244. Springer, 2010.
11. Artur Boronat, Reiko Heckel, and José Meseguer. Rewriting logic semantics and verification of model transformations. In Marsha Chechik and Martin Wirsing, editors, *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2009.
12. José Eduardo Rivera, Francisco Durán, and Antonio Vallecillo. Formal specification and analysis of domain specific languages using Maude. *Simulation: Transactions of the Society for Modeling and Simulation International*, 85(11 - 12):778–792, 2009.
13. Vlad Rusu. Embedding domain-specific modelling languages into Maude specifications. *ACM Software Engineering Notes*, 2011. To appear. Extended version available at `http://researchers.lille.inria.fr/~rusu/SoSym/`.
14. Gabriele Taentzer. AGG: A graph transformation environment for modeling and validation of software. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *AGTIVE*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer, 2003.
15. György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. VIATRA - visual automated transformations for formal verification and validation of UML models. In *ASE*, pages 267–270. IEEE Computer Society, 2002.
16. B. Combemale, X. Crégut, P.-L. Garoche, and X. Thirioux. Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification. *Journal of Software*, 4(9):943–958, November 2009.

Verification of Information Flow Properties of JAVA Programs without Approximations

Christoph Scheben and Peter H. Schmitt[⋆]

Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5, 76131 Karlsruhe, Germany

Abstract. In this paper we propose a methodology for the specification and verification of information flow properties for sequential JAVA programs. This proposal also covers declassification. We define an extension of the Java Modeling Language (JML) that significantly goes beyond previous approaches. The JML specification clauses are translated into proof obligations in Dynamic Logic. An experimental implementation within the KeY-system shows the feasibility of the approach.

1 Introduction

This paper is concerned with the specification and verification of information flow properties. A typical example of an information flow property is confidentiality: An attacker observing the public outputs of a computing system cannot gain information on secret data. The frequently quoted survey paper [18] gives a concise introduction into this area, now commonly called *language-based information-flow security*, and reviews a number of approaches that have been persued in the field. We will be concerned here, more precisely, with an approach to information flow security using program logic. In this approach information flow properties are recast as formulas in a general, as opposed to problem specific, language e.g., in the language of Hoare Logic or of a weakest precondition calculus as pioneered in the papers [5] and [1]. It is an appealing feature of this methodology that the formalization is in most cases a straight forward transcription of the informal definition. Another advantage is the possibility to use existing program verification systems and theorem provers to support verification of the specified properties. There are various options on how theorem proving support can be organized. The papers [15,4,20] investigate the use of abstraction while [8] is devoted to the integration of security types systems into program verification. Both lines of attack lead to approximate results. In this paper we follow the lines of [5] and avoid approximative methods.

The novel contributions of the present work are twofold. In the original proposal [5] to use theorem proving in the analysis of secure information flow the

[⋆] This work was supported by the German National Science Foundation (DFG) under the project "Program-level Specification and Deductive Verification of Security Properties" within the priority programme 1496 "Reliably Secure Software Systems - RS3".

proof of concept was demonstrated for a simple theoretical programming language. All consideration in this paper refer to real world sequential Java programs. As a second contribution we extend the Java Modeling Language (JML), see [12], by new constructs that allow to specify information flow properties. The formulation of such properties at the level of the internal program logic is a necessary intermediate stage given the current state of research. In the end one would want a way to specify them that is accessible to the programmer or domain expert without a background in logic. In particular, there should be no mentioning of concepts like self-composition at this level of abstraction. For functional properties the behavioral interface specification language JML has been successfully used for this purpose. In our approach, JML method contracts are extended with three new clauses: `respects`, `parameter_dependencies` and `declassify`. The first clause specifies sets of locations $\mathcal{R}$. The informal semantics of such a specification is, that for every $R \in \mathcal{R}$ the locations not in R do not interfere with locations belonging to R. This means that an attacker which can observe one of those sets of locations won't be able to deduce more information through the execution of the method than he already knew. The second clause is used to specify in which cases the parameters and the return value are observable. Finally, the `declassify`-clause is used to specify permitted additional information flow to some location set by the specification of a term. If the method is executed, an observer of the specified location set may learn the value of the term before the execution. The extension of JML integrates seamlessly with functional JML specifications. This is important since a real precise calculation of information flow dependencies can only be achieved with knowledge on the functional behavior of a program or method. This also works the other way around: knowledge on information flow dependencies does improve functional verification. The annotated Java programs are translated to JavaDL proof-obligations which express non-interference by self-composition. This translation does not over-approximate dependencies between variables. The proof-obligations are then verified with the help of the KeY-System. In combination with a functional specification, which is assumed to exist for the functional verification anyway, the KeY-system was able to verify the formulas in our examples automatically. We expect the approach to work automatically at least in any case in which the KeY-system can proof the functional specification automatically. However, at the time of writing we don't have a bigger case-study which can substantiate this claim. The approach has been implemented in an experimental version of the KeY-System.

Outline. As a starting point, a simple definition of non-interference and its formalisation in Java Dynamic Logic will be considered in the next section. The formalisation will be illustrated on a password checker example which will be used and extended throughout the paper. Section 3 gives a short introduction to JML and JML* and continues with the definition of an extension of JML* suitable for the specification of information flow properties. Section 4 extends the formalisation of the simple form of non-interference to a form which is suitable for the verification of Java programs and describes the translation of the JML* extensions into JavaDL. Finally the approach is discussed in a conclusion in Section 5.

2 Information Flow and Java Dynamic Logic

The most prominent information flow property is *non-interference*. In the simple case non-interference is defined for a program P and a partition of the program variables of P in low security variables *low* and high security variables *high*. The *low*-variables are publicly readable variables whereas the *high*-variables contain secret data which should be protected. Intuitively non-interference expresses that there should be no data-flow from *high*-variables into *low* ones. Instead of talking about *low* and *high* variables it is quite common to talk about *low* and *high* memory or heap locations. This is more realistic since the memory / heap is the place where the data is actually stored. Informally, non-interference can be stated as follows: A program P satisfies *non-interference* for a partition of the heap locations accessed by P in low-security-locations *low* and high-security-locations *high* iff running two instances of P on heaps which agree on the low-locations *low* result in heaps which also agree on *low*. This statement is formalised in Section 2.2 in Definition 1. Executing a program twice in this form is called self-composition [3,5]. Non-interference can be formalised naturally in Java Dynamic Logic, an extension of typed first-order logic, as it will be shown in Section 2.2. As a preparatory step, the next section will give a short introduction to Java Dynamic Logic.

2.1 Basics on Java Dynamic Logic

Dynamic Logic is an extension of typed first-order logic tailored towards reasoning about computer programs, see [9] for an early publication and [10] for a modern account. Typical formulas, that go beyond first-order logic, are of the form $\langle\pi\rangle F$ or $[\pi]F$ where F is again a Dynamic Logic formula and π is a program. In theoretical investigations the programs π are taken from some abstract programming language. In the instantiation of Dynamic Logic that we are concerned with, JavaDL, π can be an arbitrary, executable, sequential Java program. The semantics of Dynamic Logic is based on the notion of a program state, i.e., an assignment of values to all program variables, global and local. The formula $\langle\pi\rangle F$ is true in state s, if the program π started in s terminates and formula F is true in the terminating state. This corresponds to total correctness assertions in Hoare logic that the reader might be more familiar with. Dually $[\pi]F$ is true in state s, if either π does not terminate when started in s or $\langle\pi\rangle F$ is true in s. This corresponds to partial correctness assertions in Hoare logic. The first-order logic part of JavaDL contains types *Heap* and *Field* and thus allows quantifications $\forall$ *Heap h* and $\forall$ *Field f* . Furthermore there is an implicit program variable *heap* of type *Heap* that evaluates in any state to the current heap of the Java program. The values of fields, arrays and information on created objects are stored and accessed by suitable functions as formalized in the theory of abstract arrays, see [16, pages 69 – 70] and [22]. The details of this model play no role in the current paper. The preceeding explanations were included for the reader who might wonder how the values of program variables completely describe the computation state of a Java program. JavaDL uses an additional modal operator $\{v := t\}$ called an *update*, where v is a program variable and t a JavaDL expression. A formula $\{v := t\}F$ is true in state s if F is true in the state s' with $s'(w) = s(w)$ for variables $w \neq v$ and $s'(v)$ equals the value of expression t in state s. Updates serve more than one purpose in JavaDL. They are ultimately necessary for an axiomatization of forward symbolic execution. For the reader of this paper it suffices to think of updates as an interface between logical and program variables. While program variables may occur in formulas, logical variables are not allowed in programs. But logical variables may occur in the expression t in an update $\{v := t\}$. With this knowledge on JavaDL, it should be possible to follow the presentation of the formalisation of non-interference in JavaDL in the next section.

2.2 Formalising Simple Non-Interference in JavaDL

The informal definition of non-interference from above can be formalised as follows. Running two instances of P in parallel can be achieved by conjunction of two single runs of P: $[P]post_1 \wedge [P]post_2$. Furthermore, P needs to be executed on two arbitrary heaps which agree on the low-locations. Executing P on two arbitrary heaps can be expressed as follows:

$$\forall\ Heap\ h^1_{in}, h^2_{in}\ (\{heap := h^1_{in}\}[P]post_1\ \wedge\ \{heap := h^2_{in}\}[P]post_2)$$

Here, the implicit heap variable *heap* is updated to the arbitrary heaps h^1_{in} and h^2_{in}, respectively, before P is executed. The requirement that h^1_{in} and h^2_{in} have to agree on the low-locations will be postponed for the moment. The value of *heap* after the execution of P can be accessed only in the scope of the modality $[P]$. Thus, in order to compare the heaps after the execution of P, the values of *heap* have to be "saved" in logical variables and compared afterwards. This can be achieved by the introduction of two more heap variables, h^1_{out} and h^2_{out}, as follows:

$$\begin{aligned}&\forall\ Heap\ h^1_{in}, h^2_{in}, h^1_{out}, h^2_{out}\ (\\ &\qquad \{heap := h^1_{in}\}[P]h^1_{out} = heap\ \wedge\ \{heap := h^2_{in}\}[P]h^2_{out} = heap\\ &\quad \to (\text{comparison of } h^1_{out} \text{ and } h^2_{out})\qquad\qquad)\end{aligned}$$

Finally, the condition that the heaps after the execution of P have to agree on the low-locations if the heaps before the execution of P agreed on those locations has to be added. Since in JavaDL a location set is represented as a tuple $(o, f) \in java.lang.Object \times Field$, where o is an object and f is a field, non-interference can be defined formally as follows:

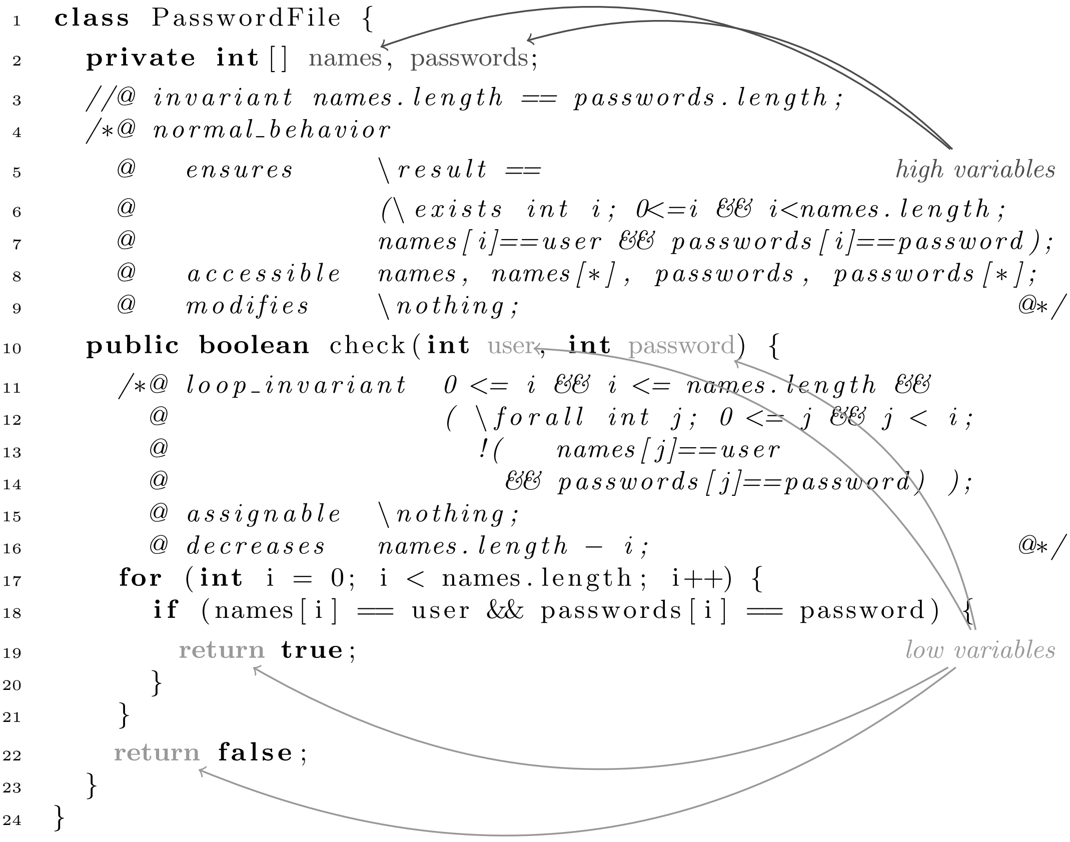
Definition 1 (Simple Non-Interference). *A program P satisfies* non-interference *for a partition of the heap locations accessed by P in low-security-locations low and high-security-locations high* iff

$$\begin{aligned}&\forall\ Heap\ h^1_{in}, h^2_{in}, h^1_{out}, h^2_{out}\ (\\ &\qquad \{heap := h^1_{in}\}[P]h^1_{out} = heap\ \wedge\ \{heap := h^2_{in}\}[P]h^2_{out} = heap\\ &\quad \to \forall\ java.lang.Object\ o\ \forall\ Field\ f\ (\\ &\qquad\qquad (o, f) \in low \wedge\ \{heap := h^1_{in}\}o.f = \{heap := h^2_{in}\}o.f\\ &\qquad\quad \to \{heap := h^1_{out}\}o.f = \{heap := h^2_{out}\}o.f\qquad)\quad)\end{aligned} \tag{1}$$

In principle, Formula (1) is a reformulation of [5, Formula (7)]. However, there are slight differences: our formulation talks about heaps instead of program variables and more essentially the renaming of the heap (the program variables in [5]) is done with the help of an update as part of the formula. Thus, there is no need for a potentially error-prone external renaming. In this formulation we also execute P in parallel instead of sequentially. However, the only impact of this fact at this point is that the formulation looks nicer because we treat both executions equivalently. Finally it should be mentioned that the given formalisation is still quite abstract and can't be used directly as proof obligation for the KeY System. Some details on general assumptions like invariants and the wellformedness of the heaps etc. are abstracted away. Furthermore, it does not cover important features of a practicable non-interference specification language for JAVA, as it can be seen in Section 4. Before we introduce a new program-level specification language for non-interference in Section 3, we want to illustrate the formalisation of simple non-interference with an example.

2.3 Example

The frequently used password checker example will be used to illustrate the formalisation. The example will be extended throughout the paper. The considered implementation (Figure 1) consists of a class `PasswordFile` with two private arrays, `names` and `passwords`, which store the user-names and their corresponding passwords at the same index. Obviously, the length of those two arrays has to coincide. This is formulated with the help of an JML-invariant in line 3. For the moment the reader may assume that such an invariant holds in any state of the program. More details on JML will be given in Section 3. Furthermore, the class contains a method `check` which takes a user-name and a password. It checks whether there exists an index i at which the array `names` contains the user-name and at which the array `passwords` contains the password. If such an index exists, the method returns true, otherwise false. The implementation covers a full functional JML-specification consisting of a method contract and a loop-invariant. Those specifications are not relevant at the moment, but will be discussed in Section 3. Still, a formal specification of *low* and *high* variables is missing, since the current version of JML does not allow for such specifications. Let's assume the arrays `names` and `passwords` and their contents are considered as *high* variables whereas the parameters `user` and `password` as well as the not explicitly named `return`-variable are considered as *low* variables. This is reasonable since user-names and passwords normally should be kept secret whereas the caller knows the user-name and password he passed as well as the returned value. Even though the parameters and the return variable are no heap-locations, in a modular context they have to be treated similar to heap locations as discussed in Section 3.2. The method *check* translates in connection with these informal *low*- and *high*-specifications to the JAVADL formula of Figure 2. The formula is assembled as follows: the first part contains some general assumptions, which have been abstracted away in Formula (1) and are not central here either. The following two parts contain the symbolic execution and comparison as introduced



```
class PasswordFile {
  private int[] names, passwords;
  //@ invariant names.length == passwords.length;
  /*@ normal_behavior
    @   ensures     \result ==
    @               (\exists int i; 0<=i && i<names.length;
    @               names[i]==user && passwords[i]==password);
    @   accessible  names, names[*], passwords, passwords[*];
    @   modifies    \nothing;                                @*/
  public boolean check(int user, int password) {
    /*@ loop_invariant  0 <= i && i <= names.length &&
      @                 ( \forall int j; 0 <= j && j < i;
      @                    !(   names[j]==user
      @                      && passwords[j]==password) );
      @ assignable  \nothing;
      @ decreases   names.length - i;                        @*/
    for (int i = 0; i < names.length; i++) {
      if (names[i] == user && passwords[i] == password) {
        return true;
      }
    }
    return false;
  }
}
```

Fig. 1. Example of a password checker in Java with a full functional JML-specification and an informal annotation for *low*- and *high*-variables.

in Section 2.2. Still, there is a slide difference: as mentioned before, the parameters and the return value of the method have to be considered similar to heap locations. Therefore they also occur in the comparison. In this particular example only the return values have to be compared, because the heap does not contain low variables and the values of the parameters are not observable after the return of the method. Next, it will be shown how Java programs can be annotated systematically with non-interference specifications.

3 Program-Level Specifications

The last section showed how, in principle, non-interference can be formalised in JavaDL and how proof obligations can be generated manually out of Java programs with an informal annotation of *low* and *high* locations. This section will discuss how Java programs can be annotated with non-interference specifications which seamlessly integrate with functional specifications in JML* and which

```
∀ Heap h_in_1, h_in_2, h_out_1, h_out_2  // independend heaps
∀ PasswordFile self                      // considered class
∀ int user1, password1, user2, password2 // method arguments
∀ boolean result1, result2               // return values
// General Assumtions + Class Invariants
    wellFormed(h_in_1) ∧ wellFormed(h_in_2) ∧ ...
// Independent Symbolic Executions
  ∧ {heap := h_in_1}\[{ ...
       boolean r = self.check(user1 ,password1)@PasswordFile;
     ...}\]( h_out_1 = heap ∧ result1 = r )
  ∧ {heap := h_in_2}\[{ ...
       boolean r = self.check(user2 ,password2)@PasswordFile;
     ...}\]( h_out_2 = heap ∧ result2 = r )
// Comparision of the low variables
  ∧ user1 = user2 ∧ password1 = password2
  → result1 = result2
```

Fig. 2. Formalisation of non-interference in JavaDL for the example of Figure 1. The three dots "..." mark passages where some less important JavaDL details have been abstracted away.

are suitable for automatic translation into JavaDL. The specification entities are in particular suitable for the (implicit) specification of security lattices and intentional information leakage. Before the new specification entities are introduced in Section 3.2, the next section will present some basics on JML and its dialect JML*. Section 3.3 illustrates the entities on the example of Figure 1 thereafter.

3.1 JML and JML*

The Java Modeling Language (JML) is a popular specification language for the behavior of Java code [12]. It can be used as a design by contract (DBC) style specification language. Java expressions enriched with other specification constructs like quantifiers are used to write predicates which in turn are used in assertions, such as pre- and postconditions and invariants. [22] introduced a dialect of JML, called JML*, which is suitable for modular specifications. Our approach will be based on this dialect. In the following, the specification entities which are most important in the context of this work will be explained shortly by the example of Figure 1. These entities are method contracts, invariants and model fields.

The method contract of Figure 1 starts with the key-word `normal_behavior`. This means that the method won't throw an exception if the precondition of the method is fulfilled. Pre-conditions are specified via the key-word `requires`. If the key-word is missing—as it is the case in our example—the pre-condition is implicitly defined as `true`. Thus, the specification guaranties that no exception will be thrown. Post-conditions are specified via the key-word `ensures`. In Figure 1, lines 6 to 8 specify the post-condition of the method `check`, which says that the result of the method is `true` iff there exists an index i at which the array `names` contains the passed user-name and at which the array `passwords` contains the passed password. Furthermore, the key-word `modifies` defines a set of heap locations whose values may be changed at most by the execution of the method. In our case, no locations may be changed. Similarly, the key-word `accessible` defines a set of heap locations whose values may be read at most by the execution of the method. Here, the specification of `check` expresses that at most the heap locations of the fields `names` and `passwords` as well as the entries of the two arrays are read. Line 3 shows an invariant specification. It says that the length of the arrays `names` and `passwords` coincide. In this work it can be assumed that invariants hold in every state of the program execution. The real semantics is more complicated, but it is not essential to understand the semantics in detail in order to follow the presentation of this work. Finally, model fields have to be considered. The following example shows a model field definition:

```
1  /*@ model \locset pwdFileManager;
2    @ accessible pwdFileManager: footprint;
3    @ represents pwdFileManager =
4    @        names, names[*], passwords, passwords[*];
5    @*/
```

The first line declares the model field itself. Lines three and four define the set of locations, to which the model field evaluates to. In the context of this work we consider only model fields of type `\locset`. Model fields can be defined either via an =-symbol followed by an expression of type `\locset` (similar to the specification of `modifies`- and `accessible`-clauses) or a `\such_that` followed by a Boolean expression. In the latter case the model field evaluates to some location set which fulfills the Boolean expression. Finally, the second line states that the evaluation of the model field—which can be thought of as a pure function here—depends only on the *footprint* of the object. The footprint of an object contains all locations which might be accessed by the object. All locations not belonging to the footprint of the object can't be read or changed by the object.

Next, the new JML* specification entities are introduced.

3.2 Extending JML* for Non-Interference Specifications

Given the simple form of non-interference from Section 2, one of the first questions one may ask is: how can *low* and *high* variables be defined in JML*? We won't answer this question directly because it turns out that for practicable information flow analysis a classification into *low* and *high* variables normally is too coarse. Most existing analysis tools use lattices of security levels instead (see for instance [13]). In a security lattice information may flow only from lower levels to higher ones. The power set of heap locations $\mathcal{P}(loc_{heap})$ forms in combination with the set union and intersection in some sense the most general security lattice [11,5]: any other type lattice is subsumed by it. Therefore it is reasonable and quite common to restrict oneself to (sublattices of) $\mathcal{P}(loc_{heap})$. Given a security lattice, a security policy is defined as a mapping of the set of heap locations to the set of security levels of the lattice.

Our approach defines security policies in a decentralised way. The intuition is that every user of a system has one or more views on the system. A view is a particular set of heap locations which can be observed by the user. A system is safe, if no view R can be used (by a user) to deduce more information about a system state through the execution of a method than the information already visible to R. In order to define a security policy for a method m, m is annotated with a set of views $\mathcal{R}$. $\mathcal{R}$ implies the following security policy. Information may flow from a heap location y to another heap location x iff y is contained in every view in which x is also present. This means that information may flow from y to x iff for all location sets $R \in \mathcal{R}$ the condition $x \in R \Rightarrow y \in R$ holds. Every heap location x can be assigned a security level $dep(x)$ as follows: $dep(x)$ is the set of all heap locations on which x is allowed to depend on. Formally the security level for x is defined as $dep(x) := \{y \mid \forall R \in \mathcal{R} : x \in R \Rightarrow y \in R\}$. The set of security levels for all heap locations forms in combination with the set-union and intersection the implicit security lattice: information may flow from a location y to a location x if the dependencies $dep(y)$ of y are a subset of the dependencies $dep(x)$ of x. The mapping dep is therefore the security policy defined by $\mathcal{R}$.

In our extension of JML* a security policy is specified implicitly with the help of the `respects`-clause as a set of location sets:

```
/*@ respects      \set_union(names, names[*]),
  @               \set_union(passwords, passwords[*]);
  @*/
boolean check(int user, int password) { ...
```

The `respects`-clause is a usual clause in a method contract. It takes a list of expressions of type `\locset` as parameter. A contract may contain multiple `respects`-clauses. In the latter case the clauses are treated as one big `respects`-clause consisting of the concatenation of all listed location sets. As sketched before, the clause states that in case the precondition of the contract is valid, for each of the listed location sets R the values of the locations of R after the execution of the method depend only on the values of the locations of R before the execution. It is notable that in JML* location sets are specified with the help of `\locset`-expressions. Such an expression is evaluated to a location set in some heap. Thus a `\locset`-expression might evaluate to different heap locations and hence implicitly define different security levels in different heaps. This is intentional. Consider a linked list: if the whole list is considered to be visible to some view R and a new element is added to the list, then also the new element of the list should be visible to R. But this implies that the location set belonging to R needs to change. Formally, views can be defined as follows:

Definition 2 (View). *A view R is an JML*-expression of type* `\locset`.

In the context of JavaDL, a view is defined similarly as a term of type *LocSet*. Sets of views can, but don't have to be specified explicitly: an alternative to list a set of views $\mathcal{R}$ explicitly in a `respects`-clause is to use an underspecified model field in order to represent $\mathcal{R}$. In this way the specification and verification effort can be reduced in many cases to the specification and verification of one single model field:

```
/*@ model \locset anyUser;
  @ accessible anyUser:
  @   footprint;
  @ represents anyUser
  @   \such_that \subset(
  @       anyUser, footprint);
  @*/

//@ respects anyUser;
boolean check( ... ) { ...
```

The above example illustrates such an underspecification. The model field `anyUser` can represent any location set which can be referred to by the object. Since all locations which are not accessed by the object can't interfere with the object, the specification ultimately says that the method fulfills non-interference for any security level and therefore that no information flows on the heap.

Till now it has been specified which information is allowed to flow between heap locations. Beside those specifications, for methods it has to be specified on which locations a value passed through its parameters may depend on at most. This can be seen with the help of the following simple example:

```
public int low;
private int high;
//@ respects low;
int m(int param) {
  low = param; return param;
}
```

Imagine `low` and `high` are low- and high-locations, respectively, and the method `m` is called with `high` as parameter. Then the high-value will be assigned to `low` and therefore the call would be unsafe. On the other hand, if `m` is called with `low` as parameter everything is fine. For a similar reason one has to specify a guaranty on which locations the return value of the method can depend on at most. In JML* allowed parameter dependencies are specified with the help of the `parameter_dependencies`-clause:

```
//@ parameter_dependencies \allLocs, \allLocs : \nothing;
boolean check(int user, int password) { ...
```

The clause assigns to every parameter as well as to the return value a view. The views for the parameters are listed in the order the parameters are declared, separated by a comma. The view for the return value follows the location set for the last parameter, separated by a colon. A method contract may contain several `parameter_dependencies`-clauses. In this case non-interference holds for every clause. The `parameter_dependencies`-clauses of the example states that the values passed by `user` and `password` may depend on any heap location and that the return value is guaranteed to depend on no heap location. The specification of the dependencies of the return value is very strict in this example: it permits only that the method returns a constant. Section 3.3 will show a more interesting specification. From some point of view, the `parameter_dependencies`-clause specifies the views which "are allowed to call the method", whereas the `respects`-clause specifies which views can't deduce information from such a call.

The parameter_dependencies-clause can be considered also from another point of view: it can be seen as a specification which defines for each set R in the respects-clause whether the parameters and the return-value are considered as *low* or *high* with respect to R: if the location set specified for a parameter *par* is a subset of R, then *par* is considered as *low*, else as *high*.

The respects-clause in combination with the parameter_ dependencies-clause are sufficient to specify non-interference for methods in JML*. However, as it is well known (see for instance [13]), non-interference on its own is too restrictive in order to check a lot of useful JAVA programs for unintentional information leaks. This includes the program of Example 1. KeY won't be able to show the universal validity of the formula of Figure 2 because the method check indeed leaks some information about the secret arrays names and passwords: the information whether the passed user-name and password are contained in names and passwords or not. In order to distinguish intentional information leaks from unintentional ones, intentional leaks have to be specified clearly. Those specifications are called declassifications [19]. In our case, declassifications are declared as part of the method contract and therefore are clearly separated form the implementation. The following example illustrates the usage of the declassify-clause.

```
/*@ declassify
  @   ( \exists int i; 0 <= i && i < names.length;
  @       names[i] == user && passwords[i] == password )
  @   \from pwdFileManager  \to checkUser  \if true;
  @*/
boolean check(int user, int password) { ...
```

The clause specifies the information to be declassified in form of a term. The value to which this term evaluates to in the prestate of the execution of the method is the information which is allowed to leak. The leakage is restricted to the locations specified in the \to-part of the clause. Furthermore, the evaluation of the term may depend at most on the locations specified in the \from-part. Finally, the declassification takes place only in case the formula in the \if-part is valid. The \from-, \to- and \if-parts are optional. Since the information can be leaked only in the body of the method, our declassification entity defines where, exactly what information may be leaked by whom and to whom. It fulfills furthermore the sanity check from [19] by the preservation of semantic consistency, conservativity, monotonicity of release of information and non-occlusion.

The next section illustrates the JML* extensions on the password checker example of Figure 1.

3.3 Example

Figure 3 gives a complete example of the specification of the check method from Figure 1. The specification of Figure 1 is extended in two parts. The first part, lines 1 to 7, declares two underspecified model fields, anyUser and checkUser, as in Section 3.2. They can stand for any view accessible by an object of type

```
/*@ model \locset checkUser, anyUser;
  @ accessible anyUser: footprint;
  @ represents anyUser \such_that \subset(anyUser, footprint);
  @
  @ normal_behavior
  @   ensures       \result ==
  @                 ( \exists int i; 0 <= i && i < names.length;
  @                   names[i]==user && passwords[i]==password);
  @   accessible  names, names[*], passwords, passwords[*];
  @   modifies    \nothing;
  @   respects    anyUser;
  @   parameter_dependencies   checkUser,checkUser:checkUser;
  @   declassify  ( \exists int i; 0 <= i && i < names.length;
  @                   names[i]==user && passwords[i]==password
  @               ) \to checkUser;                            @*/
public boolean check(int user, int password) { ...
```

Fig. 3. Complete non-interference specification for the method `check` from Figure 1 with a seamless integration to the functional specifications.

`PasswordFile`. The second part, lines 9 to 20, extends the method contract of the method `check` by a non-interference specification. The `respects`-clause states in combination with the `parameter_dependencies`-clause that in case the method is called with view `checkUser` (that is, the parameters depend at most on `checkUser`) then the view `anyUser` can't learn anything through the execution of `check`. Since `anyUser` and `checkUser` can stand for any view, the considered part of the specification states that no matter with which view the method is called, no view can learn anything by the execution of the method. The `parameter_dependencies`-clause states furthermore that the return value of the method may depend on `checkUser`, which means that the return value may depend on the parameters. Finally, the `declassify`-clause states that the information whether the passed user-name and password are contained in `names` and `passwords` or not may be learned by `checkUser`. Overall, the specification says that no matter with which view `check` is called, no information is leaked to any view except the necessary leakage to the return value. In the next section it will be shown how the introduced information flow extensions of JML* can be translated to JavaDL and checked by the KeY-System. Section 5 will discuss the overall approach in comparison with other information flow verification systems thereafter.

4 Translating JML* Non-Interference Specifications to JavaDL

Before the translation of the introduced JML* information flow entities can be considered, the translation and usage of views has to be discussed. As defined in

Section 3.2, views are JML*-expressions of type `\locset`. Those expressions are translated canonically to JavaDL terms of type *LocSet*. Terms of type *LocSet* are evaluated to a set of heap locations in a given heap. Heap locations in turn are tuples $(o, f) \in \mathit{java.lang.Object} \times \mathit{Field}$, where o is an object and f is a field. As mentioned in Section 3.2, it has to be shown for every view $R \in \mathcal{R}$ specified in the `respects`-clause that the locations of R do not interfere with locations not in R by the execution of a program P. Since R might evaluate after the execution of P to another set of locations than before the execution, the definition of non-interference has to be adopted. Informally, a program P satisfies *non-interference* for a view R iff running two instances of P on heaps which agree on the evaluation L_{pre} of R and on the values of the elements of L_{pre} result in heaps which also agree on the evaluation L_{post} of R and on the values of the elements of L_{post}. The condition, that R has to evaluate to same set of locations in the pre-heaps and post-heaps of the two executions is based on the assumption that an observer of a set of locations is able to determine which locations he is observing. Formally, this form of non-interference can be defined by a generalisation of Formula 1 as follows:

Definition 3 (Non-interference for Views). *A program P satisfies* non-interference *for a set of views $\mathcal{R}$* iff

$$\begin{aligned}&\forall\, \mathit{Heap}\ h^1_{in}, h^2_{in}, h^1_{out}, h^2_{out}\ (\\ &\qquad \{\mathit{heap} := h^1_{in}\}[P]h^1_{out} = \mathit{heap}\ \wedge\ \{\mathit{heap} := h^2_{in}\}[P]h^2_{out} = \mathit{heap}\\ &\quad \rightarrow \bigwedge_{R\in\mathcal{R}} (h^1_{in} \sim_R h^2_{in} \rightarrow\ h^1_{out} \sim_R h^2_{out}) \qquad\qquad)\end{aligned} \tag{2}$$

where the relation $h^1 \sim_R h^2$ is defined as

$$\begin{aligned}&\{\mathit{heap} := h^1\}R = \{\mathit{heap} := h^2\}R\ \wedge\ \forall\, \mathit{java.lang.Object}\ o\ \forall\, \mathit{Field}\ f\ (\\ &\qquad (o, f) \in \{\mathit{heap} := h^1\}R \rightarrow\ \{\mathit{heap} := h^1\}o.f = \{\mathit{heap} := h^2\}o.f \qquad)\end{aligned} \tag{3}$$

It is notable that with the help of this formalisation a whole set of views can be checked with the help of only two symbolic executions.

As next step, the call of the program P in the formalisation will be replaced by a call to a method m of class C with parameters $p_1, \ldots, p_n$ of type $\mathcal{T}_1, \ldots, \mathcal{T}_n$ and a return value of type $\mathcal{T}_r$. In order to reflect the parameter dependencies from the `parameter_dependencies`-clause, a set $\mathcal{R}^{par}$ of parameter dependency lists $R^{par} \in \mathcal{R}^{par}$ is introduced, where each R^{par} contains for each parameter p_i a view specification R^{par}_i. This leads to the following update of Formula 2 which is explained below:

$$\begin{aligned}&\forall\, \mathit{Heap}\ h^1_{in}, h^2_{in}, h^1_{out}, h^2_{out}\\ &\forall\, C\ \mathit{self}\ \forall\, \mathcal{T}_1\ p^1_1\ \ldots\ \forall\, \mathcal{T}_n\ p^1_n\ \forall\, \mathcal{T}_r\ r^1\ \forall\, \mathcal{T}_1\ p^2_1\ \ldots\ \forall\, \mathcal{T}_n\ p^2_n\ \forall\, \mathcal{T}_r\ r^2\ (\\ &\qquad \mathit{self} \neq \mathit{null}\\ &\quad \wedge \{\mathit{heap} := h^1_{in}\}[\mathcal{T}_r\ \mathit{result} = \mathit{self}.m(p^1_1, \ldots, p^1_n)](h^1_{out} = \mathit{heap} \wedge r^1 = \mathit{result})\\ &\quad \wedge \{\mathit{heap} := h^2_{in}\}[\mathcal{T}_r\ \mathit{result} = \mathit{self}.m(p^2_1, \ldots, p^2_n)](h^2_{out} = \mathit{heap} \wedge r^2 = \mathit{result})\\ &\ \rightarrow \bigwedge_{(R,R^{par})\in\mathcal{R}\times\mathcal{R}^{par}} (h^1_{in} \sim^{in}_{R,R^{par}} h^2_{in} \rightarrow\ h^1_{out} \sim^{out}_{R,R^{par}} h^2_{out}) \qquad)\end{aligned} \tag{4}$$

A method is always called on an object. Therefore, Formula (4) quantifies over all objects $self$ of type C, on which m can be called. Since m can't be called on $null$, it is assumed that $self$ is not $null$. Furthermore, since the result of m can be observed from outside the method, the return-values have to be stored in logical variables r^1 and r^2 such that their values can be compared in the relations $\sim^{in}_{R,R^{par}}$ and $\sim^{out}_{R,R^{par}}$ similar to heap locations. Finally, the relations $\sim^{in}_{R,R^{par}}$ and $\sim^{out}_{R,R^{par}}$ between the heaps have to be distinguished as follows. As discussed in Section 3.2, the i-th parameter is considered as a low-location if its dependency set R^{par}_i is a subset of R. If p^1_i and p^2_i label the two potentially different values of the i-th parameter in the two symbolic executions, this condition can be expressed by the following formula:

$$\bigwedge_{i\in\{1..n\}} (\{heap := h^1_{in}\}(R^{par}_i \subseteq R) \rightarrow p^1_i = p^2_i) \tag{5}$$

Thus, $\sim^{in}_{R,R^{par}}$ is defined as the conjunction of the formulas (3) and (5). Formula (5) needs not to be part of $\sim^{out}_{R,R^{par}}$ since the values of the parameters can't be observed after the return of a method any more. On the other hand, $\sim^{out}_{R,R^{par}}$ has to contain a similar condition for the return value:

$$\{heap := h^1_{in}\}(R^{par}_{result} \subseteq R) \rightarrow r^1 = r^2 \tag{6}$$

Hence, $\sim^{out}_{R,R^{par}}$ is defined as the conjunction of the formulas (3) and (6).

As final step, declassifications have to be formalised. During the formalisation of the `respects`-clause and `parameter_dependencies`-clause, an input-relation $\sim^{in}_{R,R^{par}}$ and an output-relation $\sim^{out}_{R,R^{par}}$ have been introduced through the backdoor as abbreviations. That $\sim^{in}_{R,R^{par}}$ is a relation between heaps becomes important now: declassifications can be used to define arbitrary input-relations between heaps. Such an input-relation describes which heaps can be distinguished by an observer of a view through the execution of m. [19] discusses the meaning of this view on declassification in detail. The value of a declassified term D_{term} evaluated in the pre-state of the execution of m may be learned by a view R, if the declassification $(D_{if}, D_{to}, D_{from}, D_{term}) \in \mathcal{D}$ fulfills the following conditions: (1) The condition of the `\if`-part D_{if} is fulfilled in the pre-state of the execution. (2) The `\to`-part D_{to} is a subset of R. (3) The declassified term D_{term} depends only on the locations in the `\from`-part D_{from}. These conditions can be formalised for a set of declassifications $\mathcal{D}$ as follows:

$$\bigwedge_{D\in\mathcal{D}} \left(\begin{array}{l} \quad \{heap := h^1_{in}\}D_{if} \wedge \ \{heap := h^2_{in}\}D_{if} \\ \wedge \{heap := h^1_{in}\}(D_{to} \subseteq R) \\ \wedge\ D_{term} = \{heap := anon(heap, D_{from}, anonHeap\}D_{term} \\ \rightarrow \{heap := h^1_{in}\}D_{term} = \{heap := h^2_{in}\}D_{term} \end{array} \right) \tag{7}$$

Altogether, $\sim^{in}_{R,R^{par}}$ consists of the conjunction of the formulas (3), (5) and (7) whereas $\sim^{out}_{R,R^{par}}$ is the conjunction of the formulas (3) and (6). The equivalence relation between the input-heaps is defined by the equations following the the implication in the formulas (3), (5) and (7).

Figure 4 shows the result of the translation of the JML* specification from Figure 3 to JavaDL. The formula is mainly an instantiation of the formulas (3),

```
∀ Heap h_in_1, h_in_2, h_out_1, h_out_2  // independend heaps
∀ PasswordFile self                        // considered class
∀ int user1, password1, user2, password2 // method arguments
∀ boolean result1, result2                 // return values
// General Assumtions + Class Invariants
    self ≠ null ∧ wellFormed(h_in_1) ∧ ...
// Symbolic Execution
  ∧ {heap := h_in_1} \[{ ...
      boolean r = self.check(user_1,password_1)@PasswordFile;
    ... }\]( result1 = r ∧ h_out_1 = heap )
  ∧ {heap := h_in_2}\[{ ...
      boolean r = self.check(user_2,password_2)@PasswordFile;
    ... }\]( result2 = r ∧ h_out_2 = heap )
// Input−Relation
  ∧ equalsAtLocs(h_in_1, h_in_2, self.anyUser)
  ∧ (  {heap := h_in_1}(self.checkUser ⊆ self.anyUser)
     → user1 = user2 )
  ∧ (  {heap := h_in_1}(self.checkUser ⊆ self.anyUser)
     → password1 = password2 )
  ∧ (  {heap := h_in_1}(self.checkUser ⊆ self.anyUser)
     → (  {heap := h_in_1} ∃ int i0;
             (  0 ≤ i0 ∧ i0 < self.names.length ∧ ...
               ∧ self.names[i0] = user1
               ∧ self.passwords[i0] = password1)
         ↔ {heap := h_in_2} ∃ int i1;
             (  0 ≤ i1 ∧ i1 < self.names.length ∧ ...
               ∧ self.names[i1] = user1
               ∧ self.passwords[i1] = password1 ) ) )
// Output−Relation
  → (  equalsAtLocs(h_out_1, h_out_2, self.anyUser)
      ∧ (  {heap := h_in_1}(self.checkUser ⊆ self.anyUser)
         → result1 = result2 ) )
```

Fig. 4. Translation of the JML* non-interference specification from Figure 3 to JavaDL. The three dots "..." mark passages where some less important JavaDL details have been abstracted away.

(4), (5), (6) and (7). Condition (3) is hidden in the predicate `equalsAtLocs`. The next section discusses the presented approach as well as future work.

5 Conclusion

Discussion. On a first glance, the approaches [21,7,6] seem to be the closest to our approach. The former two, [21] and [7], try to specify non-interference in JML by encoding sufficient conditions for non-interference into pre- and post-conditions. This is less expressive than our approach since the encoded conditions are only sufficient. Furthermore, they do not give hints how to encode declassifications, which is an important feature for the specification of real world programs. [6] on the other hand introduces new JML-keywords which directly define relations between the program variables of two self-composed executions. In particular two key-words to distinguish the variables of the two runs are defined. The approach does not use security policies or a security lattice. Like in JIF, our JML*-extension concentrates more on the actors of the system by the implicit definition of security policies with the help of sets of views.

The JIF system [13] is another important approach on the specification and verification of information flow properties of Java programs. The core idea of JIF is to annotate variables with security policies and derive a security type system from those annotations. Given the security type system, Java programs are analysed with the help of type checking for undesirable information flow. This approach is very efficient and has been applied to several case studies. However, in order to be efficient, security type systems use in some cases quite rough over-approximations of the dependencies between variables. Our approach is much less efficient but has the advantage that no approximations are involved. Another advantage of our approach is that our JML*-extension integrates functional and information flow specifications which makes it easier to use synergy effects of the two specifications. Furthermore, though we like the ideas of the decentralised label model (DLM) which is used in JIF for the labeling of variables, this approach is unfortunately not modular: if a new principal is introduced to a program, variables in the whole program might have to be annotated. Inspired by the DLM, we also define the overall security policy in a decentralised way. However, we do it other way around: instead of annotating variables with labels we annotate views (which are our counterpart of the labels used in JIF) with heap locations. In this way it is possible to write modular specifications. Finally, the declassification-construct in JIF which is used for delimited information release [17] can only be used within the implementation of a method. This transgresses against the rule of clear separation of specification and implementation. In contrast, our declassifications are part of the method-contract and therefore clearly separated from the implementation.

The theoretical paper [2] proposes a scheme for the specification of expressive declassification policies. The authors suggest to use program verification techniques similar to ours to show that a program complies to those declassification policies. These techniques are combined with security type systems which are used to show the compliance of the program to the baseline security policy. Though we hadn't been aware of those ideas during the development of our JML*-extension, our declassification construct complies to the scheme proposed in [2]. Still, we define the additional `\from` part which has no counterpart in [2]. Additionally, our approach is based on sequential Java (so on a real object oriented programming language) whereas [2] defines its own non object oriented programming language—though some ideas are provided how the combination of program verification and security type systems could be extended to object oriented languages.

[14] formalises information flow properties in a higher-order logic and uses Coq for the verification of those properties. This approach seems to be extremely expressive, but comes with the price of more and more complex interactions with the proof system. Further approaches use abstraction and ghost code for explicit tracking of dependencies [15,4,20]. Even though those approaches are more precise than security type systems, they still over-approximate the dependencies between variables / heap locations. Furthermore, they haven't tackled the problem of declassification yet.

Overall, one point which sets our approach apart form all existing ones is that our specification approach can be used to write fully modular specifications. Those specifications fit into the approach of dynamic frames [22] and therefore comply to the principle of information hiding as it is used in object oriented programming languages.

Future Work. Though it is possible to write fully modular specifications with the JML*-extension, we can't claim the same for our verification technique yet. It still needs to be investigated in detail how a contract of a method $m1$ can be used during the verification of a method $m2$ which calls $m1$. However, we have very promising ideas how this can be achieved. Additionally, it might be useful to define views as a set of terms instead of a set of locations. This would allow an even finer grained analysis, where attackers are modeled similar to the PER model [19]. The implementation of this extension should be straightforward. Finally, it should be possible to do a quantitative analysis on the non-interference specifications. Such an analysis would provide a guaranty how much information will be leaked by the method at most. The quantitative analysis could be preformed on the relations between heaps defined by the non-interference specification.

References

1. T. Amtoft and A. Banerjee. Information flow analysis in logical form. In R. Giacobazzi, editor, *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, volume 3148 of *Lecture Notes in Computer Science*, pages 100–115. Springer, 2004.
2. A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. *Security and Privacy, IEEE Symposium on*, 0:339–353, 2008.
3. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. *Computer Security Foundations Workshop, IEEE*, 0:100, 2004.
4. R. Bubel, R. Hähnle, and B. Weiß. Abstract interpretation of symbolic execution with explicit state updates. In F. S. de Boer, M. M. Bonsangue, and E. Madelain, editors, *FMCO*, volume 5751 of *Lecture Notes in Computer Science*, pages 247–277. Springer, 2008.
5. Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Security in Pervasive Computing*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer Berlin / Heidelberg, 2005. 10.1007/978-3-540-32004-3_20.
6. G. Dufay, A. Felty, and S. Matwin. Privacy-sensitive information flow with JML. In R. Nieuwenhuis, editor, *Automated Deduction – CADE-20*, volume 3632 of *Lecture*

Notes in Computer Science, pages 738–738. Springer Berlin / Heidelberg, 2005. 10.1007/11532231_9.

7. C. Haack, E. Poll, and A. Schubert. Explicit information flow properties in JML. In *3rd Benelux Workshop on Information and System Security (WISSec)*, November 2008.
8. R. Hähnle, J. Pan, P. Rümmer, and D. Walter. Integration of a security type system into a program logic. In U. Montanari, D. Sanella, and R. Bruni, editors, *Proc. Trustworthy Global Computing, Lucca, Italy*, volume 4661 of *LNCS*, pages 116–131. Springer-Verlag, 2007.
9. D. Harel. *First-Order Dynamic Logic*, volume 68 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
10. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. The MIT Press, 2000.
11. S. Hunt and D. Sands. On flow-sensitive security types. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 79–90, New York, NY, USA, 2006. ACM.
12. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31:1–38, May 2006.
13. A. C. Myers. Jflow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.
14. A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies with dependent types. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 165 –179, may 2011.
15. J. Pan. A theorem proving approach to analysis of secure information flow using data abstraction. Master's thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2005.
16. S. Ranise and C. Tinelli. The SMT-LIB standard, version 1.2. Technical report, University of Iowa, 2006.
17. A. Sabelfeld and A. Myers. A model for delimited information release. In K. Futatsugi, F. Mizoguchi, and N. Yonezaki, editors, *Software Security - Theories and Systems*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-37621-7_9.
18. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
19. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17:517–548, October 2009.
20. B. van Delft. Abstraction, objects and information flow analysis. Master's thesis, Institute for Computing and Information Science, Radboud University Nijmegen, 2011.
21. M. Warnier. *Language Based Security for Java and JML*. PhD thesis, Radboud University, Nijmegen, The Netherlands, 2006.
22. B. Weiß. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction.* PhD thesis, Karlsruhe Institute of Technology, 2011.

Local Rely-Guarantee Conditions for Linearizability and Lock-Freedom

Bogdan Tofan, Gerhard Schellhorn, and Wolfgang Reif

Institute for Software and Systems Engineering
University of Augsburg
{tofan,schellhorn,reif}@informatik.uni-augsburg.de

Abstract. Rely-guarantee reasoning specifications typically consider all components of a concurrent system. For the important case where components operate on a shared data object, we derive a local instance of rely-guarantee reasoning, which permits specifications to examine a single pair of representative components only. Based on this instance, we define local proof obligations for linearizability and lock-freedom, which we then apply to a non-blocking concurrent stack with explicit memory reuse. Both the derivation of this local instance and its application are mechanized in the KIV interactive theorem prover.

Keywords: Verification, Temporal Logic, Rely-Guarantee, ABA-problem, Linearizability, Lock-Freedom

1 Introduction

The rely-guarantee method [1] deals with the challenges that arise when reasoning about concurrent systems with shared resources. It provides a compositional treatment of interference between system components, i.e., to analyze properties of the overall system, each component can be examined separately based on its specification of expected environment behavior.

Rely-guarantee reasoning proves to be a valuable technique to verify non-blocking (here lock-free) implementations of concurrent data structures, such as stacks, queues or sets. Such algorithms play an important role in multi-core systems and are also contained in concurrency packages of modern, high-level object-oriented programming languages (e.g., java.util.concurrent). They try to better utilize the capacity of multi-cores by avoiding locking and thus increasing the potential of operations to execute in parallel. Their main correctness property *linearizability* [2] ensures that each interleaved execution of concrete data structure operations corresponds to an abstract, sequential execution that preserves the (real-time) order of non-interleaved concrete calls. Their main progress property *lock-freedom* [3] guarantees that in each interleaved execution, always eventually one of the running operations terminates. Lock-free implementations avoid deadlocks, livelocks, convoying and priority inversion. Thus, they are also heavily used in real-time systems (e.g., for real-time garbage collection).

The verification framework is based on interval temporal logic [4] and symbolic execution [5] and is implemented in the interactive theorem prover KIV [6]. It permits to verify the soundness of a typical form of rely-guarantee reasoning, as well as to (mechanically) derive more specific instances of it and to prove decomposition theorems for system-wide correctness or progress properties, such as linearizability or lock-freedom. This paper describes such an instance and illustrates its use.

Our earlier embedding of rely-guarantee reasoning in interval temporal logic described in [7] follows the global approach of [1]: specifications consider the overall program state, consisting of the local states of *all* components and the shared state. Here we reduce this former embedding to allow for specifying properties of concurrent systems – that are unbound in their number of components – in terms of two representative components. This valuable reduction leads to both simpler specifications and proofs, e.g., in the frequent case of verifying concurrent data structures where all processes have similar behaviors. We define local proof obligations for linearizability and lock-freedom based on this instance and show its application by verifying the major safety and liveness aspects of a lock-free stack that recycles memory from a shared pool of reusable memory locations.

To the best of our knowledge, this is the first *mechanized derivation* of a local instance of rely-guarantee reasoning. The instance permits local proofs of both linearizability and lock-freedom. Furthermore, we describe the first mechanized verification of the main aspects of a well-known lock-free stack [8] with explicit memory reuse. Although different versions of the stack have been verified before, these have mainly focused on linearizability and all except one informal proof [9] implicitly assume garbage collection, which significantly simplifies the proofs. A complete presentation of the verification of the theory and its application to several lock-free data structures, is available online [10].

The rest of this paper is structured as follows: Section 2 introduces the stack case study. Section 3 gives a short introduction to the temporal logic framework. In Section 4 we briefly describe the embedding of global rely-guarantee reasoning and derive its local instance and local proof obligations for linearizability and lock-freedom. Section 5 shows the application of this instance to the case study. Section 6 presents related work and a comparison. Finally, Section 7 concludes with a short summary and an overview of our current and future work.

2 The Lock-Free Stack

Lock-free algorithms typically apply atomic synchronization primitives such as CAS (Compare-And-Swap) instead of locks.

```
CAS($Old, New; SV, Succ$) {
  if* $SV = Old$ then {$SV := New,\ Succ := true$} else $Succ := false$}
```

CAS compares a shared value *SV* with an older local copy of it *Old* (called snapshot). If these values are equal *SV* is updated to a new value *New* and true

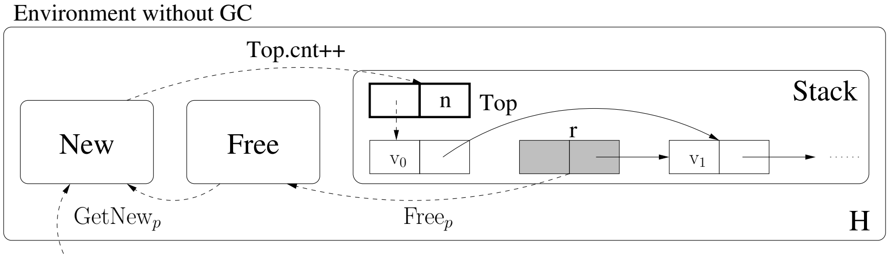


Fig. 1. Data-stack that reuses memory locations from a shared pool (Free).

is returned; otherwise false is returned. Throughout this work we use formal KIV specifications to describe programs. In the specification of CAS, the semicolon separates input from in-output parameters; the comma indicates parallel assignments and in **if*** evaluating the if-condition requires no extra step.

Explicit Memory Reuse Lock-free data structures are often used in programming environments without implicit garbage collection (GC). There, memory locations that are removed from the data structure should be reclaimed in some way to avoid memory leaks. However, removed locations can not be simply deallocated (e.g., using a *free* library call in C/C++), since they are typically still used by concurrent processes. To solve the problem an explicit garbage collection scheme is added to the main algorithm. This paper considers introducing a shared pool of reusable locations as originally proposed by Treiber [8]. (A more advanced solution are hazard pointers [11], see Section 6.) Treiber notes that the possible concurrent reuse of a location can lead to data structure corruption at runtime, when a location is concurrently reinserted in the data structure with a modified content and these intermediate modifications are not detected by CAS. This is a well-known and fundamental problem of CAS-based algorithms, called the ABA-problem. Treiber's solution attaches a modification counter to ABA-prone shared resources so that CAS detects their possible concurrent access.

This work considers a lock-free data-stack that recycles memory from a shared pool of reusable locations (Free), as shown in Figure 1. The stack stores arbitrary data values v_i in a singly linked list of cells (pairs of values and locations having .val and .nxt selector functions) which resides in the application's memory heap H[1]. A shared variable Top marks the top cell of the stack; it is a pair of a reference (location) and a (natural number) modification counter with selector functions .ref and .cnt respectively. A process p which executes a push tries to reuse locations from Free and allocates new ones only if Free is empty (GetNew$_p$). Whenever p pops a location r from the stack, it subsequently adds r to Free (Free$_p$). To detect the concurrent reuse of a location, the modification counter Top.cnt is always incremented atomically with the insertion of a new cell in the stack (Top.cnt++).

[1] H is a partial function from references $r : ref$ (with $null \in ref$) to cells with standard operations, e.g., $r \in H$ tests if r is allocated, $H[r]$ is lookup and $H + r$ is allocation.

```
U1  $Push(In; UNew, USucc, Top, Free, H)$ {
U2    $GetNew(In; UNew, USucc, Free, H)$;
U3    let $UTop = ?$ in {
U4      while $\neg\ USucc$ do {
U5        $UTop := Top$;
U6        $H[UNew].\mathrm{nxt} := UTop.\mathrm{ref}$;
U7        $\mathrm{CAS}(UTop,\ (UNew \times UTop.\mathrm{cnt} + 1); Top, USucc)$}}}

$GetNew(In; UNew, USucc, Free, H)$ {
  choose $r$ with $(r \neq null \wedge (\mathbf{if^*}\ Free = \emptyset\ \mathbf{then}\ r \notin H\ \mathbf{else}\ r \in Free))$ in {
     $Free := Free \setminus \{r\},\ H := H + r,$
     $H[r].\mathrm{val} := In,\ UNew := r,\ USucc := false$}}

O1  $Pop(; OTop, OSucc, Top, Free, H, Out)$ {
O2    let $Lo = empty, ONxt = ?$ in {
        while $\neg\ OSucc$ do {
O3        $OTop := Top$;
O4        if $OTop.\mathrm{ref} = null$ then {
O5          $OSucc := true$;
          } else {
O6          $ONxt := H[OTop.\mathrm{ref}].\mathrm{nxt}$;
O7          $\mathrm{CAS}(OTop,\ (ONxt \times OTop.\mathrm{cnt}); Top, OSucc)$}}
O8      if $OTop.\mathrm{ref} \neq null$ then {
O9        $Lo := H[OTop.\mathrm{ref}].\mathrm{val}'$;
O10       $Free := Free \cup \{OTop.\mathrm{ref}\}, OSucc := false$}
O11     $Out := Lo,\ OSucc := false$}}
```

Fig. 2. Implementation of push and pop.

The Implementation Figure 2 shows the implementation of the stack which is taken from [9] and attributed to [8]. Variables *UNew*, *USucc*, *OTop* and *OSucc* are local variables of "pUsh" resp. "pOp". They are defined as in-output parameters (instead of using **let**) to allow us to reason about them. Whenever a process executes a push, it first allocates and initializes a new cell *UNew* in one step (*GetNew*). Then it repeatedly tries to CAS the shared top pointer to this new cell. (A "?" denotes an arbitrary value.) A pop process reads the shared top in line O3 (if the snapshot's pointer is null, the special value *empty* is returned) and locally stores its next reference which becomes the target of the subsequent CAS. If it succeeds, the top cell is removed from the stack and then added to the shared reference set *Free* (freed). Variable *OSucc* (initially *false*) is used in the verification to characterize removed locations, i.e., locations that have been removed from the stack but not yet freed.

Without a modification counter, an ABA-problem could occur as follows: suppose that a pop-process p takes a snapshot of the top pointer when the stack consists of exactly one cell at location A and the free-list is empty. Process p is delayed after setting its local next reference *ONxt* to null in line O6 for another process. This other process removes A from the stack without yet freeing it.

Subsequently, a further process q executes a successful push, thereby allocating a new location B. Then A is freed and q executes a successful push of A. If now p is rescheduled, its CAS operation in line O7 would erroneously succeed, violating the semantics of pop by deleting the entire stack.

Note that the (double-word) CAS in line O7 atomically compares both a location and a counter. It fails if the snapshot location $OTop$.ref has been concurrently removed from the stack and not reinserted, since it is then not equal to Top.ref. CAS also fails if $OTop$.ref is concurrently reinserted in the stack, since the snapshot's modification counter $OTop$.cnt then does not coincide with Top.cnt ($OTop$.cnt $<$ Top.cnt). Thus, the ABA-problem is avoided. (In Section 5 we intuitively formalize and verify this non-trivial synchronization scheme.)

3 The Verification Framework

This section gives a brief overview of the underlying verification framework. We refer the interested reader to [12, 13] for further details on the syntax and semantics of the logic.

Interval Temporal Logic ITL [4] in KIV is based on algebras, to define the semantics of the signature, [2] and intervals (executions), which are finite or infinite sequences of states. A state maps variables to values. Different from standard ITL, the logic explicitly includes the behavior of a program's environment in each step (similar to reactive sequences in [14]): in an interval $I = [I(0), I'(0), I(1), I'(1), \ldots]$ the first transition from state $I(0)$ to the primed state $I'(0)$ is a program transition, whereas the next transition from state $I'(0)$ to $I(1)$ is a transition of a program's environment. In this manner program and environment transitions alternate. A variable V is evaluated over $I(0)$, whereas its primed resp. double primed version V' resp. V'' is evaluated over $I'(0)$ and $I(1)$ respectively. (For an empty interval [I(0)], both are evaluated over I(0).) E.g., formula $V \neq V'$ denotes that variable V is changed in the first program transition, whereas $V' = V''$ states that V is not changed in the first environment transition. The last state of an interval is characterized by formula **last**.

The logic provides standard temporal operators to describe interval properties: the (weak) next operator $\bullet\ \varphi$ holds in an interval I iff I is either empty, or φ holds in I's postfix interval $[I(1), \ldots]$. $\varphi\, \mathbf{until}\, \psi$ holds in I iff ψ holds in some $[I(n), \ldots]$ and φ holds in $[I(m), \ldots]$ for each $m < n$. Further standard operators are introduced as abbreviations, e.g., $\Diamond\ \varphi \equiv \mathit{true}\, \mathbf{until}\, \varphi$, or $\Box\ \varphi \equiv \neg \Diamond \neg\ \varphi$. In our embedding of rely-guarantee reasoning, temporal formulas of the form $R(V', V'') \xrightarrow{+} G(V, V')$ are of particular interest (cf. Section 4). Predicates G and R specify guarantee resp. rely conditions and the sustains operator $\xrightarrow{+}$ ensures that G is maintained by a program transition if previous environment transitions have preserved R, as shown below. Thus, $R(V', V'') \xrightarrow{+} G(V, V') \equiv \neg\, (R(V', V'')\, \mathbf{until} \neg\, G(V, V'))$, as shown below. Note that G must always hold for the first program transition.

[2] We use higher-order signatures and algebras, see [13]. An example of a higher-order variable is the local state function LSf from Section 4.2.

$$[I(0) \xrightarrow[\subseteq \mathrm{G}]{} I'(0) \xrightarrow[\subseteq \mathrm{R}]{} I(1) \longrightarrow \cdots \longrightarrow I'(n-1) \xrightarrow[\subseteq \mathrm{R}]{} I(n)_{\Rightarrow} \xrightarrow[\subseteq \mathrm{G}]{} I'(n) \ldots]$$

The formal programming language in KIV provides the common sequential constructs. Moreover, it provides a construct for weak-fair interleaving ($\|$) and one for non-fair interleaving ($\|_{\mathrm{nf}}$). Programs α and formulas can be mixed, since they both evaluate to true or false in an interval. In particular, α evaluates to true in I iff I is an execution of α with arbitrary environment steps.

The verification framework is based on the sequent calculus. A sequent is an assertion of the form $\Gamma \vdash \Delta$ (where Γ and Δ are lists of formulas), which states that the conjunction of all formulas in antecedent Γ implies the disjunction of all formulas in succedent Δ. Sequents are implicitly universally closed. A sequent (proof obligation) about concurrent programs α typically has the form $\alpha, E, F \vdash \varphi$ where α executes in an environment constrained by temporal formula E, predicate logic formula F describes the current state and φ is the property of interest. As a simple example, consider the following sequent.

$$(M := M + 1;\ \beta),\ M = 1 \ \vdash\ M' = M'' \xrightarrow{+} M' > M \tag{1}$$

The executed program is $(M := M{+}1;\ \beta)$ where β is a program and environment behavior is unrestricted ($E = \mathit{true}$ omitted). The current state maps counter M to 1 and it has to be shown that the program always increases M if previous environment transitions have not changed it ($M' = M'' \xrightarrow{+} M' > M$).

Symbolic Execution Sequents that contain temporal assertions are verified by symbolically stepping forward to the next states of an interval, calculating strongest postconditions for each program transition, which are then possibly weakened according to assumptions for the following environment transition. Restricting environment transitions to never change any program variables yields the sequential setting. Thus, the calculus is rather similar to classic symbolic execution of sequential programs [5], but in a concurrent setting.

A step is executed in two implicit phases which concern programs as well as formulas. In the first phase, information about the first program and environment transition is separated from information about the rest of an interval by applying unwinding rules. A program is unwound by calculating the effect of its first statement; $\xrightarrow{+}$ is unwound according to the following rule:

$$R \xrightarrow{+} G \ \leftrightarrow\ G \wedge (R \to \bullet\ (R \xrightarrow{+} G))$$

Applying this rule to the succedent of (1) yields $M' > M \wedge (M' = M'' \to \bullet\ (M' = M'' \xrightarrow{+} M' > M))$. That is, we must prove that M is increased in the first program transition ($M' > M$) as a first subgoal. If the following environment transition leaves M unchanged ($M' = M''$), then the sustains formula must further hold in the rest of the interval ($\bullet\ (M' = M'' \xrightarrow{+} M' > M)$). The second phase of a symbolic execution step "moves" to the rest of an interval by eliminating leading next operators. This leads to the following further subgoal when proving (1):

$$\beta,\ M = 2 \ \vdash\ M' = M'' \xrightarrow{+} M' > M$$

Induction Well-founded induction is used to deal with loops. For infinite intervals, a term for well-founded induction can be derived from a known liveness property $\diamond\ \varphi$ as the number N of steps until φ holds.

$$\diamond\ \varphi \ \leftrightarrow\ \exists\ N.\ (N = N'' + 1)\ \textbf{until}\ \varphi$$

The fresh variable N is decremented in each step until φ becomes true. (Note that $N = N'' + 1$ is equivalent to $N'' = N - 1 \wedge N > 0$.)

The proof of a sustains formula on an infinite interval I can be carried out by induction over the length of an arbitrary finite I-prefix.

$$R \xrightarrow{+} G \ \leftrightarrow\ \forall\ B.\ (\diamond\ B) \rightarrow ((R \wedge \neg\ B) \xrightarrow{+} G)$$

The fresh boolean B characterizes the length of the prefix, which ends as soon as B becomes true for the first time. Again, the number of steps until B holds is used for well-founded induction.

4 Deriving Local Rely-Guarantee Conditions for Linearizability and Lock-Freedom

Rely-guarantee reasoning basically defines proof obligations for individual components of a concurrent system instead of reasoning about their interleaved execution. This section briefly describes our concurrent system model and outlines our embedding of global rely-guarantee reasoning (cf. [7]). Then it derives its local instance – which is simpler to use when verifying concrete systems with similar components – in detail and briefly defines local proof obligations for linearizability and lock-freedom.

4.1 The Concurrent System Model and Global Rely-Guarantee Reasoning

The Concurrent System Model As shown in Figure 3, our generic concurrent system SPAWN recursively spawns $n+1$ components ($n : \mathbb{N}$) to execute in parallel. Operation SEQ defines the possible sequential behaviors of each component $p : \mathbb{N}$. Either p instantly terminates or it executes finitely or infinitely often – as denoted by the star operator * – a generic interface procedure COP or *skip* which models steps that are unrelated to COP. [3] The unspecified procedure COP models arbitrary operations that p can execute on the overall concurrent system state CS : *cstate*. Functions $\mathit{Inf} : \mathbb{N} \rightarrow \mathit{input}$ and $\mathit{Outf} : \mathbb{N} \rightarrow \mathit{output}$ are used to insert or return values $\mathit{Inf}(p)$ and $\mathit{Outf}(p)$ respectively.

Global Rely-Guarantee Reasoning To avoid tedious reasoning about interleaved executions of SPAWN, we have embedded rely-guarantee reasoning in

[3] The auxiliary function $\mathit{Actf} : \mathbb{N} \rightarrow \mathit{bool}$ distinguishes whether a component executes COP (i.e., is active) or not, since the logic does not use program counters. This is mainly relevant for the decomposition proof of lock-freedom.

```
SPAWN($n$; $Actf, Inf, CS, Outf$) {
  if* $n = 0$ then
    SEQ($0$; $Actf, Inf, CS, Outf$)
  else
      SEQ($n$; $Actf, Inf, CS, Outf$)
    || SPAWN($n - 1$; $Actf, Inf, CS, Outf$)}
```

```
SEQ($p$; $Actf, Inf, CS, Outf$) {
  {   {$Actf(p) := true$;
       COP($p, Inf(p)$; $CS, Outf(p)$);
       $Actf(p) := false$}
    ∨ skip}* }
```

Fig. 3. The concurrent system model.

the logical framework. In this embedding specifications use the overall concurrent system state CS and thus we call it "global". The basic idea is to abstract away from environment behavior using rely conditions $R_{ext} \subseteq \mathbb{N} \times cstate \times cstate$ for each component p and to guarantee a certain behavior towards p's environment according to guarantee conditions $G_{ext} \subseteq \mathbb{N} \times cstate \times cstate$. Our central rely-guarantee proof obligation for an individual component p then claims that in p's execution of $\mathrm{COP}(p, \dots)$, each program transition satisfies $G_{ext}(p, \dots)$ if the preceding environment transitions have preserved $R_{ext}(p, \dots)$.

$$\mathrm{COP}(p, Inf(p); CS, Outf(p)) \ \vdash R_{ext}(p, CS', CS'') \xrightarrow{+} G_{ext}(p, CS, CS') \tag{2}$$

We introduce further subpredicates to structure G_{ext} and R_{ext} into three categories: step invariant guarantee and rely conditions $G, R \subseteq \mathbb{N} \times cstate \times cstate$, state invariant conditions $Inv \subseteq cstate$ and local idle state conditions $Idle \subseteq \mathbb{N} \times cstate$ which hold before and after each finite execution of COP. The full version of (2) which takes into account these structural predicates simply is:

$$\begin{aligned}
&\mathrm{COP}(p, Inf(p); CS, Outf(p)),\ Inv(CS),\ Idle(p, CS) \vdash R_{ext} \xrightarrow{+} G_{ext}\\
&\text{where } G_{ext} :\leftrightarrow \quad G(p, CS, CS') \wedge (Inv(CS) \rightarrow Inv(CS'))\\
&\qquad\qquad\qquad \wedge (\mathbf{last} \rightarrow Idle(p, CS)) \wedge \forall\, q \neq p.\ Idle(q, CS) \leftrightarrow Idle(q, CS')\\
&\text{and } R_{ext} :\leftrightarrow \quad R(p, CS', CS'') \wedge (Inv(CS') \rightarrow Inv(CS''))\\
&\qquad\qquad\qquad \wedge (Idle(p, CS') \leftrightarrow Idle(p, CS''))
\end{aligned} \tag{3}$$

Program steps in $\mathrm{COP}(p, \dots)$ executions maintain $G(p, \dots)$, Inv and establish $Idle(p, \dots)$ in their last state (and do not change the idle state assumptions of other components q), as long as environment transitions maintain $R(p, \dots)$, Inv and $Idle(p, \dots)$ respectively. This embedding makes two improvements over our previous embedding [7]. First, the invariant is now decoupled from R and G to avoid unnecessarily strong rely resp. guarantee conditions. Second, we have introduced predicate $Idle$ to express local, idle state conditions.

Proving that these predicates hold indeed in every execution of SPAWN can be decomposed to basically showing (3) for an arbitrary component, according to the following theorem (cf. [7, 10] for technical details).

Theorem 1 (Global Rely-Guarantee Reasoning).
If (3) holds for an arbitrary overall system state CS and some transitive R, reflexive

G with $G(p, CS, CS') \rightarrow R(q, CS, CS')$, $\forall\, q \neq p$, *and predicates* $Inv, Idle$, *then:*

$$\textsc{Spawn},\ \Box\ R_{\textsc{Spawn}},\ Init_{\textsc{Spawn}} \vdash \Box\ ((\exists\ p.\ G(p, \dots)) \land \varphi_{Inv} \land \varphi_{Idle})$$

Spawn starts in an initial state satisfying $Init_{\textsc{Spawn}}$, which must imply Inv and $Idle$ for all components (these are initially inactive). The *system*'s environment behavior is restricted by rely $R_{\textsc{Spawn}}$, which is the identity relation over the in-output parameters of Spawn. Then each system step of a component p always satisfies $G(p, \dots)$, invariant Inv holds in each state according to φ_{Inv} and any component is idle before and after it executes COP, according to φ_{Idle}.

4.2 Deriving Process-Local Rely-Guarantee Reasoning

Theorem 1 can be applied in scenarios where each component exhibits a different behavior (e.g., the producer-channel-consumer described in [12] where $n = 2$ and $\mathrm{COP}(0, \dots)$ is the producer, $\mathrm{COP}(1, \dots)$ the channel and $\mathrm{COP}(2, \dots)$ the consumer) since specifications account for the whole system state CS, including all local states. However, this expressiveness is often not required when components have similar behaviors, in particular when all components execute the operations of a concurrent data type. As an example, consider the global specification of the following simple invariant of the stack from Section 2 where components are concurrent processes that execute push or pop: pointers to new cells – which are not yet inserted in the stack – are disjoint during concurrent push operations.

$$\forall\ p \neq q.\ \neg\ USuccf(p) \land \neg\ USuccf(q) \rightarrow UNewf(p) \neq UNewf(q)$$

Global specifications require variable functions (e.g., $UNewf : \mathbb{N} \rightarrow ref$ for variable $UNew$ in push) and quantification over all identifiers p, q. Thus they are less succinct and harder to read. Moreover, proofs that use such specifications are harder to automate, since finding right quantifier instantiations often fails.

However, the frequent case of concurrent data type implementations permits local specifications that consider say two representative components p resp. q with local states $LS : lstate$ resp. $LSQ : lstate$. The encoding of the aforementioned invariant then simply is:

$$\neg\ USucc \land \neg\ USuccq \rightarrow UNew \neq UNewq \qquad (4)$$

From Global to Local Rely-Guarantee Specifications The reduction to local specifications is based on splitting CS into its local and shared parts $LSf \times S$ where $LSf : \mathbb{N} \rightarrow lstate$ maps each component to its local state and $S : sstate$ is the shared state. Each component p now executes the same procedure $\mathrm{LCOP}(Inf(p); LSf(p), S, Outf(p))$. In the stack case study, LCOP is the nondeterministic choice between one of the operations that each process can execute.

$$\mathrm{LCOP}(In; LS, S, Out)\ \{Push(In; LS, S) \lor Pop(;\ LS, S, Out)\}$$

The shared state S of the stack consists of the shared variables Top, $Free$, H for the top-of-stack pointer, the free-set and the application's heap, whereas the local state LS is the tuple of the local variables $UNew$, $USucc$, $OTop$ and $OSucc$.

Furthermore, our local rely-guarantee embedding introduces a new invariant predicate *LDisj* to encode disjointness properties, such as (4), between the two local states. [4] The local counterpart of (3) now is:

$$\begin{array}{ll}
\mathrm{LCOP}(\mathit{In}; \mathit{LS}, S, \mathit{Out}),\ \mathit{LIID}(\mathit{LS}, \mathit{LSQ}, S),\ \mathit{LIdle}(\mathit{LS}) \vdash \mathit{LR}_{ext} \xrightarrow{+} \mathit{LG}_{ext} & \\
\text{where } \mathit{LG}_{ext} :\leftrightarrow \quad \mathit{LG}(\mathit{LS}, \mathit{LSQ}, S, \mathit{LS}', S') \wedge (\mathbf{last} \rightarrow \mathit{LIdle}(\mathit{LS})) & \\
\qquad\qquad \wedge\ (\mathit{LIID}(\mathit{LS}, \mathit{LSQ}, S) \rightarrow \mathit{LIID}(\mathit{LS}', \mathit{LSQ}', S')) & \\
\text{and } \mathit{LR}_{ext} :\leftrightarrow \quad \mathit{LS}' = \mathit{LS}'' \wedge \mathit{LR}(\mathit{LS}', S', S'') & (5) \\
\qquad\qquad \wedge\ (\mathit{LIID}(\mathit{LS}', \mathit{LSQ}', S') \rightarrow \mathit{LIID}(\mathit{LS}'', \mathit{LSQ}'', S'')) & \\
\text{and } \mathit{LIID}(\mathit{LS}, \mathit{LSQ}, S) :\leftrightarrow \mathit{LInv}(\mathit{LS}, S) \wedge \mathit{LInv}(\mathit{LSQ}, S) \wedge \mathit{LDisj}(\mathit{LS}, \mathit{LSQ}) &
\end{array}$$

Similar to (3), LCOP-steps must maintain the local guarantee conditions *LG* and the local state invariants *LIID*, plus, they must establish the local idle state *LIdle*, as long as environment transitions do not modify *LS* and they maintain the local rely *LR* and *LIID* respectively. A more detailed description of the local structural predicates is given in the following; their instantiation in the stack case study is shown in detail in Section 5.

The Local Structural Predicates The first three parameters of $LG \subseteq \mathit{lstate} \times \mathit{lstate} \times \mathit{sstate} \times \mathit{lstate} \times \mathit{sstate}$ denote the local states of the two components and the shared state before a program transition; the last two parameters stand for the executing component's local state and the shared state after this transition. Predicate $\mathit{LIdle} \subseteq \mathit{lstate}$ encodes local, idle state conditions that hold between finite executions of LCOP. In the case study for example, idle states satisfy the following local restrictions: $\mathit{LIdle}(\mathit{LS}) :\leftrightarrow \mathit{USucc} \wedge \neg\ \mathit{OSucc}$.

The first parameter of $LR \subseteq \mathit{lstate} \times \mathit{sstate} \times \mathit{sstate}$ corresponds to a component's local state before an environment transition. The second resp. third parameter is the shared state before resp. after this transition. In the case study, *LR* ensures for instance that the content of a new cell in push is not changed by the environment if this cell is not yet inserted in the stack.

$$\neg\ \mathit{USucc}' \rightarrow H''[\mathit{UNew}'] = H'[\mathit{UNew}'] \tag{6}$$

Together we can prove the following local decomposition theorem for Spawn where *CS* is replaced by $\mathit{LSf} \times S$ and COP by LCOP respectively:

Theorem 2 (Local Rely-Guarantee Reasoning). *If (5) holds for two arbitrary disjoint local states LS, LSQ, the shared state S and some transitive rely LR, reflexive predicate LG with $LG(LS, LSQ, S, LS', S') \rightarrow LR(LSQ, S, S')$, symmetric predicate LDisj and predicates LInv, LIdle, then:*

$$\textsc{Spawn}, \Box\ R_{\textsc{Spawn}}, \mathit{Init}_{\textsc{Spawn}} \vdash \Box\ ((\exists\ p.\ \varphi_{LG}(p)) \wedge \varphi_{LI} \wedge \varphi_{LIdle})$$

where $\varphi_{LG}(p)$ states that the system step of a component p does not modify the local states of other components q and it satisfies LG,

$$\varphi_{LG}(p) :\leftrightarrow \forall\ q \neq p.\ \mathit{LSf}(q) = \mathit{LSf}'(q) \wedge LG(\mathit{LSf}(p), \mathit{LSf}(q), S, \mathit{LSf}'(p), S')$$

[4] These are part of *Inv* in the global rely-guarantee theory.

the invariant conditions hold for all components at all times,

$$\begin{aligned}\varphi_{LI} :\leftrightarrow \forall\, p \neq q. \quad & \mathit{LInv}(\mathit{LSf}(p), S) \wedge \mathit{LInv}(\mathit{LSf}'(p), S') \\ & \wedge \mathit{LDisj}(\mathit{LSf}(p), \mathit{LSf}(q)) \wedge \mathit{LDisj}(\mathit{LSf}'(p), \mathit{LSf}'(q))\end{aligned}$$

and each p is idle before and after LCOP*:*

$$\varphi_{LIdle} :\leftrightarrow \forall\, p.\ \neg\ \mathit{Actf}(p) \rightarrow \mathit{LIdle}(\mathit{LSf}(p))$$

Proof. By instantiating CS with $\mathit{LSf} \times S$, COP with LCOP, predicates Inv, Idle, G and R with predicates $\mathit{Inv}_\natural$, $\mathit{Idle}_\natural$, $G_\natural$ and $R_\natural$ as given below and verifying that the preconditions of Theorem 1 follow from those of Theorem 2.

$$\begin{aligned}&\mathit{Inv}_\natural(\mathit{LSf}, S) :\leftrightarrow \forall\, p \neq q.\ \mathit{LInv}(\mathit{LSf}(p), S) \wedge \mathit{LDisj}(\mathit{LSf}(p), \mathit{LSf}(q)) \\ &\mathit{Idle}_\natural(p, \mathit{LSf}, S) :\leftrightarrow \mathit{LIdle}(\mathit{LSf}(p)); \quad G_\natural(p, \mathit{LSf}, S, \mathit{LSf}', S') :\leftrightarrow \varphi_{LG}(p) \\ &R_\natural(p, \mathit{LSf}', S', \mathit{LSf}'', S'') :\leftrightarrow \mathit{LSf}'(p) = \mathit{LSf}''(p) \wedge \mathit{LR}(\mathit{LSf}'(p), S', S'')\end{aligned}$$

4.3 Local Proof Obligations for Linearizability and Lock-Freedom

Linearizability [2] and lock-freedom [3] are major correctness resp. progress properties of concurrent systems. In this section we define local proof obligations for LCOP which imply linearizability and lock-freedom of Spawn. They are based on invariant properties LISR with one local state LS that each component may always assume during its execution of LCOP($\mathit{In}; \mathit{LS}, S, \mathit{Out}$), according to Theorem 2.

$$\mathit{LISR} :\leftrightarrow \mathit{LInv}(\mathit{LS}, S) \wedge \mathit{LInv}(\mathit{LS}', S') \wedge \mathit{LS}' = \mathit{LS}'' \wedge \mathit{LR}(\mathit{LS}', S', S'')$$

Every component can assume LInv at all times according to φ_{LI}. Since $\varphi_{\mathrm{LG}}(p)$ implies that each component p does not modify other local states and satisfies its guarantee, each component can in return also assume that its local state is never concurrently changed and that its rely holds at all times (recall $\mathit{LG} \rightarrow \mathit{LR}$).

Linearizability Based on these assumptions established by rely-guarantee reasoning, we prove linearizability by locating the linearization point (i.e., the step where a call appears to take effect) of each operation in LCOP. [5] Conceptually, the linearization point is determined in a refinement proof using an abstraction function $\mathit{Abs} \subseteq \mathit{sstate} \times \mathit{astate}$ (a partial function on shared states that satisfy LInv, which returns a corresponding abstract state). In the stack example, Abs maps the stack in memory to a finite algebraic list St of its data values.

$$\begin{aligned}&\mathit{Abs}(\mathit{Top}.\mathrm{ref}, H, [\,]) :\leftrightarrow \mathit{Top}.\mathrm{ref} = \mathit{null} \\ &\mathit{Abs}(\mathit{Top}.\mathrm{ref}, H, v + \mathit{St}) :\leftrightarrow \quad \mathit{Top}.\mathrm{ref} \neq \mathit{null} \wedge \mathit{Top}.\mathrm{ref} \in H \\ &\qquad\qquad \wedge H[\mathit{Top}.\mathrm{ref}].\mathrm{val} = v \wedge \mathit{Abs}(H[\mathit{Top}.\mathrm{ref}].\mathrm{nxt}, H, \mathit{St})\end{aligned}$$

[5] Our current approach suffices when a linearization point is within the code of the executing component, even when its location depends on future behavior. This is possible, since analyzing future states of an interval is possible in ITL (cf. [7] for a detailed description of such an example.)

```
$APush(In; St)$ {
  $skip^*$;
  $St := push(In, St)$;
  $skip^*$}
```

```
$APop(; St, Out)$ {
  let $Lo = empty$ in {
    $skip^*$;
    if* $St \neq [\,]$ then {
      $Lo := top(St), St := pop(St)$};
    $skip^*$; $Out := Lo$}}
```

Fig. 4. Abstract stack operations.

To prove linearizability, one has to show that each concrete operation from LCOP non-atomically refines a corresponding abstract operation, which is defined in a further generic procedure AOP. In the case study, AOP is the nondeterministic choice between an abstract *APush* or *APop*, which are shown in Figure 4. They use atomic operations *push* resp. *pop* to add resp. remove an element from *St* at concrete linearization points and additional skip steps at non-linearization points.

Refinement (i.e., interval inclusion) between LCOP and AOP is simply expressed as LCOP ⊢ AOP in the logical framework. Hence, the local refinement proof obligation for linearizability is:

$$\begin{array}{l}\mathrm{LCOP}(In; LS, S, Out),\ \Box\ (LISR \wedge Abs(S, AS) \wedge Abs(S', AS')),\ LIdle(LS) \vdash \\ \mathrm{AOP}(In; AS, Out)\end{array} \tag{7}$$

Lock-Freedom A concurrent system is lock-free if some of its running operations always terminates in a finite number of steps, even if individual components are arbitrarily delayed or fail. In our concurrent system model (Fig. 3), this is modeled by requiring that some active operation (expressed using activity function *Actf*) always eventually becomes inactive. This is true, even if the scheduling is non-fair $\|_{\mathrm{nf}}$ (to model failure), as discussed in [15], p. 393 and following. (Also see [10] for full proofs.) However, individual components of a lock-free system might starve. In the stack example, single push and pop operations can be forced to always retry their loop if another process modifies the shared top pointer. Yet, if such an interference always occurs, it is an interfering process which terminates its current execution and without interference, the current process eventually terminates. We formalize this intuitive argument using an additional reflexive and transitive relation $U \subseteq sstate \times sstate$ ("unchanged") which describes interference freedom. Note that U represents an unbounded amount of interference which a process might "suffer" from *or* perform. For the stack, we determine the "unchanged" relation as identity of the shared variable *Top*.

$$U(S_0, S_1) :\leftrightarrow Top_0 = Top_1$$

To prove lock-freedom (based on rely-guarantee conditions *LISR*), two local termination proofs for each operation in LCOP are sufficient: termination without interference from the *environment* ($\Box\ U(S', S'') \rightarrow \Diamond\ \mathbf{last}$) and termination after violation of U by the *program* ($\neg\ U(S, S') \rightarrow \Diamond\ \mathbf{last}$):

$$\begin{array}{l}\mathrm{LCOP}(In; LS, S, Out),\ \Box\ LISR,\ LIdle(LS) \vdash \\ \Box\ ((\Box\ U(S', S'')) \vee \neg\ U(S, S') \rightarrow \Diamond\ \mathbf{last})\end{array} \tag{8}$$

5 Local Verification of the Stack

This section describes the application of the local decomposition theory to verify memory-safety, ABA-prevention, linearizability and lock-freedom of a concurrent stack application $\textsc{Spawn}(n; \dots)$ where all $n + 1$ processes execute the push and pop operations from Figure 2. The specifications and proofs consider at most two representative processes. The explicit reuse of memory locations makes the verification notably more challenging than proving the stack under the assumption of GC, which implicitly avoids the reuse of a memory location that is referenced in some operation.

5.1 Instantiating the Local Predicates

LInv Predicate *LInv* encodes several state invariant properties of the stack $\mathit{LInv} :\leftrightarrow \varphi_{st} \wedge \varphi_n \wedge \varphi_{\mathrm{free}} \wedge \varphi_t$. According to φ_{st}, the implementation represents some finite list, i.e., $\mathit{Abs}(\mathit{Top}.\mathrm{ref}, H, \mathit{St})$ always holds for some St.

$$\varphi_{st} :\leftrightarrow \exists\ \mathit{St}.\ \mathit{Abs}(\mathit{Top}.\mathrm{ref}, H, \mathit{St})$$

To maintain this property new cells that are to be pushed on the stack must be allocated and disjoint from the stack according to φ_n. (A standard reachability predicate $\mathit{reach}(\mathit{Top}.\mathrm{ref}, r, H)$ checks whether a location r is in the stack.)

$$\varphi_n :\leftrightarrow \neg\ \mathit{USucc} \rightarrow \mathit{UNew} \neq \mathit{null} \wedge \mathit{UNew} \in H \wedge \neg\ \mathit{reach}(\mathit{Top}.\mathrm{ref}, \mathit{UNew}, H)$$

Invariant φ_{free} ensures major safety aspects of the memory reclamation scheme: freed locations $r \in \mathit{Free}$ are allocated and disjoint from the stack, since otherwise the reuse of r would cause an access error or corrupt the stack; r is also disjoint from new cells and from removed locations (i.e., removed from the stack but not yet freed $\mathit{OSucc} \wedge \mathit{OTop}.\mathrm{ref} \neq \mathit{null}$) and thus the memory pool is duplicate-free.

$$\begin{aligned}\varphi_{\mathrm{free}} :\leftrightarrow \forall\ r \in \mathit{Free}.\quad & r \neq \mathit{null} \wedge r \in H \wedge \neg\ \mathit{reach}(\mathit{Top}.\mathrm{ref}, r, H)\\ & \wedge (\neg\ \mathit{USucc} \rightarrow r \neq \mathit{UNew})\\ & \wedge (\mathit{OSucc} \wedge \mathit{OTop}.\mathrm{ref} \neq \mathit{null} \rightarrow r \neq \mathit{OTop}.\mathrm{ref})\end{aligned}$$

We must also know that locations $\mathit{OTop}.\mathrm{ref} \neq \mathit{null}$ are allocated and that removed locations are disjoint from the stack (φ_t).

$$\begin{aligned}\varphi_t :\leftrightarrow\ & \quad (\mathit{OTop}.\mathrm{ref} \neq \mathit{null} \rightarrow \mathit{OTop}.\mathrm{ref} \in H)\\ & \wedge (\mathit{OSucc} \wedge \mathit{OTop}.\mathrm{ref} \neq \mathit{null} \rightarrow \neg\ \mathit{reach}(\mathit{Top}.\mathrm{ref}, \mathit{OTop}.\mathrm{ref}, H))\end{aligned}$$

LDisj Three disjointness properties between local pointers of the two processes are used. To ensure symmetry we define $\mathit{LDisj}(\mathit{LS}, \mathit{LSQ}) :\leftrightarrow \mathit{disj}(\mathit{LS}, \mathit{LSQ}) \wedge \mathit{disj}(\mathit{LSQ}, \mathit{LS})$ where $\mathit{disj}(\mathit{LS}, \mathit{LSQ}) :\leftrightarrow (4) \wedge \delta_{rm} \wedge \delta_{tn}$. Property δ_{rm} states that concurrently removed locations are disjoint, whereas δ_{tn} ensures that removed locations are disjoint from concurrent new cells.

$$\begin{aligned}\delta_{rm} :\leftrightarrow\ & \quad \mathit{OSucc} \wedge \mathit{OTop}.\mathrm{ref} \neq \mathit{null} \wedge \mathit{OSuccq} \wedge \mathit{OTopq}.\mathrm{ref} \neq \mathit{null}\\ & \rightarrow \mathit{OTop}.\mathrm{ref} \neq \mathit{OTopq}.\mathrm{ref}\\ \delta_{tn} :\leftrightarrow\ & \mathit{OSucc} \wedge \mathit{OTop}.\mathrm{ref} \neq \mathit{null} \wedge \neg\ \mathit{USuccq} \rightarrow \mathit{OTop}.\mathrm{ref} \neq \mathit{UNewq}\end{aligned}$$

LR We define several rely conditions $LR :\leftrightarrow \rho_{st} \wedge \rho_{nst} \wedge \rho_{ge} \wedge \rho_{rm} \wedge (6)$ to intuitively formalize the non-trivial synchronization mechanism that avoids the ABA-problem. In particular, rely conditions ρ_{st} and ρ_{nst} make sure that during a pop, the ABA-prone location $OTop$.ref either stays in the stack and its contents are unchanged or if it is concurrently removed, then $OTop$.ref is not reinserted in the stack or the modification counter is increased.

$$\begin{aligned}\rho_{st} :\leftrightarrow \quad & \neg\ OSucc' \wedge OTop'.\mathrm{ref} \neq null \wedge Top' = OTop' \\ \rightarrow \quad & Top'' = Top' \wedge H''[OTop'.\mathrm{ref}] = H'[OTop'.\mathrm{ref}] \\ & \vee \neg\ reach(Top''.\mathrm{ref}, OTop'.\mathrm{ref}, H'') \vee Top''.\mathrm{cnt} > Top'.\mathrm{cnt}\end{aligned}$$

$$\begin{aligned}\rho_{nst} :\leftrightarrow \quad & \neg\ OSucc' \wedge OTop'.\mathrm{ref} \neq null \wedge \neg\ reach(Top'.\mathrm{ref}, OTop'.\mathrm{ref}, H') \\ & \rightarrow \neg\ reach(Top''.\mathrm{ref}, OTop'.\mathrm{ref}, H'') \vee Top''.\mathrm{cnt} > Top'.\mathrm{cnt}\end{aligned}$$

The remaining simple relies ensure that the modification counter never decreases (ρ_{ge}) and that the content of removed locations is unchanged (ρ_{rm}).

$$\begin{aligned}&\rho_{ge} :\leftrightarrow Top'.\mathrm{cnt} \leq Top''.\mathrm{cnt} \\ &\rho_{rm} :\leftrightarrow OSucc' \wedge OTop'.\mathrm{ref} \neq null \rightarrow H'[OTop'.\mathrm{ref}] = H''[OTop'.\mathrm{ref}]\end{aligned}$$

LG The reclamation scheme avoids memory leaks, i.e., all heap locations r are either in the stack or in the free-set or owned by a process at all times in each execution of Spawn, where every process owns its new and removed locations.

$$\begin{aligned}&owns(r, LS) : \leftrightarrow \\ &\quad (\neg\ USucc \wedge UNew = r) \vee (OSucc \wedge OTop.\mathrm{ref} \neq null \wedge OTop.\mathrm{ref} = r)\end{aligned}$$

We decompose the absence of memory leaks to a local guarantee *noleaks*, which ensures that process steps do not create leaks.

$$\begin{aligned}&noleaks(LS, S, LS', S') :\leftrightarrow \\ &\quad \forall\ r. \quad r \notin H \vee reach(Top, r, H) \vee r \in Free \vee owns(r, LS) \\ &\qquad \rightarrow r \notin H' \vee reach(Top', r, H') \vee r \in Free' \vee owns(r, LS')\end{aligned}$$

Predicate LG is then defined to maintain *noleaks* and the rely conditions of the other process $LG(LS, LSQ, \dots) :\leftrightarrow noleaks(\dots) \wedge LR(LSQ, \dots)$.

5.2 The Main Proofs

The main effort of the case study is to prove (5) –sustainment of the verification conditions LG, *LInv* and *LDisj* for the steps of each operation if the environment has previously maintained LR. We proceed by case analysis over operation $Op \in \{Push, Pop\}$. The proof resembles a Hoare-style proof of a sequential program. In particular, before executing a loop we generalize the current state assumptions to a Hoare-style invariant (and use $\xrightarrow{+}$ induction when the loop is reiterated). Each program statement in *Op* is consecutively, symbolically executed according to Section 3. Only some major arguments are outlined.

Sustainment of the Verification Conditions $Op \equiv Push$: The allocation step (*GetNew*) resets the content of a new cell. However, if the free-set is empty, this step does not affect allocated locations and otherwise invariant φ_{free} ensures that no rely conditions of the other process are violated.

$Op \equiv Pop$: After taking the snapshot in line O3 in case of a non-null top-of-stack pointer, the proof proceeds by case distinction: according to rely condition ρ_{st} there are three possible cases in the next state. First, when the shared top-of-stack pointer has not been concurrently modified, the content of the snapshot location is unchanged and the current iteration can still succeed correctly. In the second resp. third case, the snapshot location $OTop$.ref is either not in the stack anymore or it has been reinserted and thus the modification counter has been increased. In both latter cases rely conditions ρ_{nst} and ρ_{ge} ensure that the loop must be reiterated. Hence the CAS can not erroneously succeed and cause an ABA-problem.

Linearizability The proof of linearizability (proof obligation (7)) distinguishes between the two possible concrete operations. In case of a push operation, the linearization point is the successful CAS. Rely (6) ensures that the initial value of the new cell and its next reference are immutable. Hence, the successful CAS corresponds to an abstract push of the invoked value. The pop operation has one linearization point in line O3 if the stack is empty, or else in line O7 if the CAS succeeds. Relies ρ_{st} and ρ_{rm} ensure that the successful CAS corresponds to an abstract pop and that the correct value is returned.

Lock-Freedom According to (8), the proof of lock-freedom requires termination proofs for each data structure operation if environment behavior is restricted according to U and if a step violates U. The termination proofs for push and pop mainly automatically step through the code until an operation terminates or they apply induction whenever a loop is retried. The required term for induction is extracted from the always formula in the succedent of (8).

Verification Effort in KIV The soundness proofs for the improved global decomposition theory took about four man-weeks. In particular, the decomposition proof of lock-freedom is tedious as it must consider many possible interleavings. The derivation of the local instance took about two man-weeks. The main challenge was to find the right local proof obligations and the instantiation of the global predicates (see proof of Theorem 2). Using the local instead of the global theory to verify the stack under GC reduced the size of the verification conditions by around one third. The verification of the stack with explicit memory reuse took around two man-weeks and was about twice as complex as verifying the stack under GC. The main new challenge was to find the right heap invariants that ensure memory-safety (φ_{free}, *noleaks*) and the rely conditions for ABA-prevention (ρ_{st}, ρ_{nst}).

6 Related Work and Comparison

Compositional Verification Most approaches to compositional reasoning justify the rules they use on a semantic level (e.g., [14], [16]). A mechanized soundness and completeness proof for global rely-guarantee rules for interleaved programs with shared variables has been given by Nieto et al [17]. The verification is based on Isabelle's higher-order logic, and therefore in essence had to explicitly formalize intervals (using a small-step semantics for programs). Since our proof is based on a strong temporal logic (instead of just HOL), where intervals are already part of the semantics, proving the soundness of rely-guarantee rules using ITL is much simpler in our setting.[6]

Local Rely-Guarantee Proof Obligations There are two approaches [18, 19] which combine rely-guarantee and separation logic for heap-modular reasoning. Our work borrows this idea to achieve process-local reasoning and complements their work by also considering liveness (lock-freedom). Separation logic's operator $*$ and the framing rule permits to "hide" heap disjointness invariants, which we encode explicitly. Fu et al. [20] define a temporal logic of the past to manually verify ABA-prevention for a lock-free stack with hazard pointers [11]. The local rely-guarantee instance presented here mechanizes such proofs, and allows to *additionally* mechanize linearizability and lock-freedom proofs of this challenging algorithm. This is demonstrated in [21] which briefly sketches the main ideas of the local rely-guarantee theory layed out in this paper, and then mainly focusses on the generic verification of lock-free algorithms with hazard pointers. In contrast, this work gives a detailed presentation of our process-local rely-guarantee reasoning approach and outlines its application using another well-known lock-free memory reclamation scheme.

In general, techniques that exploit the symmetry of identical system components have also been developed in model checking (cf. [22] for an overview). However, proving linearizability using (symmetric) model checking often fails (cf. [23] and [24] for recent work on model checking linearizability). Model checking is good at finding bugs in lock-free algorithms by showing counter examples. However, since it checks short executions of a few processes only, it does not give full proofs.

Verification of Linearizability Mechanized verification approaches for linearizability can be roughly classified into three categories: automated approaches based on shape analysis and separation logic, and interactive approaches. Automated approaches can verify the stack example assuming garbage collection, see [25] and [26], and the latter is able to solve many interesting examples automatically, including some cases where linearization points lie outside of the code of the executing thread. Our proof obligations for linearizability have to be generalized to handle some of these examples. However, verification under GC is much simpler than verification using modification counters (which is still simpler than with hazard pointers).

The work most closely related to ours is Doherty, Groves et al. [27], which verifies linearizability of a lock-free queue with modification counters in PVS. The approach is related to ours in also using refinement to prove linearizability. It is global however, and has to encode the algorithms as a concurrent IO-Automaton. Lock-freedom is not discussed. Later on, Groves et al. [9] gave a manual verification approach for the stack with modification counters, based on trace reduction and incremental refinement. Our impression is that mechanizing their arguments

[6] We have justified some of the more difficult rules of ITL using an embedding of the semantics into HOL. Proofs over this theory are rather complex too. Like many others, the embedding of ITL into HOL is not usable to verify case studies.

about commuting steps would be hard. Nevertheless our verification benefited from knowing many of their informal arguments.

Verification of Lock-Freedom Gotsman et al. [28] developed a new logic for proving liveness properties of non-blocking algorithms based on rely-guarantee reasoning and separation logic. Their approach can automatically discharge manually derived proof obligations for lock-freedom, using a combination of tools. In contrast, we mechanically verify both decomposition theorems for safety and liveness properties of concurrent programs and local proof obligations in one logical framework and tool.

7 Conclusion

We have described a mechanically derived local rely-guarantee instance. Such local instances are useful to avoid reasoning about the overall system state when verifying concurrent algorithms where components have similar behaviors, e.g., lock-free data type implementations. Moreover, we have defined local proof obligations for linearizability and lock-freedom based on this instance and have shown its application to verify the major safety and liveness aspects of a lock-free stack with explicit memory reuse.

In current work, we have successfully applied the approach described here to locally verify linearizability and lock-freedom of the Michael-Scott queue with hazard pointers [11] and of a refined version of the stack, where the abstract free-set is replaced by a further lock-free stack. These proofs are online too [10]. Our recent work also shows that a local verification of Michael's lock-free set algorithm [29] is possible too. Moreover, we currently generalize the decomposition of linearizability to treat more complex linearization points, adapting results from [30]. These improved techniques are applied to further challenging concurrent algorithms.

References

1. Jones, C.B.: Specification and design of (parallel) programs. In: Proceedings of IFIP'83, North-Holland (1983) 321–332
2. Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. ACM Trans. on Prog. Languages and Systems **12**(3) (1990) 463–492
3. Massalin, H., Pu, C.: A lock-free multiprocessor os kernel. Technical Report CUCS-005-91, Columbia University (1991)
4. Moszkowski, B.: Executing Temporal Logic Programs. Cambr. Univ. Press (1986)
5. Burstall, R.M.: Program proving as hand simulation with a little induction. Information processing 74 (1974) 309–312
6. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured specifications and interactive proofs with KIV. In Bibel, W., Schmitt, P., eds.: Automated Deduction—A Basis for Applications. Volume II: Systems and Implementation Techniques. Kluwer Academic Publishers, Dordrecht (1998) 13 – 39
7. Bäumler, S., Schellhorn, G., Tofan, B., Reif, W.: Proving linearizability with temporal logic. Formal Aspects of Computing (FAC) **23**(1) (2011) 91–112

8. Treiber, R.K.: System programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center (1986)
9. Groves, L., Colvin, R.: Trace-based derivation of a scalable lock-free stack algorithm. Formal Aspects of Computing (FAC) **21**(1–2) (2009) 187–223
10. KIV: Presentation of proofs for concurrent algorithms (2011) URL: `http://www.informatik.uni-augsburg.de/swt/projects/lock-free.html`.
11. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**(6) (2004) 491–504
12. Bäumler, S., Balser, M., Nafz, F., Reif, W., Schellhorn, G.: Interactive verification of concurrent systems using symbolic execution. AI Communications **23**((2,3)) (2010) 285–307
13. Schellhorn, G., Tofan, B., Ernst, G., Reif, W.: Interleaved programs and rely-guarantee reasoning with ITL. In: Proc. of TIME, to appear. IEEE, CPS (2011)
14. de Roever, W.P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (2001)
15. Tofan, B., Bäumler, S., Schellhorn, G., Reif, W.: Temporal logic verification of lock-freedom. In: In Proc. of MPC 2010. Springer LNCS 6120 (2010) 377–396
16. Coleman, J.W., Jones, C.B.: A structural proof of the soundness of rely/guarantee rules. J. Logic and Computation **17** (August 2007) 807–841
17. Prensa Nieto, L.: The rely-guarantee method in Isabelle /HOL. In Degano, P., ed.: ESOP'03. Volume 2618 of LNCS., Springer (2003) 348–362
18. Feng, X., Ferreira, R., Z.Shao: On the relationship between concurrent separation logic and ag reasoning. In: Proc. ESOP. Springer LNCS 4421 (2007) 173 – 188
19. Vafeiadis, V., Parkinson, M.J.: A marriage of rely/guarantee and separation logic. In: CONCUR. Volume 4703 of Springer LNCS. (2007) 256–271
20. Fu, M., Li, Y., Feng, X., Shao, Z., Zhang, Y.: Reasoning about optimistic concurrency using a program logic for history. In: CONCUR. (2010) 388–402
21. Tofan, B., Schellhorn, G., Reif, W.: Formal verification of a lock-free stack with hazard pointers. In: Proc. ICTAC, Springer LNCS 6916 (2011)
22. Miller, A., Donaldson, A., Calder, M.: Symmetry in temporal logic model checking. ACM Comput. Surv. **38** (September 2006)
23. Vechev, M., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: Proceedings of the 16th International SPIN Workshop on Model Checking Software, Springer-Verlag (2009) 261–278
24. Cerný, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., R.Alur: Model checking of linearizability of concurrent list implementations. In: CAV. Volume 4144 of LNCS. (2010) 465–479
25. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Thread quantification for concurrent shape analysis. In: CAV'08, Springer (2008)
26. Vafeiadis, V.: Automatically proving linearisability. In: CAV. Volume LNCS 6174., Springer (2010) 450–464
27. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: FORTE 2004. Volume 3235 of LNCS. (2004) 97–114
28. Gotsman, A., Cook, B., Parkinson, M., Vafeiadis, V.: Proving that nonblocking algorithms don't block. In: POPL, ACM (2009) 16–28
29. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: In Proc. of SPAA '02. SPAA '02, ACM (2002) 73–82
30. Derrick, J., Schellhorn, G., Wehrheim, H.: Verifying linearisabilty with potential linearisation points. In: Proc. Formal Methods (FM), Springer LNCS 6664 (2011)

Author Index