Algorithmic Verification of Procedural Programs in the Presence of Code Variability

SIAVASH SOLEIMANIFARD

Doctoral Thesis
Stockholm, Sweden, 2014

KTH Royal Institute of Technology
TRITA-CSC-A 2014:11 School of Computer Science and Communications
ISSN-1653-5723 Department of Theoretical Computer Science
ISRN-KTH/CSC/A–14/11-SE SE-100 44 Stockholm
ISBN 978-91-7595-211-6 SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framläggs till offentlig granskning för avläggande av teknologie doktorsexamen i datalogi fredag den 19 sep 2014 klockan 14.00 i sal F3 Lindstedtsvägen 26 Kungl Tekniska högskolan, Stockholm.

Tryck: Universitetsservice US AB

Abstract

This thesis addresses the formal verification of temporal properties of procedural programs that are dynamically or statically configured by replacing, adapting, or adding new components. Dealing with such *variable* programs is challenging because a part of the program is either not available at verification time or changes frequently. Still, such static and dynamic variability is used in a variety of modern software systems and design paradigms, e.g., software product lines. In this thesis, we develop a generic framework and a fully automated tool support for the verification of such programs. We also show that our technique can be used for efficient verification of existing sets of products constructed from product lines. Our framework is built on top of a previously developed framework for compositional verification of control-flow safety properties of procedural programs that abstracts away all program data. The work in this study is presented through three papers.

The first paper introduces ProMoVer, a fully automated tool for procedure modular verification of control-flow temporal safety properties. Procedure modular verification is a natural instantiation of compositional verification at the procedure level. ProMoVer is described and evaluated on several real-life case studies. It is equipped with a number of features, such as automatic specification extraction and the support for several specification formalisms, to facilitate easy usage. Moreover, it provides a proof storage and reuse mechanism to minimize the need for the computationally expensive verification subtasks.

The second paper discusses the verification of software product lines (SPL). In SPL engineering, products are generated from a set of well-defined commonalities and variabilities. The products of an SPL can be described by means of hierarchical variability models specifying the commonalities and variabilities between the individual products. The number of products generated from a hierarchical model is exponential in the size of the model. Therefore, scalable verification of SPLs is only possible if compositional techniques are applied that allow reusing of the intermediate verification results. In this thesis, we propose a hierarchical variability model for modeling product families, provide a process for extracting such models from existing products, and adapt our compositional verification principle and tool support for the verification of SPLs modeled by this hierarchical model.

The third paper presents a generalization of the original framework to capture program data, still keeping its complexity within practical limits. Thus, it brings the capabilities of the framework to a whole new level. To exemplify its use, we instantiate our framework for compositional verification at three levels of data abstraction of real-life programs: full data abstraction, Boolean data as the only datatype, and heap pointers as the only datatype. We also adapt our toolset to provide support for compositional verification of the latter and evaluate the tool on a real-life case study.

Contents

Acknowledgments

It is my great pleasure to write the following lines and express my sincere gratitude towards those who have supported me to reach this end.

Firstly, I am deeply indebted to my supervisor, Dilian Gurov, for his helps and supports during these years. He constantly helped me by answering my questions, and gently criticizing my ideas. His ingenious comments has always resulted in a breakthrough in my work. Every short discussion with Dilian has been insightful.

Secondly, I am thankful to all members of the theoretical computer science department at KTH. Thanks for making such a nice work environment.

I am thankful to Karl Meinke for reading the drafts of this thesis and for his valuable comments. Your comments truly helped me improve this thesis. Thanks to Musard Balliu, Hamed Nemati, Lukáš Poláček, and Oliver Schwarz for reading and commenting on some parts of this thesis. I believe your comments enhanced the presentation of my work.

I would like to express my appreciation to Pedro de Carvalho Gomes, Emma Enström, Benjamin Greschbach, Jana Götze, Sangxia Huang, Gunnar Kreitz, Lukáš Poláček, Oliver Schwarz, Marc Vinyals, Mladen Mikša, Hamed Nemati, and Vahaid Mosavat for the interesting discussions and meaningful chats during many lunches and coffee breaks ("fika"s) we had together. I am also grateful to Pedro de Carvalho Gomes, Andreas Lundblad, Fei Niu, Karl Palmskog, Muddassar Azam Sindhu, and Björn Terelius for promptly answering my questions.

Last but not least, a sincere gratitude to my wife, Mojgan. My appreciations for her patience and forbearance while I was working days and nights on this work. Thanks for understanding, commenting, listening, and being there.

Part I

Introduction and Summary

Chapter 1

Introduction

Today's society is becoming increasingly dependent on software systems. Such systems are so integrated into our daily lives that their failures often result in the loss of our time, money or in some cases even lives. As a result, there is an increasing demand for high quality, correct and bug-free software. To meet the need, software producers spend a significant amount of money and effort to assure the quality of their products, for example by testing. Still, we occasionally experience annoying and sometimes troubling software failures. The key to avoid such disturbing incidents from happening is to develop more reliable, efficient, easy-to-use, and scalable techniques to assure the reliability of software products.

One process to assure software reliability is *formal verification*, where the correctness of a system is checked with respect to its precisely defined desired properties, often called a *specification*. Most frameworks for formal verification consist of three basic elements: a mathematical model to abstractly capture the behavior of the system, a formalism to express the specification, and a methodology to prove the correctness of the model with respect to the specification. Over the past decades there has been a wealth of advances in the development of tools and techniques for formal verification. Among these, *model checking* [37, 31, 26, 18] has shown to be practically successful [33, 13, 24, 21].

Model checking was originally proposed by E. Clarke and A. Emerson [48, 28], and independently by J. Queille and J. Sifakis [89]. The original method models programs as finite-state machines and checks the correctness of their specifications by exhaustively exploring the state-space of the models. The specifications are often expressed in a temporal logic [85], that is a logic to specify how propositions change over time without introducing time explicitly [102], such as "whenever p is true, then at some later point q must be true". The main advantage of this method is that the checking is mechanical and thus requires minimal manual work. However, it is computationally expensive, because the checking is exhaustive[1].

In this thesis, we employ model checking techniques to develop a relatively

[1] In sections 2.3 and 2.2 we explain model checking and temporal logic in detail.

cheap, flexible, scalable, and easy-to-use verification framework for a specific class of systems, termed *variable systems*.

1.1 Verification of Variable Systems

The focus of the present work is on the verification of procedural programs in the presence of code *variability*. The term variability is used here for software systems that are dynamically or statically configured by replacing, adapting, or adding new code. Thus it can be exemplified by the following four typical situations or scenarios:

(i) *code evolution*, where programs are configured by replacing or adapting existing code,

(ii) *mobile code*, where applications are configured by adding new code to provide more functionality, e.g., add-ons and extensions,

(iii) *incomplete programs*, where programs are delivered incomplete and are completed later by mobile code, and

(iv) *software product lines*, where multiple implementations of components exist and are composed to produce different products.

Throughout this thesis we use the term *variable systems* to refer to the systems in the presence of code variability and *variable (non-variable) components* to refer to the variable (non-variable) part of such systems.

The verification of procedural programs in the scenarios above is challenging because the code of the variable components is either not available at verification time or changes frequently. Therefore, an ideal verification technique for such systems should (i) *localize* the verification of variable components, and (ii) *relativize* the *global* correctness of the system on the correctness of its variable components. This can be achieved through a compositional verification scheme where system components are specified *locally* and verified independently, while the global correctness is inferred from these local specifications. As a result, such a technique allows an independent evolution of individual components (like the scenarios above), only requiring the re-establishment of their local correctness upon a code change.

The above verification scheme can be exemplified by Pnueli's *assume-guarantee* paradigm [86] in which a program is decomposed into smaller components and these are analyzed independently. By composing the analyses results, it can be determined whether the program satisfies the specification or not[2]. An algorithmic realization of assume-guarantee reasoning is achieved by replacing the local specifications with so called *maximal models* [58]. These are the most general models satisfying the specifications. Thus, if such models exist for the class of models and

[2]The assume-guarantee reasoning scheme is more elaborately explained in Section 2.4.

properties at hand, they can replace the specifications of variable components in the verification of the global properties, which then simply reduces to model checking.

Using maximal models for assume-guarantee reasoning was first proposed by O. Grumberg and D. Long [58] in 1994, and in 2000 generalized for a more expressive logic by O. Kupferman and M. Vardi [71]. Both these works address finite-state systems. Later, D. Gurov, M. Huisman, and C. Sprenger suggested the use of maximal models for verification of infinite-state pushdown systems, namely programs with procedure calls and recursion. Their technique was originally developed for compositional verification of incomplete programs in a framework called *Compositional Verification of Programs with Procedures (*CVPP*)* [60]. As we show in this thesis the CVPP methodology can be used in other variability scenarios as well.

In the CVPP framework, to verify incomplete programs, maximal models are constructed from specifications of the variable components and are used to verify the correctness of the global properties of the whole program by means of model checking. The technique abstracts away all program data. Thus, the models are essentially over-approximated control flow graphs representing the method invocations of the programs. Such a drastic abstraction, while allowing the verification of certain interesting temporal properties [100], significantly reduces the range of properties that can be handled. For instance, properties of sequences of method invocations shown in the left column of Table 1.1 can be verified, however, as a result of the extensive over-approximation, there is a high chance of false-positive results. Furthermore, the framework obviously cannot verify properties that address program data, such as the ones in the right column of Table 1.1.

Properties without data	**Properties with data**
a method that changes sensitive data is only called from within a dedicated authentication method, i.e., unauthorized access is not possible	a method that changes sensitive data can only be called after an authentication method returns `true`, i.e., unauthorized access is not possible
a method that *opens a file* should be called before a method that *closes one*	a method that opens file `f` should be called before a method that closes `f`, and `f` must not be `null` upon a call to the closing method
in a voting system, a call to the method that gets candidate selection has to be finished, before the vote can be confirmed	in a voting system, the candidate vote should be valid before the vote can be confirmed
before the invocation of a method to dump a program state into memory, a serialization method must be called to arrange the state	an object must not be `null` upon a call to its serialization method

Table 1.1: Properties with and without data

The CVPP framework is implemented in the form of a set of stand-alone independent tools and has been shown to be useful for the verification of variable systems [60, 67]. However, from a practical point of view, it has the following **restrictions** that significantly limit its usability.

1. The toolset cannot be used as a one-body full-fledged tool, i.e., the individual tools have to be invoked manually in a specific order by expert users to perform the verification.

2. Abstracting away from all data, the range of properties that the CVPP framework can handle is rather limited, and there is high chance of false-positives.

In addition, the following **principle limitations** restrict the capabilities of the framework.

1. Component specification is performed manually and requires considerable time and knowledge of the system.

2. Maximal model construction and model checking are computationally expensive.

3. Program model extraction from code is practically difficult.

The work in this thesis is developed on top of the CVPP framework. We provide facilities to address all the limitations and restrictions above. We generalize the framework to capture program data, and thus bring its capabilities to a whole new level. We also fully automate its usage and significantly enhance the usability of the framework. In addition we identify more application areas for our framework and evaluate its utility by means of case studies.

1.2 Overview of the Thesis

The work in this thesis is presented through three papers. To prepare the ground for a better understanding of these, we first provide an overview of the CVPP framework and its toolset. We then present the papers in their chronological order. We give now a short introduction of these works, and clarify their main points and achievements in Chapter 5.

Paper I: Procedure-Modular Specification and Verification of Temporal Safety Properties

This paper addresses restriction 1 and all principle limitations of the CVPP framework (mentioned above) through developing ProMoVer: a fully automated tool for the verification of temporal properties of variable Java programs. The tool is a realization of the theory explained in Chapter 3 and uses the CVPP toolset as its verification engine. Inspired by successful Hoare logic tools [52, 15, 41, 75],

ProMoVer performs *procedure-modular verification*, i.e., the correctness of the temporal global property of a program is relativized on the correctness of the specifications of its procedures. With ProMoVer we show that procedure-modular verification can be achieved for temporal properties and argue that for users, writing specifications at the procedure-level is intuitive and convenient.

We address limitation 1 of the CVPP framework by providing support for several specification formalisms. This feature is convenient in particular when writing properties at different levels of abstraction and component granularity. In Chapter 5 and Paper I we define these formalisms and illustrate their usage by examples. To reduce the effort needed to write specifications, ProMoVer provides a library of common application- and platform-specific global properties, and a facility for extracting specifications from a given implementation. The latter extracts a procedure's legal call sequences from its concrete implementation, by means of static analysis. A user thus does not have to write specifications explicitly; it suffices to inspect the extracted specifications and remove superfluous constraints that might hinder possible variability of the code. Specifications can be extracted in all supported formalisms, so a user can choose the formalism that is most appropriate for the problem at hand, or that he or she is most comfortable with.

Finally, ProMoVer addresses limitations 2 and 3 of the CVPP framework by a proof storage and reuse mechanism: only the models and specifications that are affected by a change (either in implementation or in specification) are recomputed and re-verified, all other results are reused.

Paper II: Model Mining and Efficient Verification of Software Product Lines

Software product line engineering [88] is a design paradigm aiming at developing families of software products, also called *product families*, that share a (usually large) set of *commonalities* and differ in their specific configuration of *variabilities*. During product line engineering reusable components are developed, to be later composed to realize the actual product families. In this paper, we introduce a model for product lines that in a hierarchical fashion captures their commonalities and variabilities and show that our compositional verification technique can be used for efficient verification of these systems.

Usually, the verification techniques for software product lines aim at verifying a desired property for *all* products in the product family. The challenge is that the number of products defined by a product line is (in the worst case) exponential in the number of its components. This explosion poses serious problems to ensuring the critical product requirements by static analysis or other formal verification techniques, and can render infeasible the verification of product lines by verifying all products individually. Thus, formal verification techniques will only scale if their complexity is linear in the number of components of product lines rather than linear in the number of their products. This can be achieved by compositional verification techniques that allow relativization of the product properties on properties of their

components.

In this paper, we show how the CVPP compositional verification methodology can be used to verify properties of all products of a family. We introduce a class of hierarchical models for product families that capture, the relevant aspects of commonalities and variabilities of software product lines. We show that such models suit our compositional verification paradigm. We formally define these models and provide a procedure to extract them from a class of product families. Finally we show how the extracted models can be used by ProMoVer to efficiently verify product families and that the verification effort is linear in the size of the model.

Paper III: Algorithmic Verification of Procedural Programs in the Presence of Code Variability

In this work, we address restriction 2 of the CVPP framework by generalizing it to capture program data. Providing support for data can give rise to several non-trivial complications that make the framework impractical; e.g., the program models and specifications may become too detailed and large, maximal model construction may become unmanageably complex, and the program models may become overly specific to one programming language. In this paper, we provide solutions for all these challenges; we capture program data without adding extra complexity to the maximal model construction and by keeping the size and complexity of the specifications and program models within practical limits.

We develop a generic compositional verification framework that can be instantiated to the verification of a procedural language against a selected class of properties. The idea is to define a parametric program model that combines precise representation of a user selected set of instructions with the abstract representation of the remaining ones. This combination allows us to define a uniform model that can be tuned for the verification of the class of properties of interest. The precise representation of user selected instructions allows us to verify temporal properties addressing those specific instructions, while the abstract representation of the remaining instructions prevents the specifications from becoming overly verbose, and allows us to capture program data without adding extra complexity to the maximal model construction.

To demonstrate its use, we formally define three instantiations of our generic framework, each of which representing a level of data abstraction of real-life programming languages.

- The first instantiation abstracts away all program data. The result of this instantiation is consistent with the CVPP framework, thus showing that the generic framework is a proper generalization of the original one.

- The second instantiation is to programs with Boolean as the only datatype, also known as Boolean programs [11]. This instantiation exemplifies the use of our framework for data from finite domains.

- Finally, the third instantiation is to programs with pointers as the only datatype, here called *pointer programs (PoP)* [96, 91]. This instantiation shows how our framework can be used for the verification of procedural programs with data from infinite domains.

1.3 Contributions

The principle contributions of this thesis are the following.

- A generalization of the CVPP framework that allows to capture program data and thus significantly increases the range of properties that can be verified as compared to its predecessor, CVPP. Meanwhile, it does not add extra complexity to the maximal model construction and property specification.
- Instantiations of the generic framework for data-less, Boolean and pointer programs that result in compositional verification techniques for these languages.
- Implementation of ProMoVer: a fully automated tool for compositional verification with the CVPP framework. The tool takes as input a Java program annotated with global and local specifications, and automatically verifies it. ProMoVer is accessible via a web-interface [97] for public usage and has been evaluated on several real-life case studies.
- Enhancement of the usability by developing a specification extractor, a proof storage and reuse mechanism, and support for several specification languages. These provide practical solutions for the principle limitations of the CVPP framework described above. The specification extractor generates candidate specifications from the implementation of components, thus significantly reducing the effort required for their specification. The proof storage and reuse mechanism has considerably improved the efficiency of verification by reducing the need for maximal model construction and model extraction.
- Evaluation and identification of application areas that would particularly benefit from the CVPP paradigm. As a promising application area, we show how the CVPP framework and ProMoVer can be adapted to the verification of software product families. A novel hierarchical variability model is introduced, the CVPP verification principle is adapted and its soundness is proved, and ProMoVer's annotation language and verification procedure are extended.

1.4 Publications

Our generic compositional verification framework is presented in the following papers.

- Siavash Soleimanifard, Dilian Gurov.
  Algorithmic verification of procedural programs in the presence of code variability,
  To appear in International Symposium on Formal Aspects of Component Software (FACS '14).

  extended version available as:

- Siavash Soleimanifard, Dilian Gurov.
  Algorithmic verification of procedural programs in the presence of code variability,
  `http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-128950` Technical report, KTH, 2013.

The work on ProMoVer and its usability enhancements resulted in the following papers.

- Siavash Soleimanifard, Dilian Gurov, and Marieke Huisman.
  Procedure-modular verification of control flow safety properties.
  In Workshop on Formal Techniques for Java Programs (FTfJP '10), ACM Digital Library `http://doi.acm.org/10.1145/1924520.1924525`, 2010.

- Siavash Soleimanifard, Dilian Gurov, and Marieke Huisman.
  ProMoVer: Modular verification of temporal safety properties.
  In Software Engineering and Formal Methods (SEFM '11), volume 7041 of Lecture Notes in Computer Science, pages 366–381. Springer, 2011.

- Siavash Soleimanifard, Dilian Gurov, and Marieke Huisman.
  Procedure-modular specification and verification of temporal safety properties.
  Software & Systems Modeling, 2013, DOI: 10.1007/s10270-013-0321-0.

The work on the verification of product families resulted in the following papers.

- Ina Schaefer, Dilian Gurov, and Siavash Soleimanifard.
  Compositional algorithmic verification of software product lines.
  In Formal Methods for Components and Objects (FMCO '10), volume 6957 of Lecture Notes in Computer Science, pages 184–203. Springer, 2011.

- Siavash Soleimanifard, Dilian Gurov, Bjarte M. Østvold, and Minko Markov.
  Model mining and efficient verification of software product lines.
  Submitted.

The following paper was also published during my PhD studies but not used in this thesis.

- Therese Bohlin, Bengt Jonsson, Siavash Soleimanifard.
  Inferring compact models of communication protocol entities.
  In Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA '10), Part I. volume 6415 of Lecture Notes in Computer Science, pages 666–680. Springer, 2010.

1.5 Thesis Structure

This thesis is divided into two parts. Part I provides the required background and a summary of the papers. This part is structured as follows. The next chapter explains the fundamental notions of models and logics. Chapter 3 describes the CVPP compositional verification framework, and Chapter 4 gives an overview of the toolset developed for the CVPP framework prior to the present work. In chapter 5 we provide an overview of the results presented in the papers, and Chapter 6 presents our concluding discussion. Part II includes the three papers that are enclosed to this thesis.

Chapter 2

Preliminaries

In this chapter, we briefly review the main notions and definitions that are necessary in the subsequent chapters of this thesis. Here, we provide definitions for Kripke structures, transition systems and pushdown automata. Then we define Linear-Time Temporal Logic (LTL) and modal μ-calculus, together with security automata as formalisms for expressing temporal properties. At the end, we present some background information on compositional verification of temporal properties.

2.1 Models

To model the *structure* of programs and their executions, here also called *behavior*, we use *Kripke structures*, *transition systems*, and *pushdown automata*.

2.1.1 Kripke Structures

Kripke structures are finite-state transition graphs introduced by S. Kripke [70]. These can be used to capture structure and behavior of systems.

Definition 2.1 (**Kripke Structure**). *A* Kripke Structure *is a tuple* $\mathcal{M} = (S, S_0, AP, \rightarrow, L)$ *where* S *is a finite set of* states*,* S_0 *is a set of* initial states*,* AP *is a set of* atomic propositions*,* $\rightarrow \subseteq S \times S$ *is a* transition relation*, and* $L : S \rightarrow 2^{AP}$ *is a* labeling function.

Example 2.1. The following is an example of a Kripke structure.

$$M \stackrel{\text{def}}{=} (S, S_0, AP, \rightarrow, L)$$

where,

- $S \stackrel{\text{def}}{=} \{s_0, s_1\}$
- $S_0 \stackrel{\text{def}}{=} \{s_0\}$

- $AP \stackrel{\text{def}}{=} \{p, r, q\}$
- $\rightarrow \stackrel{\text{def}}{=} \{(s_0, s_1), (s_1, s_0), (s_1, s_1)\}$
- $L \stackrel{\text{def}}{=} \{(s_0, \{p, r\}), (s_1, \{q\})\}$.

Figure 2.1 shows a graphical representation of the above Kripke structure.

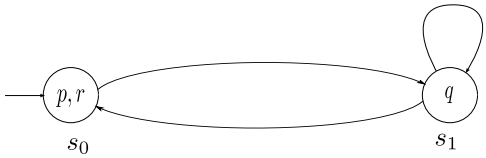


Figure 2.1: Kripke structure M

A *path* of structure $\mathcal{M}$ is an infinite sequence of states $\pi = s_0 s_1 s_2 s_3 \ldots$ such that $s_i \rightarrow s_{i+1}$ for all $i \geq 0$. Given such a path, we denote by $\pi(i)$ the i-th element s_i of π, and we denote by π^i the i-th suffix $s_i s_{i+1} s_{i+2} s_{i+3}$ of π.

Example 2.2. Consider Kripke structure M from Example 2.1. A path of M is: $\pi = s_0 s_1 s_2 \ldots$ where $\pi(1) = s_1$ and $\pi^1 = s_1 s_2 \ldots$.

2.1.2 Transition Systems

In the literature various definitions for transition systems can be found. However, we here define labeled transition systems which are transition graphs with labels (actions) on edges (transitions). Labeled transition systems are suitable for modeling the behavior of systems that have interactions with their environment; each action is interpreted as an input received from the environment.

Definition 2.2 (**Labeled Transition System**). *A* labeled transition system *is a tuple* $\mathcal{T} = (S, Act, \rightarrow)$ *where* S *is a set of* states*,* Act *is a set of* actions*, and* $\rightarrow \subseteq S \times Act \times S$ *is a* transition relation*.*

Example 2.3. The following is an example of a transition system.

$$T \stackrel{\text{def}}{=} (S, Act, \rightarrow)$$

where,

- $S \stackrel{\text{def}}{=} \{s_0, s_1\}$
- $Act \stackrel{\text{def}}{=} \{a, b\}$
- $\rightarrow \stackrel{\text{def}}{=} \{(s_0, a, s_1), (s_1, b, s_0), (s_1, b, s_1)\}$

Figure 2.2 shows a graphical representation of the above labeled transition system.

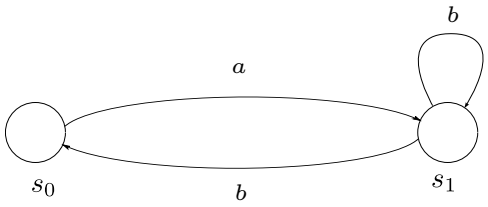


Figure 2.2: Labeled transition system T

2.1.3 Pushdown Automata and Systems

Pushdown automata and pushdown systems are natural formalisms for modeling the behavior of programs with procedure calls and recursion (see e.g., [22, 4] for analysis techniques and [49, 25] for applications). These models are variations of automata that make use of a stack to store data. In the next definitions, we define formally these formalisms.

Definition 2.3 (**Pushdown Automata**). *A* Pushdown Automaton (PDA) *is a tuple* $\mathcal{P} = (Q, \Sigma, \Gamma, \Delta, Q', \bot)$ *where* Q *is a finite set of* control states, Σ *a finite* input alphabet, Γ *finite* stack alphabet, $Q' \subseteq Q$ *are the* starting states, $\bot \in \Gamma$ *is the* initial stack symbol, *and* $\Delta \subseteq \langle Q \times \Gamma \rangle \times \Sigma \times \langle Q \times \Gamma^* \rangle$ *is a set of labeled* productions *(or rewrite rules) of the shape* $\langle q_1, A \rangle \overset{a}{\hookrightarrow} \langle q_2, \gamma \rangle$.

Definition 2.4 (**Configuration**). *A* configuration *of a PDA is a pair* $\langle q, \gamma \rangle \in Q \times \Gamma^*$. *The set of PDA configurations* $Q' \times \{\bot\}$ *is called* initial configurations.

The productions of a PDA induce a labeled transition relation of configurations as the least relation which contains the initial configuration and is closed under the prefix rewrite rule: $\langle q_1, A \cdot \gamma' \rangle \xrightarrow{a} \langle q_2, \gamma \cdot \gamma' \rangle$ whenever $\langle q_1, A \rangle \overset{a}{\hookrightarrow} \langle q_2, \gamma \rangle \in \Delta$.

Example 2.4. The following is a pushdown automaton that recognizes the language $L = \{a^n b^n \mid n \geq 1\}$.

- $Q \overset{\text{def}}{=} \{q_a, q_b\}$
- $\Sigma \overset{\text{def}}{=} \{a, b\}$
- $\Gamma \overset{\text{def}}{=} \{S, A\}$
- $\Delta \overset{\text{def}}{=} \left\{ \begin{array}{l} \langle q_a, S \rangle \overset{a}{\hookrightarrow} \langle q_a, A \rangle, \\ \langle q_a, A \rangle \overset{a}{\hookrightarrow} \langle q_a, AA \rangle, \\ \langle q_a, A \rangle \overset{b}{\hookrightarrow} \langle q_b, \epsilon \rangle, \\ \langle q_b, A \rangle \overset{b}{\hookrightarrow} \langle q_b, \epsilon \rangle \end{array} \right\}$
- $Q' \overset{\text{def}}{=} q_a$

- $\bot \stackrel{\text{def}}{=} S$

Figure 2.3 shows a graphical representation of pushdown automaton P. In the figure, the productions are depicted by transitions, e.g., production $\langle q_a, A\rangle \overset{b}{\hookrightarrow} \langle q_b, \varepsilon\rangle$ is depicted by a transition from state q_a to state q_b with the label $b;A/\varepsilon$.

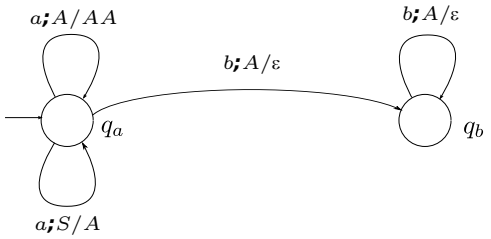


Figure 2.3: Pushdown automaton P

Definition 2.5 (**Pushdown Systems**). *A* pushdown system *(PDS) is a pushdown automaton without labels on transitions.*

2.2 Temporal Logics

Temporal logic formulas are used to express how propositions are changed over time without introducing time explicitly. Time is discrete and extends infinitely into the future. Formulas of temporal logic express *when* (w.r.t., time) a logical proposition is true.

Various temporal logics have been introduced and used in formal verification. Here, we define *Linear-Time Temporal Logic (LTL)* introduced by A. Pnueli [85] and *Modal μ-calculus* introduced by D. Kozen [69]. We also define *Security Automata* proposed by F. B. Schneider [93] as a formalism for expressing *safety properties* [72], that are properties stipulating that no "bad thing" happens during any path.

2.2.1 Linear-Time Temporal Logic

Linear-Time Temporal Logic (LTL) [85] is one of the commonly used temporal logics. It is defined as follows.

Definition 2.6 (**Syntax of LTL**). *The formulas of* LTL *are inductively defined by the abstract syntax below:*

$$\Phi ::= p \mid \neg\Phi \mid \Phi \wedge \Psi \mid \Phi \vee \Psi \mid \mathtt{X}\,\Phi \mid \mathtt{G}\,\Phi \mid \mathtt{F}\,\Phi \mid \Phi\,\mathtt{U}\,\Psi$$

where p ranges over a given set of atomic propositions AP.

In the above syntax, the symbols $\wedge$, $\vee$, and $\neg$ are the usual Boolean connectives. The symbols $\mathtt{X}$, $\mathtt{G}$, $\mathtt{F}$, and $\mathtt{U}$ are temporal operators, $\mathtt{X}\,\Phi$ is true if Φ holds in the next state, $\mathtt{G}\,\Phi$ if Φ holds in all future states including the current state, $\mathtt{F}\,\Phi$ if Φ is true in some future state, and $\Phi\,\mathtt{U}\,\Psi$ if Φ remains true until a state is reached where Ψ is true.

Next, we give the precise semantics of LTL.

Definition 2.7 (**Semantics of LTL**). *Let $\mathcal{M} = (S, S_0, AP, \rightarrow, L)$ be a model, and let π be a path of $\mathcal{M}$.*

$$
\begin{array}{lcl}
\pi \models^{\mathcal{M}} p & \stackrel{def}{\Longleftrightarrow} & p \in L(\pi(0)) \\
\pi \models^{\mathcal{M}} \neg\Phi & \stackrel{def}{\Longleftrightarrow} & \pi \not\models \mathcal{M}\Phi \\
\pi \models^{\mathcal{M}} \Phi \wedge \Psi & \stackrel{def}{\Longleftrightarrow} & \pi \models^{\mathcal{M}} \Phi \textit{ and } \pi \models^{\mathcal{M}} \Psi \\
\pi \models^{\mathcal{M}} \Phi \vee \Psi & \stackrel{def}{\Longleftrightarrow} & \pi \models^{\mathcal{M}} \Phi \textit{ or } \pi \models^{\mathcal{M}} \Psi \\
\pi \models^{\mathcal{M}} \mathtt{X}\Phi & \stackrel{def}{\Longleftrightarrow} & \pi^1 \models^{\mathcal{M}} \Phi \\
\pi \models^{\mathcal{M}} \mathtt{G}\Phi & \stackrel{def}{\Longleftrightarrow} & \forall i \geq 0.\, \pi^i \models^{\mathcal{M}} \Phi \\
\pi \models^{\mathcal{M}} \mathtt{F}\Phi & \stackrel{def}{\Longleftrightarrow} & \exists i \geq 0.\, \pi^i \models^{\mathcal{M}} \Phi \\
\pi \models^{\mathcal{M}} \Phi\,\mathtt{U}\,\Psi & \stackrel{def}{\Longleftrightarrow} & \exists i \geq 0.\, (\pi^i \models^{\mathcal{M}} \Psi \wedge \forall j < i.\, \pi^j \models^{\mathcal{M}} \Phi)
\end{array}
$$

A few other LTL operators have been defined in the literature. In this thesis, we will use the *weak until* operator $\mathtt{W}$. We say $\Phi\,\mathtt{W}\,\Psi$ is true on π if Φ remains true until a state is reached where Ψ is true or Φ remains true in all states of π. The formal semantics of operator $\mathtt{W}$ can be given through operators $\mathtt{U}$ and $\mathtt{G}$ as follows.

$$\Phi\,\mathtt{W}\,\Psi = (\Phi\,\mathtt{U}\,\Psi) \vee \mathtt{G}\,\Phi$$

Model $\mathcal{M}$ at state $s \in S$ satisfies formula Φ, denoted $\mathcal{M}, s \models \Phi$, if all paths π of $\mathcal{M}$ starting at s satisfy Φ. As a shorthand for $\mathcal{M}, s_0 \models \Phi$, where $s_0 \in S_0$, we use $\mathcal{M} \models \Phi$, and we say that property Φ holds for model $\mathcal{M}$.

Example 2.5. The property "r and q are never true at the same time" can be expressed by the following LTL formula.

$$\mathtt{G}\ \neg\ (r \wedge q)$$

This property holds for Kripke structure M from Example 2.1.

2.2.2 Modal μ-Calculus

Modal μ-calculus [69] is an extension of Hennessy-Milner logic [64] with the extremal fixed point operators. It is an expressive logic that subsumes most other well-known temporal logics such as CTL and LTL.

Definition 2.8 (**Syntax of Modal μ-Calculus**). *The formulas of the modal μ-calculus are defined by the following abstract syntax [101].*

$$\Phi ::= \mathbf{tt} \mid \mathbf{ff} \mid Z \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid \langle\alpha\rangle\,\Phi \mid [\alpha]\,\Phi \mid \mu Z.\,\Phi \mid \nu Z.\,\Phi$$

where α ranges over a given set of action labels Act and Z ranges over a set of propositional variables Var.

Definition 2.9 (**Semantics of Modal μ-Calculus**). *For a labeled transition system $\mathcal{T} = (\mathcal{S}, Act, \rightarrow)$, let $\mathcal{V} : Var \rightarrow 2^{\mathcal{S}}$ be a valuation. The semantics of the modal μ-calculus for all $s \in \mathcal{S}$ is defined as follows.*

$$\begin{array}{rcl}
\|\mathbf{tt}\|^{\mathcal{T}}_{\mathcal{V}} & \stackrel{def}{=} & S \\
\|\mathbf{ff}\|^{\mathcal{T}}_{\mathcal{V}} & \stackrel{def}{=} & \emptyset \\
\|Z\|^{\mathcal{T}}_{\mathcal{V}} & \stackrel{def}{=} & \mathcal{V}(Z) \\
\|\Phi \vee \Psi\|^{\mathcal{T}}_{\mathcal{V}} & \stackrel{def}{=} & \|\Phi\|^{\mathcal{T}}_{\mathcal{V}} \cup \|\Psi\|^{\mathcal{T}}_{\mathcal{V}} \\
\|\Phi \wedge \Psi\|^{\mathcal{T}}_{\mathcal{V}} & \stackrel{def}{=} & \|\Phi\|^{\mathcal{T}}_{\mathcal{V}} \cap \|\Psi\|^{\mathcal{T}}_{\mathcal{V}} \\
\|\langle\alpha\rangle\, \Phi\|^{\mathcal{T}}_{\mathcal{V}} & \stackrel{def}{=} & \|\langle\alpha\rangle\|^{\mathcal{T}} \left(\|\Phi\|^{\mathcal{T}}_{\mathcal{V}}\right) \\
\|[\alpha]\, \Phi\|^{\mathcal{T}}_{\mathcal{V}} & \stackrel{def}{=} & \|[\alpha]\|^{\mathcal{T}} \left(\|\Phi\|^{\mathcal{T}}_{\mathcal{V}}\right) \\
\|\mu Z.\, \Phi\|^{\mathcal{T}}_{\mathcal{V}} & \stackrel{def}{=} & \bigcap \left\{ S' \subseteq \mathcal{S} \mid S' \supseteq \|\Phi\|^{\mathcal{T}}_{\mathcal{V}[S'/Z]} \right\} \\
\|\nu Z.\, \Phi\|^{\mathcal{T}}_{\mathcal{V}} & \stackrel{def}{=} & \bigcup \left\{ S' \subseteq \mathcal{S} \mid S' \subseteq \|\Phi\|^{\mathcal{T}}_{\mathcal{V}[S'/Z]} \right\}
\end{array}$$

where $\|\langle\alpha\rangle\|^{\mathcal{T}} : 2^{\mathcal{S}} \rightarrow 2^{\mathcal{S}}$, and $\|[\alpha]\|^{\mathcal{T}} : 2^{\mathcal{S}} \rightarrow 2^{\mathcal{S}}$ are defined as follows:

$$\begin{array}{rcl}
\|\langle\alpha\rangle\|^{\mathcal{T}}(S') & \stackrel{def}{=} & \left\{ s \in \mathcal{S} \mid \exists s' \in \mathcal{S}.\, (s \xrightarrow{\alpha} s' \wedge s' \in S') \right\} \\
\|[\alpha]\|^{\mathcal{T}}(S') & \stackrel{def}{=} & \left\{ s \in \mathcal{S} \mid \forall s' \in \mathcal{S}.\, (s \xrightarrow{\alpha} s' \Rightarrow s' \in S') \right\}
\end{array}$$

An alternative, but equivalent, interpretation of extremal fixed points is through approximates. We provide a characterization where *Ord* is the set of ordinals, $\alpha \in Ord$ are ordinals, and $\lambda \in Ord$ is a limit ordinal. Let $(\sigma Z.\Phi)^{\alpha}$ be the α-*approximant* (alternatively α-unfolding) of $\sigma Z.\Phi$ ($\sigma \in \{\mu, \nu\}$) with the following interpretation:

$$\left\|(\mu Z.\, \Phi)^0\right\|^{\mathcal{T}}_{\mathcal{V}} \stackrel{\text{def}}{=} S \qquad \left\|(\nu Z.\, \Phi)^0\right\|^{\mathcal{T}}_{\mathcal{V}} \stackrel{\text{def}}{=} \emptyset$$

$$\left\|(\mu Z.\, \Phi)^{\alpha+1}\right\|^{\mathcal{T}}_{\mathcal{V}} \stackrel{\text{def}}{=} \|\Phi\|^{\mathcal{T}}_{\mathcal{V}[\|(\mu Z.\Phi)^{\alpha}\|^{\mathcal{T}}_{\mathcal{V}}/Z]} \qquad \left\|(\nu Z.\, \Phi)^{\alpha+1}\right\|^{\mathcal{T}}_{\mathcal{V}} \stackrel{\text{def}}{=} \|\Phi\|^{\mathcal{T}}_{\mathcal{V}[\|(\nu Z.\Phi)^{\alpha}\|^{\mathcal{T}}_{\mathcal{V}}/Z]}$$

$$(\mu Z.\Phi)^{\lambda} \stackrel{\text{def}}{=} \bigcup\{\|(\mu Z.\Phi)^{\alpha}\|^{\mathcal{T}}_{\mathcal{V}} \mid \alpha \leq \lambda\} \qquad (\nu Z.\Phi)^{\lambda} \stackrel{\text{def}}{=} \bigcap\{\|(\nu Z.\Phi)^{\alpha}\|^{\mathcal{T}}_{\mathcal{V}} \mid \alpha \leq \lambda\}$$

Example 2.6. The property "exactly after an occurrence of action a, another action a cannot happen" for a system with actions a and b, can be formalized by the following μ-calculus formula.

$$\nu\, Z.\ [b]Z \wedge [a]\left([a]\mathbf{ff} \wedge [b]Z\right)$$

The above formula holds for the labeled transition system of Example 2.3.

2.2.3 Security Automata

The notion of security automata was first introduced as a class of Büchi automata [47] to recognize and specify safety properties [3, 93]. Security automata are similar to non-deterministic finite-state automata but have a different acceptance criterion.

Definition 2.10 (**Security Automata**). *A* security automaton *is a tuple* $\mathcal{A} = (S, S_0, Act, \rightarrow)$ *where* S *is a countable set of* states, $S_0 \subseteq S$ *is a countable set of* initial states, Act *is a countable set of* input symbols *(or actions), and* $\rightarrow \subseteq S \times Act \times S$ *is a* transition relation.

To process a sequence $a_1 a_2 ...$ of input symbols, the *current state* S' of the security automaton starts equal to S_0 and the sequence is read one input symbol at a time. As each input symbol a_i is read, the security automaton changes S' to any state that could be reached from the current state by a_i. If S' is ever the empty set, then the input is rejected; otherwise the input is accepted. Notice that this acceptance criterion means that a security automaton can accept sequences that have infinite length as well as those having finite length.

Example 2.7. The following security automaton expresses property "exactly after an occurrence of action a, another action a cannot happen" for a system with actions a, b (*cf.* the logical representation in Example 2.6).

$$A \stackrel{\text{def}}{=} (S, S_0, I, \rightarrow)$$

where,

- $S \stackrel{\text{def}}{=} \{s_0, s_1\}$
- $S_0 \stackrel{\text{def}}{=} \{s_0\}$
- $Act \stackrel{\text{def}}{=} \{a, b\}$
- $\rightarrow \stackrel{\text{def}}{=} \{(s_0, b, s_0), (s_0, a, s_1), (s_1, b, s_0)\}$

Figure 2.4 shows a graphical representation of the above security automaton.

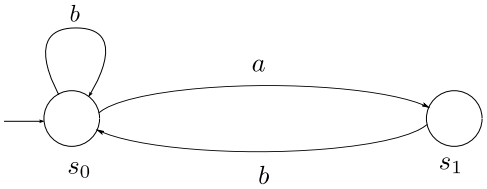


Figure 2.4: Security automaton A

2.3 Model Checking

Model checking is a state-exploration approach which automatically decides if a property is satisfied by a model or not. Usually the property is expressed in a temporal logic and the model is a state machine, such as a Kripke structure. The method was first developed for the verification of finite-state systems [32, 76], and later generalized for some classes of infinite-state systems [22], such as pushdown systems [94, 68, 49, 12] and context-free processes [23].

The main advantage of model checking is that the checking is mechanical and the verification requires minimal manual work. Its main limitation, however, is that the size of the model is often very large, thus its whole state space cannot be explicitly explored in practice. This is the result of the *state-explosion problem*, and has been addressed by several approaches, e.g., *symbolic model checking* that represents the model implicitly by logical formulas [79, 29, 30, 34], *bounded model checking* that unrolls the model for a fixed number of steps to avoid large or infinite unrollings [19], *counterexample guided abstraction refinement (CEGAR)* that begins checking with a coarse (imprecise) abstraction and iteratively refines it until the property is violated both by the abstract model and the actual system [9, 53, 65, 14], *property preserving reduction* techniques such as *partial-order reduction* that reduces the size of the state space of concurrent asynchronous systems by exploiting commutativity of concurrently executed transitions resulting in the same state [83, 55, 103], and *compositional verification* that is explained in Section 2.4 below.

In the present work, we do not develop new model checking algorithms. However, we employ existing finite- and infinite-state model checking algorithms for verifying temporal properties expressed in several formalisms.

2.4 Background on Compositional Verification of Temporal Properties

Compositional verification was introduced to alleviate the state-explosion problem by breaking the verification of the whole system into the verification of its *components*. The main idea is to verify properties of each individual component, and relativize the correctness of the system on these properties, as a result avoiding to compute the state space of the whole system.

Compositional verification has been studied in the context of *compositional model checking* as in [35, 36], *reactive models* as in [1, 2] and *assume-guarantee* reasoning as in [86, 58] (see [84] for a survey). Here, we focus on the latter.

Assume-guarantee reasoning was first proposed in the context of temporal logic verification by A. Pnueli [86]. In Pnueli's system, one works with triples of the form $\langle\Phi\rangle M\langle\Psi\rangle$ which are read as "assuming the environment of M satisfies Φ, in this environment, component M guarantees to satisfy Ψ". Then a typical chain of reasoning would be as follows.

$$\frac{\begin{array}{c}\langle true\rangle M'\langle\Phi\rangle \\ \langle\Phi\rangle M\langle\Psi\rangle\end{array}}{\langle true\rangle M'|M\langle\Psi\rangle}$$

where $M'|M$ represents the composition of M' and M. The above principle states that if M' satisfies Φ in any environment, and M in an environment satisfying Φ, satisfies Ψ then the composition of M' and M will satisfy Ψ.

The obvious advantage of verification by the above approach is that we never have to examine the composite state space of $M'|M$. The downside is that the user must determine an appropriate Φ, which requires knowledge of the behavior of the system.

The automation of the above compositional verification principle has been studied by O. Grumberg and D. Long in [58] for finite-state systems. To this end, they proposed a simulation preorder (denoted $\preceq$) on finite-state structures that has the following important properties.

1. It is preserved by composition, i.e., for models M, M', Q and Q',

$$(M \preceq M') \wedge (Q \preceq Q') \Rightarrow (M \mid Q) \preceq (M' \mid Q')$$

2. It preserves satisfaction of the logical formulas in $\forall$CTL—the universal fragment of *Computation Time Logic (CTL)* [27][1]—, i.e., for models M and Q and formula Φ in $\forall$CTL,

$$(M \preceq Q) \wedge Q \models \Phi \Rightarrow M \models \Phi$$

Furthermore, Grumberg and Long showed that for any formula Φ in $\forall$CTL there exists a so called *maximal model* (denoted by $\mathcal{M}ax(\Phi)$) that is a model simulating any model satisfying Φ. Formally, for model M and formula Φ in $\forall$CTL

$$M \preceq \mathcal{M}ax(\Phi) \iff M \models \Phi \tag{2.1}$$

They also introduced an algorithm for constructing maximal models for formulas in $\forall$CTL. Properties 1 and 2, together with the existence of maximal models (2.1) justify the following sound compositional verification principle.

$$\frac{M' \models \Phi \qquad\qquad \mathcal{M}ax(\Phi) \mid M \models \Psi}{M \mid M' \models \Psi}$$

The soundness of the above rule is established in the following way. $M' \models \Phi$ together with (2.1) infer that $M' \preceq \mathcal{M}ax(\Phi)$. This result together with Property (1) entail $M \mid M' \preceq \mathcal{M}ax(\Phi) \mid M$. Using this, Property (2), and $\mathcal{M}ax(\Phi) \mid M \models \Psi$ we infer that $M \mid M' \models \Psi$.

Intuitively, the above rule reduces checking $M \mid M' \models \Psi$ to the following steps:

[1] Knowing the temporal logic CTL is not necessary here.

1) decomposition of the global property Ψ to local property Φ of component M',

2) the check that M' satisfies Φ,

3) construction of a maximal model $\mathcal{M}ax(\Phi)$ for Φ, and

4) the check that $\mathcal{M}ax(\Phi) \mid M$ satisfies Ψ.

The idea is that the property Φ expresses the behavior of M' at a level of abstraction that is sufficient for verifying the global property Ψ. This results in a model $\mathcal{M}ax(\Phi)$ that is considerably smaller than M' and thus state-explosion problem is alleviated.

In a later work, O. Kupferman and M. Vardi showed that the property 2 holds also for the more expressive logic $\forall$CTL*—the universal fragment of CTL*. They also showed that maximal models exist for this logic and introduced an algorithm for constructing them.

The CVPP framework (explained in Chapter 3) implements the above compositional verification schema for infinite-state pushdown systems, but a weaker logic, namely safety fragment of μ-calculus with the box and greatest fixed point operators. From a practical point of view, unlike the works above, CVPP makes use of an automation of assume-guarantee reasoning for the verification of incomplete procedural programs and not for avoiding the state-explosion problem. In this thesis we show that the CVPP framework is also applicable in other variability scenarios.

Chapter 3

A Verification Framework

In this chapter, we describe the theoretical underpinnings of the framework for *compositional verification of programs with procedures (*CVPP*)*, developed by D. Gurov, M. Huisman and C. Sprenger and presented in [60]. As mentioned in the introduction, CVPP is a compositional verification technique for verifying temporal properties of (pushdown) infinite-state systems in the presence of variability. Intuitively, the CVPP framework relativizes the correctness of the *global properties* of a system on the *local specifications* of its components. The local specifications are provided by the users and their correctness should be checked independently for each individual component. Such a verification scheme decouples the implementations of the components from the global correctness reasoning, thus allows independent evolution of components' code.

CVPP separates program structure from its operational semantics (behavior); It defines separate models and specification languages for program structure and its induced behavior. Program structure is modeled by a control flow graph, which makes the analysis independent from the programming language, while program behavior is an infinite model induced by the structure in a context-free fashion. By this separation both structural and behavioral aspects of programs can be analyzed.

The structural models, termed *flow graphs*, are extracted from the implementation of components or can be constructed from the local specifications. The constructed flow graphs are called *maximal flow graphs*. A maximal flow graph for a specification is a flow graph that simulates exactly those flow graphs satisfying the specification. Thus, it can be used as a representative of all flow graphs satisfying the specification for the purpose of relativizing the correctness of the global properties on the local specifications of components.

The main restriction of the CVPP framework is that all program data is abstracted away and programs are analyzed only for the safety temporal properties of their over-approximated control flow. Although this sounds like a severe restriction, still some useful properties can be expressed at this level of abstraction. These include platform-specific security properties and application-specific properties such

as those listed in the left column of Table 1.1.

In this chapter, we will formally define CVPP's *models*, *flow graphs* and the *specification language*. Then we will briefly explain the *maximal flow graph construction*. And at the end, we formally state the *compositional verification principle* which CVPP is based on. The material of this chapter are adapted from [60] where the details and proofs can be found.

3.1 Model and Logic

We begin by formally defining the general notion of model and the logic for specifying properties. Structural and behavioral models and logics will be defined as instantiations of the general notions below.

Definition 3.1 (**Model**). *A* model *is a (Kripke) structure* $\mathcal{M} = (S, L, \rightarrow, A, \lambda)$ *where* S *is a set of states,* L *a set of labels,* $\rightarrow \subseteq S \times L \times S$ *a labeled transition relation,* A *a set of atomic propositions, and* $\lambda : S \rightarrow 2^A$ *a valuation, assigning to each state* s *the set of atomic propositions that hold in* s*. An* initialized model $\mathcal{S}$ *is a pair* $(\mathcal{M}, E)$ *with* $\mathcal{M}$ *a model and* $E \subseteq S$ *a set of initial states.*

The reachable part of an initialized model $\mathcal{S} = (\mathcal{M}, E)$ is defined by $\mathcal{R}(\mathcal{S}) = (\mathcal{M}', E)$, where $\mathcal{M}'$ is obtained from $\mathcal{M}$ by deleting all states and transitions not reachable from any entry state in E.

Next, we define a simulation preorder on models. Intuitively, initialized model S_1 simulates S_2 if it is "more general" than S_2.

Definition 3.2 (**Simulation**). *A simulation is a binary relation* R *on* S *such that whenever* $(s,t) \in R$ *then* $\lambda(s) = \lambda(t)$*, and whenever* $s \xrightarrow{a} s'$ *then there is some* $t' \in S$ *such that* $t \xrightarrow{a} t'$ *and* $(s', t') \in R$*. We say that* t *simulates* s*, written* $s \leqslant t$*, if there is a simulation* R *such that* $(s,t) \in R$*.*

Simulation on two models $\mathcal{M}_1$ and $\mathcal{M}_2$ is defined as simulation on their disjoint union $\mathcal{M}_1 \uplus \mathcal{M}_2$. The transitions of $\mathcal{M}_1 \uplus \mathcal{M}_2$ are defined by $in_i(s) \xrightarrow{a} in_i(s')$ if $s \xrightarrow{a} s'$ in $\mathcal{M}_i$ and its valuation by $\lambda(in_i(S)) = \lambda_i(S)$, where in_i (for $i \in \{1,2\}$) injects S_i into $S_1 \uplus S_2$. Simulation is extended to initialized models $(\mathcal{M}_1, E_1)$ by defining $(\mathcal{M}_1, E_1) \leqslant (\mathcal{M}_2, E_2)$ if there is a simulation R on $\mathcal{M}_1 \uplus \mathcal{M}_2$ such that for each $s \in E_1$ there is some $t \in E_2$ with $(in_1(s), in_2(t)) \in R$. Initialized model $\mathcal{S}_1$ is simulation equivalent to $\mathcal{S}_2$, written $\mathcal{S}_1 \simeq \mathcal{S}_2$ if $\mathcal{S}_1 \leqslant \mathcal{S}_2$ and $\mathcal{S}_2 \leqslant \mathcal{S}_1$. Disjoint union is extended to initialized models (by $(\mathcal{M}_1, E_1) \uplus (\mathcal{M}_2, E_2) = (\mathcal{M}_1 \uplus \mathcal{M}_2, E_1 \uplus E_2)$). The following theorem shows that simulation is preserved by disjoint union.

Theorem 3.3 (**Composition**). *If* $\mathcal{S}_1 \leqslant \mathcal{T}_1$ *and* $\mathcal{S}_2 \leqslant \mathcal{T}_2$ *then* $\mathcal{S}_1 \uplus \mathcal{S}_2 \leqslant \mathcal{T}_1 \uplus \mathcal{T}_2$*.*

As property specification language CVPP uses the safety fragment of the modal μ-calculus [101] with the box and ν operators, which is parametrized on a set of atomic propositions A and labels L. The syntax and semantics of the modal μ-calculus is explained in Section 2.2.2.

Definition 3.4 (**Simulation Logic**). *The formulas of* simulation logic *are inductively defined by:*

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

where $p \in A$, $a \in L$ and X ranges over propositional variables.

Satisfaction on states $(\mathcal{M}_b, s) \models \phi$ is defined in the standard fashion [69]. For instance, formula $[a]\phi$ holds of state s in model $\mathcal{M}_b$ if ϕ holds in all states accessible from s via an edge labeled a. An initialized model $(\mathcal{M}_b, E_b)$ satisfies a formula ϕ, denoted $(\mathcal{M}_b, E_b) \models \phi$, if all its initial configurations E_b satisfy ϕ. The constant formulas *true* (denoted `tt`) and *false* (`ff`) are definable. For convenience, we use $p \Rightarrow \phi$ to abbreviate $\neg p \vee \phi$.

Alternative to simulation logic defined above, one can use modal equation systems where instead of the ν operator of μ-calculus, equations are used to formulate recursion. The construction of maximal flow graphs is based on modal equation systems, hence we define basic simulation logic and modal equation systems to provide the basis for maximal flow graph construction.

Definition 3.5 (**Basic Simulation Logic**). *The formulas of* basic simulation logic *are inductively defined by:*

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

where $p \in A$, $a \in L$ and X ranges over a set of countably infinite propositional variables.

Definition 3.6 (**Modal Equation System**). *A* modal equation system $\Sigma = \{X_i = \Phi_i | i \in I\}$ *over L and A is a finite set of equations such that the variables X_i are pairwise distinct and each Φ_i is a formula of basic simulation logic over L and A. The set of variables occurring in Σ is partitioned into the set of* bound *variables, defined by* $\mathtt{bv}(\Sigma) = \{X_i | i \in I\}$, *and the set of* free *variables* $\mathtt{fv}(\Sigma)$.

Using the definitions of basic simulation logic and modal equation systems, the formulas of simulation logic are defined by $\Phi[\Sigma]$ over L and A, where Φ is a formula of basic simulation logic and Σ is a modal equation system. These formulas are equivalent in expressiveness to the modal μ-calculus without diamond modalities and least fixed points as defined in Definition 3.4. The translation of simulation logic to modal equation systems defined in Definition 3.6 is based on Bekič's principle described in [17, 8]. The translation in the other direction is straightforward and is done simply by replacing each fixed point by an equation.

3.2 Program Model

The program model is either representing the finite-state model of the program's control flow structure, or the infinite-state behavior induced from it.

3.2.1 Program Structure

A program's structure is modeled as a control flow graph, abstracting away all its data. *Flow graphs* are essentially a collection of *method graphs*, one for each procedure of the program. Let *Meth* be a countably infinite set of method names. A method graph is an instance of the general notion of initialized model (Definition 3.1).

Definition 3.7 (**Method graph**). *A* method graph *for method $m \in Meth$ over a set $M \subseteq Meth$ of method names is an initialized model $(\mathcal{M}_m, E_m)$ where $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$ is a finite model and $E_m \subseteq V_m$ is a non-empty set of* entry points *of m. V_m is the set of* control nodes *of m, $L_m = M \cup \{\varepsilon\}$, $A_m = \{m, r\}$, and $\lambda_m : V_m \rightarrow 2^{A_m}$ so that $m \in \lambda_m(v)$ for all $v \in V_m$ (i.e., each node is tagged with its method name). The nodes $v \in V_m$ with $r \in \lambda_m(v)$ are* return points.

Notice that methods can have multiple entry points. Flow graphs that are extracted from program source have single entry points, but the maximal flow graphs that are generated for compositional verification can have multiple entry points.

Every flow graph is equipped with an *interface* which is defined as follows.

Definition 3.8 (**Flow graph interface**). *A* Flow graph interface *is a pair $I = (I^+, I^-)$, where $I^+, I^- \subseteq Meth$ are finite sets of names of* provided *and* required *methods, respectively. The* composition *of two interfaces $I_1 = (I^+, I^-)$ and $I_2 = (I^+, I^-)$ is defined by $I_1 \bigcup I_2 = (I_1^+ \cup I_2^+, I_1^- \cup I_2^-)$. An interface $I = (I^+, I^-)$ is closed if $I^- \subseteq I^+$ and otherwise it is open.*

A flow graph is *closed* if its interface is closed, and it is *open* otherwise.

Flow graph *composition* is defined as the disjoint union $\uplus$ of their method graphs.

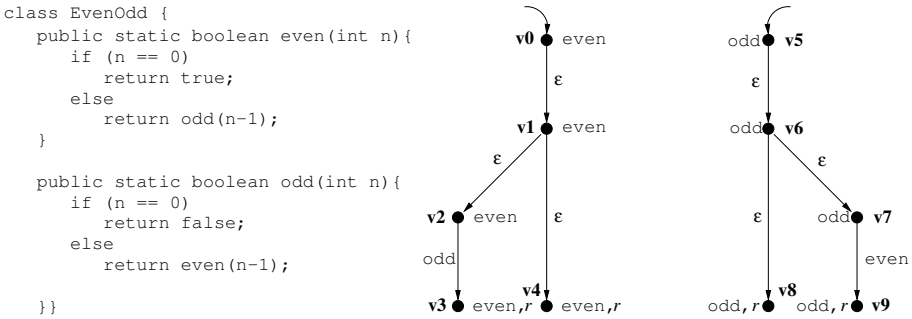


Figure 3.1: Flow graph of EvenOdd

Example 3.1. Figure 3.1 shows a Java program and its flow graph. It consists of two method graphs, `even` and `odd`. Entry nodes are depicted as usual by incoming

edges without source. The interface of this flow graph is $I^+ = \{\texttt{even}, \texttt{odd}\}, I^- = \{\texttt{even}, \texttt{odd}\}$, thus the flow graph is closed.

The instantiation of simulation logic to express properties of flow graphs is called *structural simulation logic*, where the set of atomic propositions consists of method names and `r`, and labels are method names and ε.

Example 3.2. Consider the closed flow graph in Example 3.1. The property "if the program execution starts in method `even`, the first call is to method `odd` and after returning no other method can be called" is formalized by the following structural formula.

$$\texttt{even} \Rightarrow \nu X.[\texttt{even}]\texttt{ff} \wedge [\varepsilon]X \wedge [\texttt{odd}]\phi$$

where ϕ is:

$$\nu Y.[\texttt{even}]\texttt{ff} \wedge [\texttt{odd}]\texttt{ff} \wedge [\varepsilon]Y$$

3.2.2 Program Behavior

In this section, we define the flow graph behavior for closed and open flow graphs. The definition of behavior for open flow graphs is a generalization of the one for closed flow graphs. We think it is more understandable if flow graph behavior is defined in an incremental fashion. Thus, first, we define the behavior of closed flow graphs and thereafter generalize the definition for open ones.

Behavior of Closed Flow Graphs

In CVPP the *behavior* of a flow graph is defined as a labeled transition system (LTS). In the behavior transition label τ is used for internal transfer of control, m_1 *call* m_2 for the invocation of method m_2 by method m_1 when method m_2 is provided by the program, m_2 *ret* m_1 for the corresponding return from the call.

Definition 3.9 (Behavior). *Let $\mathcal{G} = (\mathcal{M}, E) : (I^+, I^-)$ be a flow graph such that $\mathcal{M} = (V, L, \rightarrow, A, \lambda)$. The behavior of $\mathcal{G}$ is defined as initialized model $b(\mathcal{G}) = (\mathcal{M}_b, E_b)$, where $\mathcal{M}_b = (S_b, L_b, \rightarrow_b, A_b, \lambda_b)$, such that $S_b = V \times V^*$, i.e., states (or configurations) are pairs of control points v and stacks σ, $L_b = \{m_1\ k\ m_2 \mid k \in \{call, ret\}\ \} \cup \{\tau\}$, $A_b = A$, $\lambda_b((v, \sigma)) = \lambda(v)$, and $\rightarrow_b \subseteq S_b \times L_b \times S_b$ is defined by the rules:*

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

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

Example 3.3. Consider the flow graph from Example 3.1. One example run through its (branching, infinite-state) behavior, from an initial to a final configuration, is:

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

Flow graph behavior can alternatively be expressed as *pushdown automata* (PDA), *pushdown systems* (PDS), or *context-free processes*. This can be exploited by using pushdown automata/systems model checking for verifying behavioral properties.

The instantiation of simulation logic to express properties of flow graph behavior is called *behavioral simulation logic*, where atomic propositions are method names and `r`, and labels are behavioral labels.

Example 3.4. Consider the closed flow graph in Example 3.3. The property "if the execution starts in method `even`, the first call is not to method `even` itself" is formalized by the following behavioral formula.

$$\texttt{even} \Rightarrow \nu X.\ [\texttt{even call even}]\texttt{ff} \wedge [\tau]X$$

The instantiation of simulation for flow graph behavior is called behavioral simulation, denoted by $\leqslant_b$, and defined as follows.

$$\mathcal{G}_1 \leqslant_b \mathcal{G}_2 \iff b(\mathcal{G}_1) \leqslant b(\mathcal{G}_2).$$

A result that is essential for the correction of the CVPP compositional verification framework is that if two flow graphs are related by structural simulation, then their behaviors are related by behavioral simulation.

Theorem 3.10. *For flow graphs* $\mathcal{G}_1$ *and* $\mathcal{G}_2$*, if* $\mathcal{G}_1 \leqslant \mathcal{G}_2$ *then* $\mathcal{G}_1 \leqslant_b \mathcal{G}_2$*.*

Behavior of Open Flow Graphs

As mentioned above, a flow graph is called *open* if its interface is open (i.e., it requires external methods). Therefore, we can generalize Definition 3.9 for open flow graphs where some required methods are external by adding the following labels to set L_b.

- *call*! is used when an internal method calls an external one,
- *call*? is used when an external method calls an internal one,
- *ret*! is used when an internal method returns to an external one,

- *ret*? is used when an external method returns to an internal one.

Extending the set of labels with the ones above, the rewrite rules in Definition 3.9 should also be extended by the following rules.

$$\text{[call!]} \quad (v_1,\sigma)\xrightarrow{m_1\ \texttt{call!}\ m_2}(m_2,v_1'\cdot\sigma) \quad \textit{if} \quad m_1\in I^+, m_2\notin I^+, v_1\xrightarrow{m_2}_{m_1} v_1', v_1\models\neg r$$

$$\text{[call?]} \quad (m_2,\sigma)\xrightarrow{m_3\ \texttt{call?}\ m_1}(v,m_3\cdot m_2\cdot\sigma) \quad \textit{if} \quad m_1\in I^+, m_2,m_3\notin I^+, v\models m_1, v\in E$$

$$\text{[ret!]} \quad (v,m_3\cdot m_2\cdot\sigma)\xrightarrow{m_1\ \texttt{ret!}\ m_3}(m_2,\sigma) \quad \textit{if} \quad m_1\in I^+, m_2,m_3\notin I^+, v\models m_1\wedge r$$

$$\text{[ret?]} \quad (m_2,v_1\cdot\sigma)\xrightarrow{m_2\ \texttt{ret?}\ m_1}(v_1,\sigma) \quad \textit{if} \quad m_1\in I^+, m_2\notin I^+, v_1\models m_1$$

Example 3.5. Consider method `even` from Example 3.1 but this time as an open flow graph with interface $I^+ = \{\texttt{even}\}, I^- = \{\texttt{odd}\}$. One example run through its (infinite-state) behavior, from an initial to a final configuration, is the following:

$(v_0,\bot)\xrightarrow{\tau}(v_1,\varepsilon)\xrightarrow{\tau}(v_2,\varepsilon)\xrightarrow{\texttt{even call! odd}}(\texttt{odd},v_3)\xrightarrow{\texttt{odd call? even}}(v_0,\texttt{odd}\cdot\texttt{odd}\cdot v_3)\xrightarrow{\tau}(v_1,\texttt{odd}\cdot\texttt{odd}\cdot v_3)\xrightarrow{\tau}(v_4,\texttt{odd}\cdot\texttt{odd}\cdot v_3)\xrightarrow{\texttt{even ret! odd}}(\texttt{odd},v_3)\xrightarrow{\texttt{odd ret? even}}(v_3,\varepsilon)$.

Example 3.6. Consider the open flow graph in Example 3.5. The property "the first call is to external method `odd` and no other method can be called after returning from the call" is formalized by the following open behavioral formula.

$$\nu X.[\texttt{even call even}]\texttt{ff}\wedge[\tau]X\wedge[\texttt{even call! odd}][\texttt{odd ret? even}]\phi$$

where ϕ is:

$$\nu Y.[\texttt{even call even}]\texttt{ff}\wedge[\texttt{even call! odd}]\texttt{ff}\wedge[\tau]Y$$

Next we introduce maximal flow graphs and briefly explain their construction process.

3.3 Maximal Flow Graph

Intuitively, a maximal model for a property ϕ is a model that satisfies ϕ and simulates all models satisfying it. The use of maximal models was first proposed by O. Grumberg and D. Long in [58]. As explained in Section 2.4, they proposed a maximal model construction for a subset of *Computational Tree Logic (CTL)*, termed ∀CTL. In this section, however, we describe a maximal model construction for models of Definition 3.1 and logical formulas defined in Definition 3.4 which was presented by D. Gurov, M. Huisman, and C. Sprenger in [60]. Since the program structure and behavior are instances of models, the maximal model construction can be used for both structures and behaviors.

For structural properties, the existence and uniqueness of maximal models is proved in [60]. However, in general, these maximal models are not legal flow graphs. This problem can be solved for a fixed interface by a so-called *characteristic formula* that is a formula constructed from the interface to constraint the constructed maximal models to legal flow graphs. Basically the characteristic formula for interface I precisely defines all flow graphs with I. Concretely, if σ_{I_m} is the characteristic formula of component m and σ_m is the property of m, we construct the maximal model of $\sigma_{I_m} \wedge \sigma_m$ which is a legal flow graph with interface I_m and simulates all flow graphs satisfying the property σ_m. We define the characteristic formula and the algorithm for constructing maximal flow graphs in this section.

For behavioral properties however, there is no such a way to characterize all models that are loyal to the definition of flow graph behavior. The intuitive reason is that the behavioral logic cannot express context-free properties. As a solution, Gurov et al. provide an algorithm that translates a behavioral property into a set of structural ones. The resulting properties can then be used for maximal flow graph construction.

In this section, we first explain the construction of maximal models, then define the characteristic formula to be used for constructing maximal flow graphs. The translation of behavioral properties into a set of structural ones is explained in [59].

3.3.1 Maximal Model Construction

To start with the construction of maximal models we define two auxiliary functions θ and χ which form a *Galois connection* between finite models and formulas in simulation logic. Function χ translates a *finite* model into a formula, and function θ translates formulas into (finite) models. Both functions are defined on formulas in a so-called *simulation normal form (SNF)*. Here, we only define the auxiliary functions χ and θ, and SNF. We show that every simulation logic formula has an SNF representation and provide an algorithm to convert formulas to their SNF.

Definition 3.11 (χ)**.** *Function χ maps a finite initialized model $(\mathcal{M}, E)$ into its* characteristic formula $\chi(\mathcal{M}, E) = \phi_E[\Sigma_\mathcal{M}]$*, where $\phi_E = \bigvee_{s\in E} X_s$ and $\Sigma_\mathcal{M}$ is defined by an equation below.*

$$X_s = \bigwedge_{a\in L} [a] \bigvee_{s \xrightarrow{a} t} X_t \wedge \bigwedge_{p\in\lambda(s)} p \wedge \bigwedge_{q\notin\lambda(s)} \neg q$$

where $s \in S$, and $\phi_E = \bigvee_{s\in E} X_s$.

Example 3.7. Consider the model $\mathcal{S}$ shown in Figure 3.2. Its characteristic formula is $\chi(\mathcal{S}) = (X_{s_1} \vee X_{s_2})[\Sigma]$, where

$$\Sigma = \left[\begin{array}{lcl} X_{s_1} & = & [a]\mathtt{ff} \wedge [\varepsilon]X_{s_2} \wedge p \wedge q \\ X_{s_2} & = & [a](X_{s_1} \vee X_{s_3}) \wedge [\varepsilon]\mathtt{ff} \wedge p \wedge \neg q \\ X_{s_3} & = & [a]X_{s_1} \wedge [\varepsilon]X_{s_2} \wedge \neg p \wedge \neg q \end{array}\right].$$

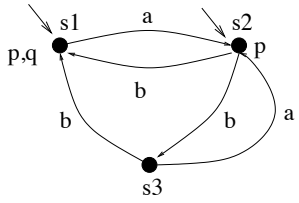


Figure 3.2: Specification example

The next result shows that function χ precisely translates an initialized model to a formula. This is a variation of an earlier result by Larsen [73].

Theorem 3.12. *Let $\mathcal{S}_1, \mathcal{S}_2$ be two models and suppose $\mathcal{S}_2$ is finite. Then $\mathcal{S}_1 \leqslant \mathcal{S}_2$ if and only if $\mathcal{S}_1 \models \mathcal{X}(\mathcal{S}_2)$.*

Definition 3.13 (**Simulation normal form**). *A formula $\phi[\Sigma]$ of simulation logic over L and A is in* simulation normal form (SNF) *if ϕ has the form $\bigvee \chi$ for some finite set $\chi \subseteq \mathtt{bv}(\Sigma)$ and all equations of Σ have the following* state normal form

$$X = \bigwedge_{a \in L} [a] \bigvee \mathcal{Y}_{X,a} \wedge \bigwedge_{p \in B_X} p \wedge \bigwedge_{q \notin A - B_X} \neg q$$

where each $\mathcal{Y}_{X,a} \subseteq \mathtt{bv}(\Sigma)$ is a finite set of variables and $B_X \subseteq A$ is a set of atomic propositions.

Theorem 3.14. *Every formula of simulation logic has an equivalent one in SNF.*

Gurov et al. propose an algorithm that iteratively translates simulation logic formulas into their equivalent ones in SNF [60]. Intuitively, the algorithm works as follows. Let us use $\mathit{Labels}(X)$ and $\mathit{Atoms}(X)$ to refer to the set of labels and atoms of the defining equation for X, respectively. For a given set of atoms A, labels L, and a formula $\phi[\Sigma]$, in each iteration, the algorithm augments each equation of Σ by conjoining its missing labels as $\bigwedge_{l \in L \wedge l \notin \mathit{Labels}(X)} [l]\mathtt{tt}$, and missing atoms as $\bigwedge_{a \in A \wedge a \notin \mathit{Atoms}(X)} (a \vee \neg a)$, and then transforms the resulting formula to SNF by introducing new equations for disjunctions of formulas not guarded by any box and $\mathtt{tt}$. Thus, (in the worst case) the size of the resulting equation system is exponential in the size of L and A.

Having this algorithm, Gurov et al. define a function (defined below) that translates a formula in SNF into an initialized model that graphically represents the formula. Thus, maximal models can be constructed.

Definition 3.15 (θ). *Function θ translates a formula in SNF into a (finite) initialized model. From formula $(\bigvee \mathcal{X})[\Sigma]$ over L and A in SNF we derive the specification*

$$\theta((\bigvee \mathcal{X})[\Sigma]) = ((S, L, \rightarrow, A, \lambda), E)$$

where $S = \mathtt{bv}(\Sigma)$, $E = \mathcal{X}$ *and the equation for* X *induces transitions* $\{X \xrightarrow{a} Y \mid Y \in \mathcal{Y}_{X,a}\}$ *and truth assignment* $\lambda(X) = B_X$.

Lemma 3.16. χ *and* θ *are each mutually inverse up to equivalence, that is,*

1. $\theta(\chi(\mathcal{S})) \cong \mathcal{S}$ *(*$\cong$ *is isomorphism*[1]*) for finite* $\mathcal{S}$*, and*
2. $\chi(\theta(\phi)) \equiv_\alpha \phi$ *(*$\equiv_\alpha$ *is* α*-convertibility) for* ϕ *in SNF.*

Finally, we are ready to relate simulation logic to simulation.

Theorem 3.17 (Maximal model theorem). *For* ϕ *in SNF, we have* $\mathcal{S} \leqslant \theta(\phi)$ *if and only if* $\mathcal{S} \models \phi$.

Thus, the model $\theta(\phi)$ is a maximal model for ϕ, in the sense that $\theta(\phi)$ is a model that satisfies ϕ and simulates all models satisfying it.

Consequences. We mention a few consequences of Theorems 3.12 and 3.17. Let $(\mathsf{S}, \leqslant)$ be the preorder of (isomorphism classes of) *finite* models over given L and A ordered by simulation and let $(\mathsf{L}, \models)$ be the preorder of formulas of simulation logic over L and A ordered by the logical consequence relation.

Corollary 3.18. χ *and* θ *are monotone.*

Simulation preserves logical properties:

Corollary 3.19. *For all initialized models* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *we have* $\mathcal{S}_1 \leqslant \mathcal{S}_2$ *and* $\mathcal{S}_2 \models \phi$ *imply* $\mathcal{S}_1 \models \phi$.

The pair (χ, θ) of maps forms a Galois connection between the preorders $(\mathsf{L}, \models)$ and $(\mathsf{S}, \leqslant)$.

Corollary 3.20. *For finite initialized model* $\mathcal{S}$ *and all formulas* ϕ*, we have* $\mathcal{S} \leqslant \theta(\phi)$ *if and only if* $\chi(\mathcal{S}) \models \phi$.

3.3.2 Maximal Flow Graph Construction

As explained above, to construct a maximal flow graph from a structural property ϕ and interface I, the characteristic formula for interface I has to be used. This formula precisely defines flow graphs with interface I. Here, we formally define these formulas.

The characteristic formula of component m is defined by the following construction.

$$\sigma_{I_m} = \bigvee\nolimits_{m \in I^+} \nu X \cdot P_m \wedge [I^-, \epsilon]X$$

$$\text{where, } P_m = m \wedge \bigwedge\nolimits_{m' \in I^+ \setminus \{m\}} \neg m'$$

[1] Here, isomorphism means a bijection of states and transitions, but labels have to be equal.

Alternatively, the characteristic formula can be defined by using modal equation systems as follows.

$$\begin{array}{lcl} \sigma_{I_m} & = & \bigvee_{m \in I^+} X_m \\ \Sigma_I & = & \{X_m = [I^-, \varepsilon] X_m \wedge p_m \mid m \in I^+\} \\ p_m & = & m \wedge \bigwedge \{\neg m' \mid m' \in I^+, m' \neq m\} \end{array}$$

With the help of characteristic formulas we obtain a variant of Theorem 3.17 for flow graphs.

Theorem 3.21. *Let I be an interface. For any initialized model $\mathcal{S} = (\mathcal{M}, E)$ over labels $L = I^- \cup \{\varepsilon\}$ and atomic propositions $A = I^+ \cup \{r\}$ we have*

$$\mathcal{S} \models \sigma_I \text{ if and only if } \mathcal{R}(\mathcal{S}) : I$$

where $\mathcal{R}(\mathcal{S})$ defines the reachable part of the initialized model $\mathcal{S}$ as defined in Section 3.1.

Thus, a maximal model that is constructed for the conjunction of a formula ϕ and characteristic formula for interface I, is a maximal flow graph for I and ϕ.

3.4 Compositional Verification

For models and formulas as defined in Definitions 3.1 and 3.4 maximal models exist and are unique up to isomorphism. Therefore for this choice of model and logic the following sound and complete compositional verification principle can be obtained.

> To show that $\mathcal{M}_1 \uplus \mathcal{M}_2 \models \phi$, it suffices to show $\mathcal{M}_1 \models \psi$ where ψ is a local property of $\mathcal{M}_1$, and $\mathcal{M}ax(\psi, I) \uplus \mathcal{M}_2 \models \phi$ where $\mathcal{M}ax(\psi, I)$ is the maximal flow graph constructed from property ψ and interface I.

Formally, the above compositional verification is instantiated for structural properties as shown below.

$$(\mathsf{struct} - \mathsf{comp}) \ \frac{\mathcal{G}_1 \models_s \psi \qquad \mathcal{M}ax(\psi, I) \uplus \mathcal{G}_2 \models_s \phi}{\mathcal{G}_1 \uplus \mathcal{G}_2 \models_s \phi} \tag{3.1}$$

where $\mathcal{G}_1$ and $\mathcal{G}_2$ are flow graphs and ψ and ϕ are structural properties. For behavioral properties, however, the principle is as follows.

$$(\mathsf{beh} - \mathsf{comp}) \ \frac{\mathcal{G}_1 \models_b \psi \qquad \biguplus_{\chi \in \Pi(\psi)} \mathcal{M}ax(\chi, I) \uplus \mathcal{G}_2 \models_b \phi}{\mathcal{G}_1 \uplus \mathcal{G}_2 \models_b \phi} \tag{3.2}$$

where $\Pi(\psi)$ translates behavioral property ψ to a set of structural formulas, and ϕ is a behavioral formula.

As proved in [60], the above rules are sound and complete when interfaces describe all provided and required methods.

In addition to the above principles, a "mixed" rule is proposed in [60], where local structural properties are combined with global behavioral ones.

The proof rules presented above are sufficiently flexible to be used for reasoning about a combination of concrete components (i.e., given through their implementation) and abstract ones (i.e., given through their specification), both at the structural and the behavioral levels [67]. In Chapter 4 we show how these proof rules can be instantiated for several verification scenarios such as verification of *incomplete programs*.

3.5 Private Method Abstraction

Often private methods are viewed as a means of implementation of public ones. Thus for the purpose of verification they should often be seen as implementation details that local specifications should abstract from. To support reasoning on the level of public methods, Gurov et al. proposed a transformation of flow graphs that inlines private methods graphs into the public ones: Given a flow graph $\mathcal{G}$, every call to a private method m in $\mathcal{G}$ is replaced by m's method graph (see [60] for details). The recursive calls to private methods are not inlined, but create loops in the resulting graph. This transformation is sound (i.e., over-approximates the behavior) in general, and complete for last-call recursive programs and τ-insensitive properties.

The private method abstraction has been used for the verification of several case studies. The experiments show that such an abstraction results in significantly smaller specifications and less expensive maximal flow graph construction.

Chapter 4

A Verification Toolset

The CVPP compositional verification framework, explained in Chapter 3, is implemented as a set of stand-alone tools. As we shall see, these tools can be invoked in different orders to be used in various verification scenarios. In this chapter, we will provide an overview of this toolset prior to the work in this thesis, and explain its application in several verification scenarios. The content of this chapter is adapted from [67].

4.1 Overview of the Toolset

Figure 4.1 shows an overview of the CVPP toolset. In the figure, the rounded boxes represent data formats, squared boxes denote tool components, and the dashed boxes are used to show the external tools or data formats. In this section we describe the data formats and the tools of the toolset, while in the next section we describe the verification scenarios they could be used for.

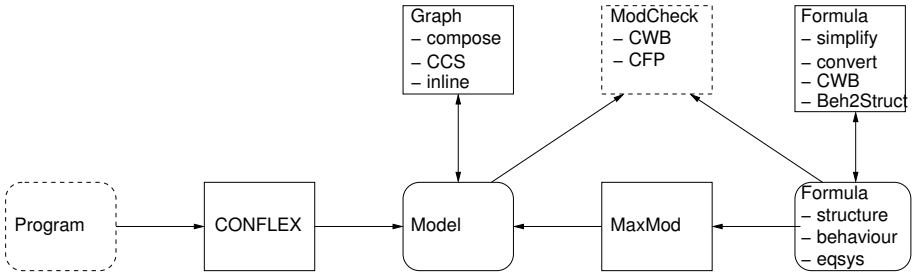


Figure 4.1: Overview of the CVPP toolset

4.1.1 Data Formats

The CVPP toolset includes the following data formats.

- *Model:* the representation of the program in the CVPP framework. Programs are modeled by flow graphs that are either extracted from the implementation of the program or constructed through a maximal flow graph construction. The textual representation of the flow graph of the program in Example 3.1 is as follows.

```
node v0 meth(even) entry
node v1 meth(even)
node v2 meth(even)
node v3 meth(even) ret
node v4 meth(even) ret
node v5 meth(odd) entry
node v6 meth(odd)
node v7 meth(odd)
node v8 meth(odd) ret
node v9 meth(odd) ret
edge v0 v1 eps
edge v1 v2 eps
edge v1 v4 eps
edge v2 v3 odd
edge v5 v6 eps
edge v6 v7 eps
edge v6 v8 eps
edge v7 v9 even
```

  In the textual representation, `node` defines a node of the flow graph, followed by a list of atomic propositions that hold in that node. Atomic propositions `entry` and `ret` specify entry and return nodes, respectively, `meth` is to distinguish method names from atomic propositions `entry` and `ret`. Also `edge` defines a transition through starting node, target node, and the transition label, respectively. The labels are either a method name or `eps` which serves as the textual representation of ε.

- *Formula:* the property representation. Formulas can specify structural and behavioral simulation logic, and modal equation systems. As an example, the textual representations of the properties in Examples 3.2 and 3.4 are shown below.

  Structural formula in Example 3.2:

```
meth(even) => nu X1. (([even]ff) /\ ([eps]X1) /\ [odd]
```

```
(nu X2. (([even]ff) /\ ([odd]ff) /\ ([eps]X2))))
```

Behavioral formula in Example 3.4:

```
meth(even) =>  nu X1. (([even call even]ff) /\ ([tau]X1))
```

- *Interface:* the representation of the flow graph interfaces. The interfaces are pairs of sets of *required* and *provided* methods. They are used by almost all tools in the CVPP toolset and thus not shown in Figure 4.1. The textual representation of the interface of the flow graph shown in Figure 3.1 is as follows.

```
provided even,odd
required even,odd
```

4.1.2 Tools

- ConFlEx: the tool for extracting flow graphs from program code. Currently, ConFlEx extracts flow graphs of Java programs from Java bytecode. It is based on Sawja—a static analysis library for Java programs [66]. To extract the flow graph of Java program P, ConFlEx disregards all stack operations and data in P's bytecode and returns its over-approximated control flow graph[1] [45, 46, 6].

- Graph: a collection of algorithms to perform transformations on the program model. By this tool, program models are composed together, CCS format of program models are generated, and method graphs of private methods are inlined into the public ones. The tool for the latter is called Graph Inliner and explained in Section 3.5. Graph is also used to generate the behavior of flow graphs represented as CFP.

- Formula: the tool for translation, transformation and simplification of the property representation. Using this tool, behavioral formulas are translated to sets of structural formulas, and simulation logic formulas are simplified and translated to CWB μ-calculus formulas. The behavioral to structural translation is the main functionality of the tool where a behavioral formula is translated into a set of structural ones to be used in the maximal flow graph construction (see Chapter 3).

[1] At the moment, CVPP only supports Java programs because the ConFlEx tool only extracts flow graphs of Java programs. But the whole machinery can be used to prove the correctness of any procedural language if a flow graph extractor of the language is provided.

- MAXIMAL MODEL: the tool to construct maximal flow graphs. This tool constructs a maximal flow graph from a structural formula and an interface using the algorithm explained in Section 3.3. It accepts as input structural properties specified in modal equation systems. Hence, for using this tool, the formula has to be transformed to a modal equation system by FORMULA.

- *Model Checkers*: a collection of external model checkers for the local and global model checking tasks. CFP [54] is used for model checking context-free process representations of behaviors against μ-calculus formulas. On the other hand, *Edinburgh Concurrency Workbench* (CWB) [39] is used for the local model checking of μ-calculus formulas on CCS representation of flow graphs.

4.2 Typical Verification Scenarios

The components of the CVPP toolset can be used in different verification scenarios, of which three are explained in this section. In Paper I, we describe PROMOVER which is a tool for executing one of the most useful verification scenarios (namely procedure-modular verification) with the CVPP toolset.

4.2.1 Verification of Incomplete Programs

The CVPP framework and consequently its toolset are designed for the compositional verification of incomplete programs, these are programs where some components are available by their code (concrete components) and some are available only by their specification (abstract components). This typically happens in mobile code or dynamic systems when the code of some components is not available or stable at the verification time.

The verification of a global property of an incomplete program is relativized on the implementation of the concrete components and specification of the abstract ones. Thus the verification rule 3.2 can be applied as follows.

Local Check: for the abstract components, when the code becomes available, check that the implementation matches the specifications,

Global Check: construct maximal flow graphs from the specifications of abstract components and check that the composition of these together with the flow graphs of the concrete components entails the global property.

The concrete tasks by using the CVPP toolset are as follows.

Local Check: once the implementation of an abstract component is available, check that the implementation of the component matches its specification by the non-compositional verification described in Section 4.2.3.

Global Check:

(a) extract a flow graph from the code of the concrete components by ConFlEx tool, and use Graph Inliner to generate the publicly visible flow graph,

(b) for each of the abstract components, construct a maximal flow graph from its specification by using Maximal Model,

(c) compose the maximal flow graphs of the abstract components and the extracted flow graph of the concrete components by using Graph,

(d) check that the composition result satisfies the global property using non-compositional verification described in Section 4.2.3 below.

4.2.2 Modular Verification

The goal of modular verification is to verify each module (component) of the system, independently and relativize the correctness of its global properties on the basis of the correctness of the modules. Modular verification with CVPP is an instantiation of the verification of incomplete programs explained above. In this scenario the local checks are performed as in the non-compositional verification scenario, and then the global checks are accomplished as if all the components are abstract. This eliminates task (a) and simplifies task (c) of the *Global Check* of the incomplete programs scenario, resulting in the following steps.

Local Check: use non-compositional verification described in Section 4.2.3 to check that the implementation of each component matches its specification,

Global Check:

(a) for each component, construct a maximal flow graph from its specification by using Maximal Model,

(b) compose the maximal models of the components by using Graph,

(c) check that the composition result satisfies the global property using non-compositional verification described in Section 4.2.3 below.

4.2.3 Non-Compositional Verification

Both verification of incomplete programs and modular verification give rise to some non-compositional verification tasks. CVPP can be used for these tasks through the steps below.

(a) extract a flow graph from the code of the concrete components by ConFlEx, and use Graph Inliner to generate the publicly visible flow graph,

if the property is structural perform b and c, otherwise d.

(b) convert the flow graph to a CCS term by using GRAPH,

(c) match the flow graph interface with the implementation and model check the CCS term against the local property by CWB,

(d) cast the flow graph as a context-free term using GRAPH, and model check it against the global property using CFP.

This scenario can sometimes be used for efficient model checking of behavioral properties. Such properties are normally checked on infinite-state behavioral models such as context-free processes generated from flow graphs. However, sometimes it is more efficient to translate these to structural ones and check the results on the original (finite-state) flow graphs. This can be done by following the steps below.

(a) translate the behavioral property to a set of structural ones,

(b) extract a flow graph from the code of the concrete components by CONFLEX, and use GRAPH INLINER to generate the publicly visible flow graph,

(c) convert the flow graph to a CCS term by using GRAPH,

(d) model check the CCS term against the set of structural properties resulting from the first step by CWB.

Chapter 5

Summary of the Papers

This thesis includes three papers. As explained in Chapter 1, the papers are centered around the CVPP framework. They address CVPP's restrictions and limitations described in Section 1.1 and show how the framework can be adapted for the verification of product families. Paper I introduces ProMoVer as a tool for automated verification with the CVPP framework. This paper provides pragmatic solutions to restriction 1 and all principle limitations of the CVPP framework discussed in Section 1.1. Paper II shows that the verification of software product families could benefit from the CVPP verification methodology. It presents a novel technique for the verification of temporal properties of product families, and shows that our verification effort is linear in the number of artifacts forming the products, rather than exponential. Finally, Paper III presents our generic framework for compositional verification that is capable of capturing program data. By this, we lift restriction 2 of the CVPP framework. Our generic framework is essentially a recipe for developing compositional verification techniques for checking temporal safety properties of procedural programs in the presence of code variability. In this chapter, we provide an overview of these works, while the presentation of the technical details is delegated to the attached papers.

5.1 ProMoVer

As explained in the previous chapter, the CVPP toolset consists of a number of stand-alone tools that can be used to verify temporal safety properties of Java programs. These tools cannot directly interact with each other. They need to be invoked by a skillful user in a specific order to provide correct verification results. Further, in some cases, the returned results from a tool have to be manually adjusted in order to be used by other parts of the toolset. We, therefore, designed and implemented ProMoVer [98, 99, 100, 97] as a tool to support automated compositional verification with CVPP.

Compositional verification with the CVPP framework and its toolset can be

achieved at different levels of granularity; components can be packages, classes or procedures. We believe that the procedure level is convenient to use, because it is the level of abstraction at which testers, developers and engineers think. We therefore provide this level of granularity in ProMoVer.

Compositional verification at the procedure level, also known as *procedure-modular verification*, has been successfully used by tools for Hoare-style verification, where specifications would describe the local effect of invoking a procedure [51, 81, 15] (see [62] for a survey). ProMoVer however, uses and verifies temporal logic specifications, that are better suited for capturing the interaction of a procedure with the environment, such as the allowed sequences of procedure invocations. With ProMoVer we show that procedure-modular verification is also appropriate for control flow safety properties: for each procedure the local property specifies its legal call sequences, while the system's global property specifies the allowed interactions of the system as a whole. Thus, temporal specifications provide a meaningful abstraction for procedures.

ProMoVer demonstrates that procedure-modular verification of temporal safety properties can be automated completely by using annotated programs as a single input. From the users perspective, the input to the tool is a Java program annotated with global and method-local properties written in one of the supported formalism (mentioned in Section 5.1.1). For example consider the Java program in Figure 5.1. The program consists of two methods, `even` and `odd`, and it is annotated with global and local properties and interfaces (we provide intuitive description of the local and global properties in Section 5.1.1). ProMoVer takes such annotated programs as input and automatically invokes a number of tools from the CVPP toolset to perform the individual local and global correctness checks. Using private method abstraction of the CVPP framework (explained in Section 3.5), ProMoVer only requires the public procedures to be annotated; the private ones are being considered merely as an implementation means.

5.1.1 Specification Languages

Temporal properties can be expressed in various formalisms, such as automata-based or process-algebraic notations, as well as in temporal logics such as LTL [101]. As already mentioned, the CVPP toolset internally uses the safety fragment of the modal μ-calculus [69] as a property specification language. However, to facilitate the specification of global and local properties, ProMoVer accepts properties written in various temporal formalisms.

ProMoVer accepts *local properties* written (a) as a variant of security automata [93] here termed *safety-automata*, (b) in the *safety fragment of LTL* which includes the `G`, `X` and `W` operators, (c) in *structural* simulation logic, or (d) in a restricted version of behavioral simulation logic here we call *Caret logic* (see Section 2.2 for definitions). In Paper I, we explain the syntax and semantics of safety-automata and safety LTL. Structural simulation logic is defined in Section 3.1. We here intuitively explain the Caret logic.

```
/**
 * @global_ltl_prop: even -> X ((even && !entry) W odd)
 */
public class EvenOdd {
   /** @local_interface: required odd
    *  @local_caret_prop:
    *   nu X1. (([even call even]ff) /\ ([tau]X1) /\
    *     [even caret odd] nu X2.
    *       (([even call even]ff) /\
    *        ([even caret odd]ff) /\ ([tau]X2))
    */
   public boolean even(int n) {
      if (n == 0)
        return true;
      else
        return odd(n-1);
   }

   /** @local_interface: required even
    *  @local_sa_prop:
    *         node s0 odd, entry
    *         node s1 odd, entry
    *         node s2 odd, entry, r
    *         edge s0 s0 tau
    *         edge s0 s1 odd caret even
    *         edge s0 s2 odd caret even
    *         edge s1 s1 tau
    *         edge s1 s2 tau
    */
   public boolean odd(int n) {
      if (n == 0)
        return false;
      else
        return even(n-1);
   }
}
```

Figure 5.1: A simple annotated Java program

Caret logic is a restricted version of the behavioral simulation logic for open flow graphs where all calls to external methods are followed by an immediate return. The formulas of this logic express properties of a generalization of the behavior of closed flow graphs (Definition 3.9) to open ones. Thus, Caret behavior is defined as a generalization of Definition 3.9 with the following additional label (to the set of

labels L_b) for calls to external methods and their immediate corresponding return.

$$[\text{caret}] \quad (v_1,\sigma)\xrightarrow{m_1 \texttt{ caret } m_2}(v_1',\sigma) \quad \textit{if} \quad \begin{array}{l} m_1 \in I^+, m_2 \notin I^+, \\ v_1 \xrightarrow[m_1]{m_2} v_1', v_1 \models \neg r \end{array}$$

By using Caret logic and behavior, the intermediate behavior between external method invocations and their corresponding returns is ignored. This treatment of method calls is inspired by the temporal logic CaRet defined for LTL by R. Alur and others in [5], and is convenient for specifying local behavior of flow graphs. The resulting restricted properties turn out to be adequate for specifying local properties of individual procedures (without self-calls), but are in general too inexpressive for higher levels of component granularity, and in particular for specifying global properties.

For example, the local property of method `even` in Figure 5.1 is written in Caret logic. It expresses that "the first call is to external method `odd` and no other method can be called after returning from the call" (*cf.* the property in Example 3.6). The local property of method `odd` has an analogous meaning, but specified as a safety automaton graphically shown in Figure 5.2.

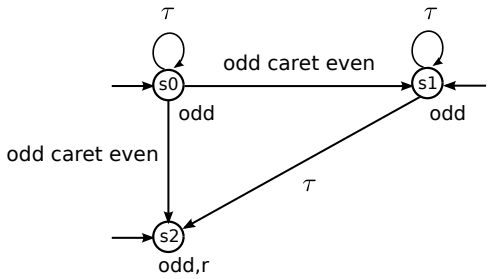


Figure 5.2: Graphical representation of the local specification of method `odd`

As explained in Chapter 3, maximal flow graphs are constructed from structural simulation logic. Therefore, ProMoVer translates all local properties written in LTL and Caret logics to structural simulation logic in order to invoke the maximal flow graph constructor. The safety fragment of LTL is less expressive than simulation logic and can be uniformly encoded in it [44]. Caret formulas can also be directly translated to structural ones. As we shall see, local properties specified as safety-automata can be directly translated to maximal flow graphs. This often results in a significant performance gain due to eliminating the need for maximal flow graph construction. Compositional verification for this case is explained in Paper I.

For *global properties* however the specification languages are restricted by the available model checkers. ProMoVer uses Moped [68] as a pushdown systems model checker, CFP as a model checker for context free processes, and CWB as

a finite state model checker. MOPED is used when global properties are specified in LTL, CFP is used when they are specified in behavioral simulation logic, and CWB for structural ones. Global properties expressed as safety-automata are translated to behavioral simulation logic formulas and thus CFP is used for their model checking.

For example, the global property in Figure 5.1 is written in safety LTL and expresses that "if the execution starts in method `even`, the first call is not to method `even` itself" (*cf.* behavioral simulation logic representation shown in Example 3.4). This global property is checked on the PDS representation of the behavior induced from the flow graphs of methods `even` and `odd` by MOPED.

The users can choose any of the supported languages to specify their local and global properties at hand. PROMOVER automatically detects the languages, performs the translations, and calls model checkers accordingly. However, as we show in Paper I, we observed that often global properties are more naturally expressed as LTL formulas, while local properties are more clearly expressed as automata or in simulation logic. The reason is that usually global properties are partial in nature, expressing certain critical requirements on the behavior of the whole program, while local properties should be as complete as possible, so that all interesting global properties are entailed.

5.1.2 Advanced Features

PROMOVER is equipped with a number of features to address CVPP's principle limitations and facilitate usability, such as *automatic specification extraction*, *proof storage and reuse*, and a *library of global properties*.

Automatic Specification Extraction. PROMOVER provides a facility to extract legal call sequences from a given concrete procedure implementation, by means of static analysis. A user thus does not have to write annotations explicitly: it suffices to inspect the extracted specifications and remove superfluous constraints that might hinder possible evolution of the code. Specifications can be extracted both in simulation logic and as safety automata, so a user can choose the formalism that is more appropriate for the problem at hand, or that he or she is most comfortable with. For example, the local specification of method `odd` in Figure 5.1 has been extracted by using this feature.

Proof Storage and Reuse. PROMOVER fully benefits from compositionality by providing a proof storage and reuse mechanism that stores maximal and program flow graphs, properties, and model checking results. These are reused when the program is re-verified due to a change in a variable component: only the properties that are affected by a change (either in implementation or in specification) are re-verified, all other results are reused. In this way, we avoid the expensive recomputation of models and intermediate results.

Library of Global Properties. To reduce the effort needed to write global properties, ProMoVer is equipped with a library of ready-made common application- and platform-specific global properties. For example, the library currently contains several Java Card specific safety properties such as "no non-atomic operation within a transaction".

5.1.3 Evaluation

In Paper I, we evaluate ProMoVer by a number of Java programs from two application domains. Firstly, we perform experiments on a number of typical Java Card e-commerce applications. We verify the absence of calls to non-atomic methods within transactions. Such properties, specifying legal call sequences for security-related methods, are an important class of platform-specific security properties. Secondly, we use an under-development web application to illustrate the verification of an incomplete program in the presence of code evolution. We verify that only a single connection to a database is created for each incoming request, and that it is properly closed. Properties of this type, specifying safe and efficient usage of a resource, are application-specific properties that are of major importance in the ICT business.

In Paper II, we study the compositional verification of software product lines as an application area for ProMoVer and the CVPP verification methodology. As we discuss in Section 5.2 (and in a greater detail in Paper II) ProMoVer can be used for efficient verification of a class of product families.

5.1.4 Limitations

The first limitation of ProMoVer concerns the level of granularity of verification. The tool implements procedure-modular verification of control flow properties for sequential programs. The restriction to modularity at procedure level is meaningful (as we argued above) but not fundamental, and can be relaxed.

The second limitation concerns the class of properties the tool can handle. Currently, ProMoVer automates verification with the CVPP framework, thus it abstracts away all program data. We plan to release a new version that automates the verification techniques that are developed as instantiations of our generic framework, e.g., Boolean and PoP instantiations (see Section 5.3 and Paper III).

The tool's performance is limited by the principle limitations of the method: the maximal model construction and model checking of global properties (both exponential in the size of the formula), as well as the extraction of precise program models (in particular concerning virtual call resolution and exception propagation). The proof reuse mechanism is our main means of addressing these bottlenecks. Also, as mentioned, the use of safety automata for specifying local properties eliminates the need to construct maximal models.

From a user's point of view, the main limitation is the effort needed to write specifications. However, to assist users, ProMoVer accepts several property spec-

ification languages. It is also equipped with a library of common platform-specific global properties, and a facility for extracting specifications from a given implementation.

5.1.5 Author's Contributions

The first paper included in this thesis is an extended version of two conference papers [98, 99]. I actively took part in writing these papers. I am the main author of these two as well as the enclosed paper. I am the sole developer of ProMoVer. The development of ProMoVer required various extensions and changes in the toolset and model checkers, CWB and Moped. In addition, to provide support for the specification languages Caret and safety-LTL, I developed their translations to simulation logic. I adapted the tools to support the verification using safety-automata specifications. Finally, I performed all the experiments.

5.2 Compositional Verification of Product Families

In the software industry, there is a rising demand for software systems that simultaneously exist in many different variants for different application contexts. *Software product line engineering* [88] is an industrial design approach for developing such systems, that has proven to be commercially successful [78]. This approach aims at developing a *family of products* (or a *product family*) with well-defined *commonalities* and *variabilities* by managed reuse in order to decrease time to market and improve quality. During product line engineering reusable artifacts (or components) are developed that are used to realize the actual products during application engineering.

For the verification of software product lines, often a desired property should be verified for *all* products that can be constructed from a product line. However, the number of products of a product line is (in the worst case) exponential in the number of its artifact implementations. Thus, the verification of software product lines is challenging and would not scale if done by ordinary techniques. The key to scalable verification of such systems is the reuse of verification results by means of *compositionality*. As mentioned in Section 1.1, software product line demonstrates a variability scenario that can particularly benefit from our compositional verification paradigm.

In this work, we develop a compositional verification technique for a specific class of product families here termed *simple families*, following the steps below.

1. We introduce a class of models that specifies—in a hierarchical fashion—the commonalities and variabilities of product lines (see Section 5.2.1 below).

   We define and implement a procedure for constructing these hierarchical models from simple families: Given an existing set of products that form a simple family, our procedure constructs a hierarchical model capturing their commonalities and variabilities (see Section 5.2.1 below).

2. We adapt ProMoVer for efficient verification of hierarchically defined product lines (see Section 5.2.2 below).

The combination of these two steps essentially provides an efficient verification technique for existing simple families. As we shall see, our method is linear in the size of the constructed variability models rather than the number of products.

5.2.1 Hierarchical Variability Models

The variability of the products generated from a software product line can be modeled at different levels [43]. One approach is to describe product variation in terms of so-called *features*, that represent user-visible product characteristics. In this case, the set of valid feature configurations defines the set of possible products. However, features are not necessarily related to the actual *artifacts* that are used to realize the products. Another approach to model product variation is at the design and implementation level where product variation is defined in terms of different artifact implementations that are used to build the actual products. In this work, we are interested in the properties of the *implementation* (code) of product families, thus we focus on the latter approach.

We introduce the notion of *hierarchical variability model (HVM)* to represent the variability of products of product families at the implementation level. On each level of HVMs the commonalities of the sub-products (products that could be constructed by resolving variability up to that level) are specified in a common core. The variabilities are represented by explicit variation points. Each variation point is associated with a set of variants that represent choices for realizing the product variations. A variant can itself contain commonalities defined in a common core and variabilities specified by variation points introducing a new level of hierarchy.

As an example consider the HVM shown in Figure 5.3, where cores, variation points and variants are depicted by red ovals, yellow diamonds and blue squares, respectively. It models a product line of cash desks[1] that process purchases by retrieving the prices for all items to be purchased and calculating the total price. After the customer has paid, a receipt is printed and the stock is updated accordingly. The commonality of all cash desks is that every purchase is processed following the same process, that is implemented by method `sale`. However, the cash desks differ in the way the items are entered. Some cash desks allow entering products using a keyboard, others only provide a scanner, and a third group provides both options which can be chosen by the cashier. These variations are achieved by three different implementations of method `enterProd`. Payment at some cash desks can only be made in cash. Other cash desks only accept credit cards, while a third group allows the choice between cash and credit card payment. These variations are achieved by three different implementations of method `payment`.

[1]The example is a simplified version of the trading system product line case study proposed in [90].

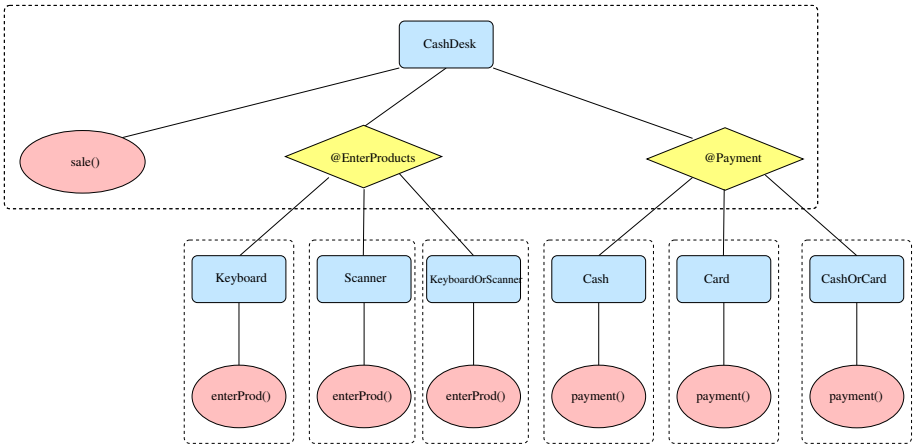


Figure 5.3: The CashDesk variability model

In Paper II, we formally define the semantics of these models and reason about their well-formedness and uniqueness. We define a class of families that can be represented by unique HVMs. We term this class of families *simple* and their hierarchical representation *simple hierarchical variability models (SHVMs)*. We also define and implement a procedure for constructing such models from simple product families. As we show in the paper, the CashDesk variability model shown in Figure 5.3 is an SHVM extracted from a family of cash desks.

To the extent of our knowledge, this work presents the first approach for constructing a hierarchical model from a given product family that captures variability at the implementation level. In fact, most models for product families, such as [105, 56, 16, 104, 82, 7, 63], consider a representation of artifact variability without any hierarchy.

5.2.2 Compositional Verification of SHVMs

As mentioned above, the verification of product families will only scale if performed compositionally. Paper II shows that the CVPP methodology can be adapted to efficiently verify temporal safety properties of product families modeled by SHVMs. To this end, we require local specifications at all variation points. These specifications should abstractly express the legal behavior of all their underlying variants. The idea is that for the verification of variants their underlying variation points and core methods (i.e., their children variation point and core nodes in the graph) can be viewed as abstract and concrete components, respectively. Then, we *localize* the verification of each variation point, and *relativize* the correctness of each variant on the local specifications of its underlying variation points. This results in a hierarchical verification scheme that is realized by following the steps below.

1. Verify each variation point by checking, using step (2), that all its underlying variants satisfy its specification. This essentially means that underlying variants attached to a variation point inherit the property of their parent variation point.

2. Verify each variant by checking that the composition of maximal flow graphs constructed from the local specifications of its underlying variation points, together with the flow graphs extracted from its core methods, satisfy the property of the variant. By this, we basically verify that *all* sub-products constructed by composing the different artifact implementations below a variant satisfy its property.

Since the family corresponds to the root variant, the global property of the software family is the property of the top-level variant of its SHVM. Notice that the verification of variants is relativized on the properties of their underlying variation points, while the correctness of variation points is established through verifying their underlying variants. This results in a hierarchical verification scheme.

For example, to verify the CashDesk product line modeled by the SHVM in Figure 5.3, variation points `@EnterProducts` and `@Payment` are locally specified, and the desired global property of all products would be the property of variant `CashDesk`. Then the verification procedure follows the steps below.

1. Verify that each individual variation point satisfies its property independently. This is achieved for instance for variation point `@EnterProducts` by independently checking that the variants `Keyboard`, `Scanner`, and `KeyboardOrScanner` satisfy the local specification of `@EnterProducts`.

2. Construct maximal flow graph for variation points `@EnterProducts` and `@Payment`, compose these with the flow graphs extracted from the core method `sale` and model check the result against the property of `CashDesk`.

As we show in Paper II, this verification procedure is *sound*, established by the following theorem.

Theorem 5.1. *Let $\mathcal{S}$ be an SHVM with global property ϕ. If the verification procedure succeeds for $\mathcal{S}$, then $p \models \phi$ for all products p of $\mathcal{S}$.*

In Paper II, we describe how ProMoVer is adapted for the verification of product families. The resulting tool accepts Java files annotated with variation point specifications and global properties. ProMoVer then invokes the underlying tools to acquire the verification results. The exact ordering for invoking the tools, and more examples can be found in the paper.

Our approach is one of the first compositional verification techniques for software product lines. It allows to guarantee efficiently that all products of a product line satisfy certain desired control-flow safety properties. The only previously existing compositional verification techniques were proposed by Blundell et al. [20] and

Liu et al. [77]. Both techniques are based on assume-guarantee style reasoning (see [40] for a survey). Other model checking approaches for product lines [50, 80, 57, 74, 38] use a monolithic model of the complete product line, thus face severe state-explosion since all possible products are analyzed in the same analysis step.

5.2.3 Limitations

Our verification technique for product lines is based on the CVPP framework and ProMoVer. Thus, its limitations are similar to the ones of these two: (a) The tool can only verify control-flow properties in the absence of data, (b) it requires manually specified local properties for variation points, and (c) its performance is limited by maximal flow graph construction, model extraction, and model checking. The latter, however, is addressed by means of ProMoVer's proof storage and reuse. Also restriction (a) can be addressed by using our generic compositional verification framework (discussed in Section 5.3 and Paper III) as the verification engine.

5.2.4 Author's Contributions

Paper II is a compilation and extension of the two previous works [92, 61] in which I co-authored the first one. Dilian Gurov and I came up with the verification procedure for SHVM and I proved Theorem 2. I figured out the annotation language, developed the tool, and performed the experiments. I am also the main author of the enclosed paper.

5.3 A Generic Framework for Compositional Verification

Despite that the CVPP framework and ProMoVer have been shown to be practically useful and capable of verifying programs with static and dynamic variability, the fundamental limitation of the method is the restricted class of properties it can verify; the technique abstracts programs by disregarding all data, hence, the range of properties it can handle is limited to control-flow properties in the absence of data. In Paper III, we develop a generalization of the original CVPP framework (explained in Chapter 3) that is capable of capturing program data, and thus bring the capabilities of the framework to a whole new level.

The original CVPP framework abstracts all program data to obtain an algorithmic verification technique that is sufficiently light-weight to be used in practice. Our challenge here is to include data into the program models, specification languages, and maximal model construction by keeping their complexity within practical limits. Thus, our goal is to generalize the CVPP framework for the verification of temporal safety properties of programs written in any procedural language, with minimal additional complexity to the maximal flow graph construction and property specification.

Our key idea is to define a flow graph notion that combines precise representation of some *user selected* types of instructions of the programming language and an abstract representation of the remaining ones. The rationale is that often in temporal reasoning one is interested in observing the ordering of certain events of the system (or instructions of the programming language); the exact ordering of the remaining events need not to be captured, only their cumulative effect. For example, consider the lines of code shown in the left and right columns of Figure 5.4. The columns show two chunks of code that differ in the ordering of the assignments in lines 2 and 3. Let's assume that we are interested in verifying temporal properties talking about methods `malloc` and `delete` and the values of variables exactly before and after a call to them. For the verification of these properties the difference in the ordering of the assignments is irrelevant.

```
   ...                          ...
1  x = malloc(10);          1   x = malloc(10);
2  y = x;                   2   z = x;
3  z = x;                   3   y = x;
4  delete(x);               4   delete(x);
   ...                          ...
```

Figure 5.4

Such irrelevant orderings would not play any role in the verification of the properties of interest; however, they make program models too specific, property formalizations large and complex and maximal model construction inefficient. Thus, in order to have a practically feasible verification technique, we abstract away the exact ordering of the statements that are not relevant for the verification of the properties of interest (in this case assignment statements), only keeping the cumulative effect of series of such statements between two observable statements (in this case the statements `delete(x)` and `malloc(10)`). This effect is captured through Hoare-logic style *assertions* associated with the states of the flow graph. For example, the cumulative effect of assignment statements of the code chunks above can be represented by the logical formula $y' = x \wedge z' = x \wedge x' = x$, where x,y, and z represent the values of the respective variables exactly after the execution of the statement `x = malloc(10)`, and x',y', and z' represent the values of these variables exactly after the execution of both assignment statements.

Using such assertions we define a novel notion of parametric flow graphs that can be fine tuned for the verification of the properties of interest. This will, for instance, result in a flow graph $\mathcal{F}_1$ shown in Figure 5.5 for both of the code chunks above. Notice that each state of these flow graphs is tagged with a set of atomic propositions (in this case m) as well as an assertion. (The formal definition of such flow graphs can be found in Section 3 of Paper III.)

We assume a *semantic entailment* relation on these logical assertions and use it to adapted the definitions of simulation relation (defined in Definition 3.2), simu-

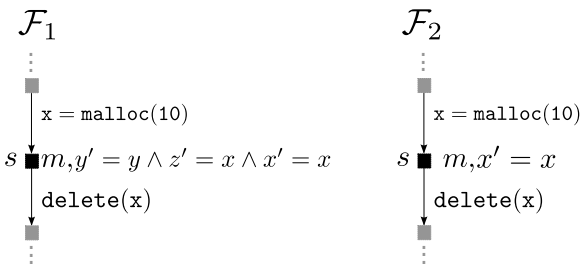


Figure 5.5: $\mathcal{F}_1$: a flow graph modeling both code chunks in Figure 5.4 we assume that the code chunks belong to method m
$\mathcal{F}_2$: a flow graph simulating flow graph $\mathcal{F}_1$

lation logic and logical satisfaction (defined in Definition 3.4) to the new definition of program models.

We say that state s_1 is simulated by state s_2 if the assertion of s_1 entails the one of s_2 and s_1 is simulated by s_2 according to the criterion explained in Definition 3.2. Intuitively, this means that s_2 allows more flexibility for the values of variables as compared to s_1. For example, the flow graph $\mathcal{F}_2$ shown in Figure 5.5 simulates $\mathcal{F}_1$, allowing any change to the value of variables y and z between the statements `x = malloc(10)` and `delete(x)`. By properly defining the behavior, we guarantee that $\mathcal{F}_2$ simulates $\mathcal{F}_1$ implies that the behavior of $\mathcal{F}_2$ simulates the behavior of $\mathcal{F}_1$ (i.e., the re-establishment of Theorem 3.10 which is essential for the correctness of our compositional verification technique).

We also extend the definition of simulation logic to include assertions as atomic formulas (see Definition 3.4). Then the definition of satisfaction is extended accordingly: we say that state s of a model satisfies assertion ϕ if the assertion of s entails ϕ. For example, state s of the flow graphs in Figure 5.5 satisfies the atomic formula "$x' = x$".

Given the new definitions of simulation logic and satisfaction, we can generalize the maximal model construction algorithm explained in Section 3.3 for the construction of maximal flow graphs with data without adding extra complexity to it. This is achieved by the following two steps: 1) construct the maximal flow graph using the algorithm explained in Section 3.3, only carrying over the assertions of the formula's equations, and 2) assign the top element of the lattice of assertions (ordered by the semantic entailment relation) to the states of the resulting flow graph that are not tagged with any assertion. In Paper III, we prove that maximal flow graphs can be constructed by the above procedure.

5.3.1 Instantiations of the Generic Framework

We present three instantiations of our generic verification framework. The first instantiation abstracts away all data as in the original CVPP framework. We

show that this instantiation is isomorphic to the original framework. The second one is an instantiation for the verification of *Boolean programs* [11]. Boolean programs are procedural programs with Boolean variables as the only datatype. This language has been studied and used as abstract representation of real-life programming languages in several works [14, 10, 12, 87, 42]. In our previous work, we used this language as a first step towards adding full data support to the CVPP framework [95][2]. The capability of handling Boolean data shows that our generic framework can handle data from finite domains. In a third and most challenging instantiation, we exemplify the use of our framework for the verification of programs written in a procedural language with heap pointers as the only datatype (PoP). Dealing with this language is challenging because, in addition to unbounded call stacks, it can give rise to infinite state spaces for yet another reason, namely unbounded pointer creation. This instantiation shows how our framework can cope with data from infinite domains.

To evaluate our theory we adapted the CVPP toolset for the verification of PoP programs and performed some experiments. The results of the experiments are shown in Paper III.

5.3.2 Limitations and Future Work

Our generic framework inherits principle limitations of the original CVPP framework explained in Section 1.1: the maximal flow graph construction and model computation are complex, and property specification is delicate. However, as we discussed above and show in more details in Paper III, we do not add extra computational complexity to the original CVPP framework. These limitations can be pragmatically addressed by extending ProMoVer for instantiations of the generic framework.

So far, except the flow graph extractor, we have lifted all other tools to provide support for PoP programs. Still, our toolset is lacking a flow graph extractor to extract PoP flow graphs from programs written in real-life procedural languages, e.g., C++ and Java. We are currently working on a parametric flow graph extractor to extract flow graphs of Java programs for the given sets of actions and assertions.

Our tools currently handle PoP programs. It is not difficult to extend the support for Boolean programs. However, it is more challenging to deal with programs with more than one datatype. In this case, a set of assertions needs to be defined for each of the datatypes.

5.3.3 Author's Contributions

This work was mainly carried out by me. I came up with the idea of abstracting a sequence of instructions by a logical formula during my discussions with Frank de Boer, Marcello Bonsangue, and Jurriaan Rot. Later, this idea was developed to a

[2] As we shall see, the way we capture Boolean data in that work is different with the one we explain here.

program model combining the precise ordering of some instructions with abstract effect of others by me, under a careful supervision of Dilian Gurov.

The generalization of the CVPP framework was carried out by me. I re-established the necessary results, proved the correctness of the technique, developed a generalization of the toolset to PoP language, and performed the experiments. I am the main author of the paper and the technical report. All these were achieved in constant collaboration with Dilian Gurov.

Chapter 6

Conclusion

This thesis provides a solution for the verification of temporal safety properties of procedural programs with static and dynamic variability. This work is developed on top of the CVPP framework; we address its limitations and restrictions by introducing ProMoVer tool and a novel generic compositional verification framework that generalizes it. We also provide an efficient compositional verification technique for software product families as an important application area for our framework. In this section, we provide the highlights of these works. The detailed conclusions and future work are delegated to the attached papers.

ProMoVer automates the procedure-modular verification of temporal safety properties. It takes as input a Java program annotated with temporal specifications. The tool is capable of performing verification in the presence of variability, and therefore can be used in code evaluation and customization scenarios such as incomplete programs, mobile code, and software families. With ProMoVer we show that procedure-modular verification of temporal safety properties can be done automatically, and temporal logic provides a meaningful abstraction for individual program methods.

ProMoVer provides pragmatic solutions for the principle limitations of the CVPP framework. It accepts as input several specification formalisms to simplify the property specification. It is also equipped with the specification extractor which extracts candidate specifications from the implementation of methods. In addition, ProMoVer's proof storage and reuse mechanism minimizes the use of the flow graph extraction, maximal flow graph construction, and model checking tasks.

We discussed the compositional verification of software families as a particular application area for our technique. We explained that in Paper II a novel hierarchical variability model is proposed for product families and ProMoVer is adapted for the verification of these hierarchical models. It is also shown that the number of verification tasks resulting from non-compositional verification of software product families is exponential in the size of the variability model; however, it is linear for our compositional verification technique.

The main restriction of the CVPP framework is the abstraction from all program data. In Paper III this restriction is lifted by theoretically extending the definitions of flow graph, program behavior, and specification languages to include data values. Also the maximal flow graph construction is adapted accordingly. It is shown that most of the previous definitions and theorems can be successfully adapted for the framework with data as well. By this extension, a significantly wider range of properties is supported.

Bibliography

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

[2] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–535, May 1995.

[3] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[4] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27:786–818, 2005.

[5] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic for nested calls and returns. In *Tools and Algorithms for the Analysis and Construction of Software (TACAS)*, volume 2998 of *Lecture Notes in Computer Science*, pages 467–481. Springer, 2004.

[6] A. Amighi, P. de Carvalho Gomes, D. Gurov, and M. Huisman. Sound control-flow graph extraction for java programs with exceptions. In *Software Engineering and Formal Methods (SEFM)*, volume 7504 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2012.

[7] S. Apel, F. Janda, S. Trujillo, and C. Kästner. Model Superimposition in Software Product Lines. In *International Conference on Model Transformation (ICMT)*, volume 5563 of *Lecture Notes in Computer Science*, pages 4–19. Springer, 2009.

[8] A. Arnold and D. Niwiński. *Rudiments of μ-calculus*, volume 146 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Publishing, 2001.

[9] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *ACM SIGPLAN Notices*, volume 36, pages 203–213. ACM, 2001.

[10] T. Ball, A. Podelski, and S. Rajamani. Boolean and cartesian abstraction for model checking C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2001.

[11] T. Ball and S. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, 2000.

[12] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN Workshop on SPIN Model Checking and Software Verification*, pages 113–130. Springer-Verlag, 2000.

[13] T. Ball and S. K. Rajamani. The slam toolkit. In *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264. Springer, 2001.

[14] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Principles of programming languages (POPL)*, pages 1–3, 2002.

[15] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS, pages 49–69, Berlin, Heidelberg, 2005. Springer-Verlag.

[16] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transaction Software Engineering*, 30(6):355–371, 2004.

[17] H. Bekič. Definable operators in general algebras, and the theory of automata and flowcharts. Technical report, IBM Laboratory, 1967.

[18] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and software verification: model-checking techniques and tools*. Springer Publishing Company, Incorporated, 2010.

[19] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 193–207. Springer-Verlag, 1999.

[20] C. Blundell, K. Fisler, S. Krishnamurthi, and P. van Hentenryck. Parameterized Interfaces for Open System Verification of Product Lines. In *Automated Software Engineering (ASE)*, pages 258–267. IEEE, 2004.

[21] B. Boigelot and P. Godefroid. Model checking in practice: An analysis of the access.bus protocol using SPIN. In *International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods (FME)*, pages 465–478. Springer-Verlag, 1996.

[22] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. In *Handbook of Process Algebra*, pages 545–623. North Holland, 2000.

[23] O. Burkart and B. Steffen. Model checking for context-free processes. In *CONCUR*, volume 630 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 1992.

[24] S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: An industrial case study. In *International Conference on Software Engineering (ICSE)*, pages 431–441, New York, NY, USA, 2002. ACM.

[25] G. Chugunov, L. Åke Fredlund, L.-A. Fredlund, and D. Gurov. Model checking of multi-applet javacard applications. In *Smart Card Research and Advanced Application Conference (CARDIS)*, pages 87–95. USENIX Publications, 2002.

[26] E. Clarke. The birth of model checking. In *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin Heidelberg, 2008.

[27] E. Clarke and E. Emerson. Design and synthesis of synchronisation skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[28] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin Heidelberg, 1982.

[29] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[30] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *Computer Aided Verification (CAV)*, pages 419–422. Springer, 1996.

[31] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, Apr. 1986.

[32] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[33] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the futurebus+ cache coherence protocol. In *IFIP WG10.2 International Conference Sponsored by IFIP WG10.2 and in Cooperation with IEEE COMPSOC on Computer Hardware Description Languages and Their Applications (CHDL)*, pages 15–30, Amsterdam, The Netherlands, 1993. North-Holland Publishing Co.

[34] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 176–194. Springer-Verlag, 2001.

[35] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking. In *NATO ASI DPD*, pages 305–349, 1996.

[36] E. M. Clarke, D. E. Long, and K. L. Mcmillan. Compositional model checking. In *Logic in Computer Science (LICS)*. IEEE Computer Society, 1989.

[37] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[38] A. Classen, P. Heymans, P. Schobbens, A. Legay, and J. Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *International Conference on Software Engineering (ICSE)*, pages 335–344. IEEE, 2010.

[39] R. Cleaveland, J. Parrow, and B. Steffen. A semantics based verification tool for finite state systems. In *International Symposium on Protocol Specification, Testing and Verification (PSTV)*, pages 287–302. North-Holland Publishing Co., 1990.

[40] P. Clements and L. Northrop. *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[41] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. Vcc: A practical system for verifying concurrent C. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 23–42, Berlin, Heidelberg, 2009. Springer-Verlag.

[42] B. Cook, D. Kroening, and N. Sharygina. *Symbolic model checking for asynchronous boolean programs*. Springer, 2005.

[43] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[44] M. Dam. CTL* and ECTL* as fragments of the modal μ-calculus. In *Colloquium on Trees in Algebra and Programming, (CAAP)*, volume 581 of *LNCS*, pages 145–164. Springer, 1992.

[45] P. de Carvalho Gomes, A. Picoco, and A. Amighi. ConFlEx. `http://www.csc.kth.se/~pedrodcg/conflex`.

[46] P. de Carvalho Gomes, A. Picoco, and D. Gurov. Sound control flow graph extraction from incomplete java bytecode programs. In *Fundamental Approaches to Software Engineering (FASE)*, volume 8411 of *Lecture Notes in Computer Science*, pages 215–229, Berlin, 2014. Springer.

[47] S. Eilenberg. *Automata, languages, and machines*. Pure and Applied Mathematics. Elsevier Science, 1974.

[48] E. Emerson and E. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Automata, Languages and Programming (ICALP)*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer Berlin Heidelberg, 1980.

[49] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 2000.

[50] A. Fantechi and S. Gnesi. Formal Modeling for Product Families Engineering. In *Software Product Line Conference (SPLC)*, pages 193–202. IEEE, 2008.

[51] C. Flanagan and K. Leino. Houdini, an annotation assistant for ESC/Java. In *Formal Methods for Increasing Software Productivity (FME)*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer Berlin / Heidelberg, 2001.

[52] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java, 2002.

[53] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. *ACM SIGPLAN Notices*, 37(1):191–202, 2002.

[54] N. Gawell. Automatic verification of applet interaction properties. Master's thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2009. Ref.: TRITA-CSC-E 2009:128.

[55] P. Godefroid. Partial-order methods for the verification of concurrent systems - an approach to the state-explosion problem, 1995.

[56] H. Gomaa. *Designing Software Product Lines with UML*. Addison Wesley, 2004.

[57] A. Gruler, M. Leucker, and K. Scheidemann. Modeling and model checking software product lines. In *Formal Methods for Open Object-based Distributed Systems (FMOODS)*, volume 5051 of *Lecture Notes in Computer Science*, pages 113–131. Springer, 2008.

[58] O. Grumberg and D. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):843–871, 1994.

[59] D. Gurov and M. Huisman. Reducing behavioural to structural properties of programs with procedures. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 5403 of *Lecture Notes in Computer Science*, pages 136–150. Springer, 2009.

[60] D. Gurov, M. Huisman, and C. Sprenger. Compositional verification of sequential programs with procedures. *Information and Computation*, 206(7):840–868, 2008.

[61] D. Gurov, B. Østvold, and I. Schaefer. A hierarchical variability model for software product lines. In *International Symposium On Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA)*, pages 181–199. Springer Berlin Heidelberg, 2012.

[62] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral interface specification languages, 2009.

[63] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. Olsen, and A. Svendsen. Adding Standardized Variability to Domain Specific Languages. In *Software Product Line Conference (SPLC)*, pages 139–148. IEEE, 2008.

[64] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32:137–161, 1985.

[65] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Model Checking Software*, pages 235–239. Springer, 2003.

[66] L. Hubert, N. Barré, F. Besson, D. Demange, T. Jensen, V. Monfort, D. Pichardie, and T. Turpin. Sawja: Static Analysis Workshop for Java. In *Formal Verification of Object–Oriented Software (FoVeOOS)*, volume 6528 of *Lecture Notes in Computer Science*. Springer, 2010.

[67] M. Huisman and D. Gurov. CVPP: A tool set for compositonal verification of control-flow safety properties. In *Formal Verification of Object–Oriented Software (FoVeOOS)*, volume 6528 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2010.

[68] S. Kiefer, S. Schwoon, and D. Suwimonteerabuth. Moped - a model-checker for pushdown systems. `http://www.informatik.uni-stuttgart.de/fmi/szs/tools/moped/`.

[69] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[70] S. A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[71] O. Kupferman and M. Vardi. An automata-theoretic approach to modular model checking. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):87–128, 2000.

[72] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, Mar. 1977.

[73] K. Larsen. Modal specifications. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 232–246. Springer, 1989.

[74] K. Lauenroth, K. Pohl, and S. Toehning. Model checking of domain artifacts in product line engineering. In *Automated Software Engineering (ASE)*, pages 269–280. IEEE, 2009.

[75] K. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer Berlin Heidelberg, 2010.

[76] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Principles of programming languages (POPL)*, pages 97–107, New York, NY, USA, 1985. ACM.

[77] J. Liu, S. Basu, and R. R. Lutz. Compositional model checking of software product lines using variation point obligations. *Automatic Software Engineering*, 18(1):39–76, 2011.

[78] J. D. McGregor, D. Muthig, K. Yoshimura, and P. Jensen. Guest editors' introduction: Successful software product line practices. *IEEE Software*, 27:16–21, 2010.

[79] K. L. McMillan. *Symbolic model checking*. Springer, 1993.

[80] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[81] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer, 2002.

[82] N. Noda and T. Kishi. Aspect-Oriented Modeling for Variability Management. In *Software Product Line Conference (SPLC)*, pages 213–222. IEEE, 2008.

[83] D. Peled. All from one, one for all: on model checking using representatives. In *Computer Aided Verification (CAV)*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer Berlin Heidelberg, 1993.

[84] H. Peng and S. Tahar. A survey on compositional verification. Technical report, Concordia University, 1998.

[85] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science (FOCS)*, Lecture Notes in Computer Science, pages 46–57. IEEE Computer Society, 1977.

[86] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and models of concurrent systems*, pages 123–144. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[87] A. Podelski and T. Wies. Boolean heaps. In *Static Analysis*, volume 3672 of *Lecture Notes in Computer Science*, pages 268–283. Springer Berlin Heidelberg, 2005.

[88] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques.* Springer, 2005.

[89] J. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer Berlin Heidelberg, 1982.

[90] Requirement Elicitation, Aug. 2009. Deliverable 5.1 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[91] J. Rot, F. S. de Boer, and M. M. Bonsangue. Unbounded allocation in bounded heaps. In *Fundamentals of Software Engineering (FSEN)*, volume 8161 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2013.

[92] I. Schaefer, D. Gurov, and S. Soleimanifard. Compositional algorithmic verification of software product lines. In *Formal Methods for Components and Objects (FMCO)*, volume 6957 of *Lecture Notes in Computer Science*, pages 184–203. Springer, 2010.

[93] F. B. Schneider. Enforceable security policies. *ACM Transaction Infinite Systems Security*, 3(1):30–50, 2000.

[94] S. Schwoon. *Model-Checking Pushdown Systems.* PhD thesis, Technische Universität München, 2002.

[95] S. Soleimanifard. Procedure-modular verification of temporal safety properties, 2012. QC 20120507.

[96] S. Soleimanifard and D. Gurov. Algorithmic verification of procedural programs in the presence of code variability, 2014. Technical Report, Available at `http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-128950`.

[97] S. Soleimanifard, D. Gurov, and M. Huisman. ProMoVer web interface. `http://www.csc.kth.se/~siavashs/ProMoVer`.

[98] S. Soleimanifard, D. Gurov, and M. Huisman. Procedure–modular verification of control flow safety properties. In *Workshop on Formal Techniques for Java Programs (FTfJP)*, 2010.

[99] S. Soleimanifard, D. Gurov, and M. Huisman. Promover: Modular verification of temporal safety properties. In *Software Engineering and Formal Methods (SEFM)*, volume 7041 of *Lecture Notes in Computer Science*, pages 366–381. Springer, 2011.

[100] S. Soleimanifard, D. Gurov, and M. Huisman. Procedure-modular specification and verification of temporal safety properties. *Software & Systems Modeling*, pages 1–18, 2013.

[101] C. Stirling. *Modal and Temporal Logics of Processes*. Springer, 2001.

[102] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdds. In *International Conference on Computer Aided Design (ICCAD)*, pages 130–133, 1990.

[103] A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer Berlin Heidelberg, 1991.

[104] M. Völter and I. Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *Software Product Line Conference (SPLC)*, pages 233–242. IEEE, 2007.

[105] T. Ziadi, L. Hélouët, and J. Jézéquel. Towards a UML Profile for Software Product Lines. In *Software Product Familiy Engineering (PFE)*, volume 3014 of *Lecture Notes in Computer Science*, pages 129–139. Springer, 2003.

Part II

Included Papers

Appendix A

Paper I: Procedure-Modular Specification and Verification of Temporal Safety Properties

Procedure-Modular Specification and Verification of Temporal Safety Properties*

Siavash Soleimanifard[1], Dilian Gurov[1], and Marieke Huisman[2]

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

Abstract. This paper describes ProMoVer, a tool for fully automated procedure–modular verification of Java programs equipped with method–local and global assertions that specify safety properties of sequences of method invocations. Modularity at the procedure–level is a natural instantiation of the modular verification paradigm, where correctness of global properties is relativized on the local properties of the methods rather than on their implementations. Here it is based on the construction of maximal models for a program model that abstracts away from program data. This approach allows global properties to be verified in the presence of code evolution, multiple method implementations (as arising from software product lines), or even unknown method implementations (as in mobile code for open platforms).
ProMoVer automates a typical verification scenario for a previously developed tool set for compositional verification of control flow safety properties, and provides appropriate pre– and post–processing. Both linear–time temporal logic and finite automata are supported as formalisms for expressing local and global safety properties, allowing the user to choose a suitable format for the property at hand. Modularity is exploited by a mechanism for proof reuse that detects and minimizes the verification tasks resulting from changes in the code and the specifications. The verification task is relatively light–weight due to support for abstraction from private methods and automatic extraction of candidate specifications from method implementations. We evaluate the tool on a number of applications from the domains of Java Card and web–based application.

1 Introduction

In modern computing systems, code changes frequently. Modules (or components) evolve rapidly or exist in multiple versions customized for various users, and in mobile contexts, a system may even automatically reconfigure itself. As a result, systems are no longer developed as monolithic applications; instead they are composed of ready–made off–the–shelf components, and each component

* Soleimanifard's work is funded by the ContraST project of the Swedish Research Council VR, and Gurov's work by the EU FET project FP7–ICT–2009–3 HATS. Huisman's work is partially funded by ERC grant 258405 for the VerCors project.

may be dynamically replaced by a new one that provides improved or additional functionality. This static and dynamic *variability* makes it more important to provide formal correctness guarantees for the behaviour of such systems, but at the same time also more difficult. *Modularity* of verification is a key to providing such guarantees in the presence of variability.

In modular verification, correctness of the software components is specified and verified independently (*locally*) for each module, while correctness of the whole system is specified through a *global* property, the correctness of which is verified relative to the local specifications rather than relative to the actual implementations of the modules. It is this relativization that enables verification of global properties in the presence of static and dynamic variability. In particular, it allows an independent evolution of the implementations of individual modules, only requiring the re–establishment of their local correctness.

Hoare logic provides a popular framework for modular specification and verification of software, where it is natural to take the individual procedures as modules, in order to achieve scalability, see e.g., [22]. While Hoare logic allows the *local effect* of invoking a given procedure to be specified, temporal logic is better suited for capturing its *interaction with the environment*, such as the allowed sequences of procedure invocations. This paper shows that procedure–modular verification is also appropriate for control flow safety temporal logic: for each procedure the local property specifies its legal call sequences, while the system's global property specifies the allowed interactions of the system as a whole. Thus, temporal specifications provide a meaningful abstraction for procedures.

Control flow safety properties can be expressed in various formalisms, such as automata–based or process–algebraic notations, as well as in temporal logics such as LTL [30] or the safety fragment of the modal μ-calculus [19]. The approach that is described in this paper supports two of those formalisms, namely LTL and a variant of finite automata termed here safety automata. This is convenient in particular when writing properties of different nature and at different levels of abstraction and component granularity. Global specifications, for instance, are usually partial in nature, expressing certain critical requirements on the behaviour of the whole program. In contrast, local specifications should be as complete as possible, so that all interesting global properties are entailed. So, candidate local specifications extracted from an existing implementation would be more naturally represented with automata, while an abstract, global temporal restriction may be more naturally phrased in LTL. On the other hand, expressiveness of specification provides as usual convenience at the expense of algorithmic efficiency. Certain algorithmic problems, such as model checking and maximal flow graph construction (see below) are more efficiently solved if procedure calls are treated atomically, essentially reducing the context–free infinite–state behaviour of the program to its finite–state textual structure. The resulting restricted properties turn out to be adequate for specifying local properties of individual procedures (without self–calls), but are in general too inexpressive for higher levels of component granularity, and in particular for specifying global properties.

To support our approach, we have developed a fully automated verification tool, ProMoVer, which can be tried via a web–based interface [28]. It takes as input a Java program annotated with global and method–local correctness assertions written in temporal logic and it automatically invokes a number of tools from cvpp, a previously developed tool set for compositional verification [17], to perform the individual local and global correctness checks. Internally, cvpp uses the safety fragment of the modal μ-calculus as a property specification language, but ProMoVer also allows the user to write specifications in LTL, or as so–called safety automata, which are a variant of Schneider's security automata [27].

Essentially, ProMoVer is a wrapper that performs a standard verification scenario in the general tool set, to demonstrate that procedure–modular verification of temporal safety properties can be automated completely by using annotated programs as a single input. Importantly, ProMoVer only requires the public procedures to be annotated; the private ones are being considered merely as an implementation means. In addition, ProMoVer provides a facility to extract a method's legal call sequences by means of static analysis, given a concrete procedure implementation. A user thus does not have to write annotations explicitly; it suffices to inspect the extracted specifications and remove superfluous constraints that might hinder possible evolution of the code. Specifications can be extracted both in LTL and as safety automata, so a user can choose the formalism that is more appropriate for the problem at hand, or that he or she is most comfortable with. Finally, ProMoVer also practically supports modularity by providing proof storage and reuse: only the properties that are affected by a change (either in implementation or in specification) are reverified, all other results are reused.

We show validity of the approach on a number of Java programs from two application domains. Firstly, we perform experiments on some typical Java Card e-commerce applications. Such security–relevant applications are an important target for formal verification techniques. Here, we verify the absence of calls to non–atomic methods within transactions. Such properties, specifying legal call sequences for security–related methods, are an important class of platform–specific security properties. Secondly, we use an under–development web application to illustrate the verification of an open system in the presence of code evolution. Here, we verify that only a single connection to a database is created for each incoming request, and that it is properly closed. Properties of this type, specifying safe and efficient usage of a resource, are application–specific properties that are of major importance in the ICT business. The ProMoVer web interface allows the user to verify both platform– and application–specific properties, for which ready–made formalizations are provided.

To allow efficient algorithmic modular verification, the tool set currently abstracts away from all data, thus considering safety properties of the control flow; in particular, method calls in Java programs are over–approximated by non–deterministic choice on possible method implementations that the virtual call resolution might resolve to. This rather severe restriction on the *program model* facilitates the maximal model construction that is at the core of our modular

verification technique (see [13] for a proof of soundness and completeness for this program model). Still, many useful properties can be expressed at this level of abstraction. Besides the platform–specific and application–specific security properties discussed above, we can for example express properties such as: (*i*) a method that changes sensitive data is only called from within a dedicated authentication method, i.e., unauthorized access is not possible; or (*ii*) in a voting system, candidate selection has to be finished, before the vote can be confirmed. Extending the technique with data, either over finite domains or over pointer structures, will allow for a wider range of properties and possible applications, but requires a non–trivial generalization of the maximal model construction, and needs to be combined with abstraction techniques to control the complexity of verification and of model extraction from a program. We are currently investigating this.

The work in this paper is closely related to the development of CVPP [17]. As already pointed out, ProMoVer is essentially a wrapper that automates a typical verification scenario for CVPP, where modularity is applied at the procedure–level. In addition, ProMoVer provides support for different property specification languages, proof reuse, specification extraction, a collection of ready–formalized properties, and a translation between the different intermediate formats and formalisms. Results on a previous version of ProMoVer are reported in [29]. The present paper extends this earlier work by introducing an automata–based specification language and its modular verification principle. The use of the additional specification language is evaluated on a number of case studies, and is compared with the verifications based on the original LTL specifications. Furthermore, this paper presents an evaluation of ProMoVer on a significantly larger case study, representing an open system in the presence of code evolution.

Limitations. ProMoVer currently handles *procedure–modular verification* of control flow properties for sequential programs. The restriction to modularity at procedure level is meaningful (as we argue above) but not fundamental, and will be relaxed in future versions. As mentioned above, we are working on extending the method with data. The underlying theory for modeling multi–threaded programs has been developed earlier (see [16]), but the model checking problem is not decidable in general and has to be approximated suitably.

From a more practical point of view, the two main limitations are performance and the effort needed to write specifications. With respect to the first limitation, known theoretical bottlenecks are the maximal model construction and model checking of global properties (both exponential in the size of the formula), as well as the efficient extraction of precise program models (in particular concerning virtual call resolution and exception propagation). The support for proof reuse is our main means of addressing these bottlenecks. Notice also that the use of safety automata for specifying local properties eliminates the need to construct maximal models, since the automata themselves play the role of maximal models. As to the second limitation, to reduce the effort needed to write specifications, ProMoVer provides a library of common platform–specific

global properties, and a facility for extracting specifications from a given implementation, as explained above.

Related Work. A non–compositional verification method based on a program model closely related to ours is presented by Alur *et al.* [3]. It proposes a temporal logic CaRet for nested calls and returns (generalized to a logic for nested words in [1]) that can be used to specify regular properties of local paths within a procedure that skips over calls to other procedures. esp is another example of a successful system for non–compositional verification of temporal safety properties, applied to C programs [8]. It combines a number of scalable program analyses to achieve precise tracking (simulation) of a given property on multiple stateful values (such as file handles), identified through user–defined source code patterns. Maven is a modular verification tool addressing temporal properties of procedural languages, but in the context of aspects [11]. Recent work by Alur and Chauhuri proposes a unification of Hoare–style and Manna–Pnueli–style temporal reasoning for procedural programs, presenting proof rules for procedure–modular temporal reasoning [2].

Overview. The rest of this paper is organized as follows. Section 2 presents the use of ProMoVer from a user's point–of–view. Section 3 describes the underlying program model and Section 4 explains the property specification languages and compositional verification method based on constructing maximal models. Then, Section 5 describes the ProMoVer tool, while Section 6 describes several realistic case studies using the tool. Finally, the last section draws conclusions and suggests directions for future research.

2 ProMoVer: A User's View

We start by illustrating how ProMoVer is used on a small example. Both local method and global program properties are provided as assertions in the form of program annotations. We use a JML–like syntax for annotations (*cf.* [21]). ProMoVer is procedure–modular in the sense that correctness of the global program property is relativized on the local specifications of the individual methods. Thus, the overall verification task divides into two independent subtasks:

(*i*) a check that each method implementation satisfies its local specification, and
(*ii*) a check that the composition of local specifications entails the global property.

Notice that the second subtask only relies on the local specifications and does not require the implementations of the individual methods. Thus, changing a method implementation does not require the global property to be reverified, only the local specification. If the second subtask fails, ProMoVer translates the counterexample provided by the underlying tools into the form of a program behavior that is allowed by the local specifications, but violates the global one[1].

[1] Unfortunately, not all tools that we use provide counterexamples.

```
/**
 * @global_ltl_prop: even -> X ((even && !entry) W odd)
 */
public class EvenOdd {
   /** @local_interface: required odd
    *  @local_ltl_prop:
    *          G (X (!even || !entry) && (odd -> X G even))
    */
   public boolean even(int n) {
      if (n == 0)
        return true;
      else
        return odd(n-1);
   }

   /** @local_interface: required even
    *  @local_sa_prop:
    *          node s0 odd, entry
    *          node s1 odd, entry
    *          node s2 odd, entry, r
    *          edge s0 s0 tau
    *          edge s0 s1 odd caret even
    *          edge s0 s2 odd caret even
    *          edge s1 s1 tau
    *          edge s1 s2 tau
    */
   public boolean odd(int n) {
      if (n == 0)
        return false;
      else
        return even(n-1);
   }
}
```

Fig. 1: A simple annotated Java program

In addition to the properties, the technique also requires global and local *interfaces*. A global interface consists of a list of the methods *provided* (i.e., implemented) and *required* (i.e., used) by the program. The local interface of method m contains a list of the methods *required* by the method (as the provided method is obvious). ProMoVer can extract both global and local interfaces from method implementations.

Example 1. Consider the annotated Java program in Figure 1. It consists of two methods, `even` and `odd`. The program is annotated with a global control flow safety property expressed in LTL, and every method is annotated with a local property and an interface specifying the required methods. The local property of method `even` is expressed in LTL, while method `odd` is specified with a safety automaton. Here we only give an intuitive description of the properties specified in the example; formal definitions of the temporal logic LTL and safety automata are given in Section 4.

The global property expresses that "in every program execution starting in method `even`, the first call is not to method `even` itself". The local property of method `even` expresses that "method `even` can only call method `odd`, and after returning from the call, no other method can be called". The local property of

method `odd` is analogous but is expressed as a safety automaton (ASCII notation in Figure 1, and visualized in Figure 3 on page 12).

As mentioned above, the interfaces and local method specifications can be extracted from the method implementations automatically by ProMoVer (see Section 5).

As explained above, the annotated program is correct if (*i*) methods `even` and `odd` meet their respective local specifications, and (*ii*) the composition of all local specifications entails the global one. In fact, the annotated program is correct and our tool therefore returns an affirmative result.

Example 2. If we change the global property of the previous example to "in every program execution starting in method `even`, no call to method `odd` is made", the tool detects this and rechecks the global property for the already computed composition of local specifications. The local specifications do not have to be reverified. The verification of the global property fails. As a counterexample, ProMoVer returns the following program execution that is allowed by the local specifications, but violates the global one:

$$(\texttt{even}, \varepsilon) \xrightarrow{\texttt{even call odd}} (\texttt{odd}, \texttt{even}) \xrightarrow{\texttt{odd ret even}} (\texttt{even}, \varepsilon)$$

This counterexample, adapted for user understandability by replacing program points with the names of the methods they belong to (*cf.* Definition 4), should be understood as follows: from method `even`, method `odd` is called, and then method `odd` returns, and control is given back to `even`. This violates the desired global property, because `odd` is called from `even`.

3 Program Model

In this and the following section, we briefly present the formal framework underlying the ProMoVer tool that supports procedure–modular verification as illustrated above. It is heavily based on our earlier work on compositional verification [13, 12]. Here, we define our program model.

3.1 Models and Simulation

First, we formally define the abstract structure on which our program model and its operational semantics are based.

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

The definition of *simulation* on models is standard.

Definition 2 (Simulation). *A simulation on model $\mathcal{M}$ is a binary relation R on S such that whenever $(s,t) \in R$ then $\lambda(s) = \lambda(t)$, and whenever $s \xrightarrow{a} s'$ then there is some $t' \in S$ such that $t \xrightarrow{a} t'$ and $(s',t') \in R$. We say that t simulates s, written $s \leqslant t$, if there is a simulation R such that $(s,t) \in R$.*

Simulation on two models $\mathcal{M}_1$ and $\mathcal{M}_2$ is defined as simulation on their disjoint union $\mathcal{M}_1 \uplus \mathcal{M}_2$. The transitions of $\mathcal{M}_1 \uplus \mathcal{M}_2$ are defined by $in_i(s) \xrightarrow{a} in_i(s')$ if $s \xrightarrow{a} s'$ in $\mathcal{M}_i$ and its valuation by $\lambda(in_i(S)) = \lambda_i(S)$, where in_i (for $i \in \{1,2\}$) injects S_i into $S_1 \uplus S_2$. Simulation is extended to initialized models $(\mathcal{M}_1, E_1)$ by defining $(\mathcal{M}_1, E_1) \leqslant (\mathcal{M}_2, E_2)$ if there is a simulation R on $\mathcal{M}_1 \uplus \mathcal{M}_2$ such that for each $s \in E_1$ there is some $t \in E_2$ with $(in_1(s), in_2(t)) \in R$.

3.2 Flow Graphs

Our program model is based on the notion of *flow graph*, abstracting away from all data in the original program. It is essentially a collection of *method graphs*, one for each method of the program. Let *Meth* be a countably infinite set of methods names. A method graph is an instance of the general notion of initialized model.

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

Notice that methods can have multiple entry nodes. Flow graphs that are extracted from program source have single entry points, but the maximal models that we generate for compositional verification may have several.

Every flow graph $\mathcal{G}$ is equipped with an *interface* $I = (I^+, I^-)$, denoted $\mathcal{G} : I$, where $I^+, I^- \subseteq Meth$ are the *provided* and externally *required* methods, respectively. These are needed to construct maximal flow graphs (see Section 4.2).

A flow graph is *closed* if its interface does not require any methods, and it is *open* otherwise. Flow graph *composition* is defined as the disjoint union $\uplus$ of their method graphs.

Example 3. Figure 2 shows the flow graph of the program from Figure 1. Its interface is $(\{\texttt{even}, \texttt{odd}\}, \emptyset)$, thus the flow graph is closed. It consists of two method graphs, for method `even` and method `odd`, respectively. Entry nodes are depicted as usual by incoming edges without source.

The operational semantics of flow graphs, referred to here as flow graph *behavior*, is also defined as an instance of an initialized model. We use transition label τ for internal transfer of control, $m_1 \,\mathsf{call}\, m_2$ for the invocation of method m_2 by method m_1 when method m_2 is provided by the program, $m_2 \;\mathsf{ret}\; m_1$ the corresponding return from the call, and label $m_1 \;\mathsf{caret}\; m_2$ for the (atomic) invocation of and return from an external method m_2 by method m_1.

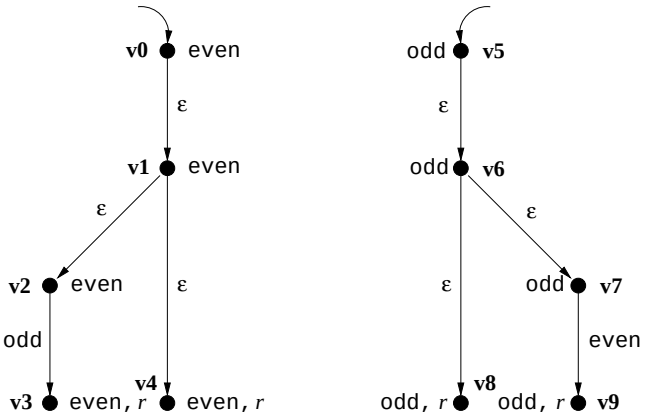


Fig. 2: Flow graph of EvenOdd

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

$$\text{[transfer]}\ (v,\sigma) \xrightarrow{\tau} (v',\sigma) \quad \text{if } m \in I^+,\ v \xrightarrow{\varepsilon}_m v',\ v \models \neg r$$

$$\text{[call]}\ (v_1,\sigma) \xrightarrow{m_1\ \mathsf{call}\ m_2} (v_2, v_1' \cdot \sigma) \quad \text{if } m_1, m_2 \in I^+, v_1 \xrightarrow{m_2}_{m_1} v_1', v_1 \models \neg r, v_2 \models m_2, v_2 \in E$$

$$\text{[ret]}\ (v_2, v_1 \cdot \sigma) \xrightarrow{m_2\ \mathsf{ret}\ m_1} (v_1,\sigma) \quad \text{if } m_1, m_2 \in I^+, v_2 \models m_2 \wedge r, v_1 \models m_1$$

$$\text{[caret]}\ (v_1,\sigma) \xrightarrow{m_1\ \mathsf{caret}\ m_2} (v_1',\sigma) \quad \text{if } m_1 \in I^+, m_2 \in I^-, v_1, v_1 \xrightarrow{m_2}_{m_1} v_1', v_1' \models m_1, v_1 \models \neg r$$

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

Notice that return transitions always hand back control to the caller of the method. Calls to external methods are modeled with `caret` transitions that jump immediately from the external method invocation to the corresponding return, without considering the intermediate behavior. This treatment of method calls is inspired by the temporal logic CaRet [1] mentioned in the introduction, and is convenient for specifying the local behavior of flow graphs. When writing global specifications, however, one has to be aware that in this way possible callbacks from external methods are not captured.

Example 4. Consider the flow graph from Example 3. An example run through its (branching, infinite–state) behavior, from an initial to a final state, is:

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

Now, consider just the method graph of method `even` as an open flow graph, having interface $(\{\texttt{even}\},\{\texttt{odd}\})$. The *local contribution* of method `even` to the above global behavior is the following run:

$$(v_0,\varepsilon)\xrightarrow{\tau}(v_1,\varepsilon)\xrightarrow{\tau}(v_2,\varepsilon)\xrightarrow{\texttt{even caret odd}}(v_3,\varepsilon)$$

Pushdown systems (PDS) and *Context Free Processes* (CFP) are alternative formalisms to express flow graph behavior (see e.g., [5]). We exploit this by using PDS model checking (concretely the tool MOPED [18]) and an own CFP model checker for verifying program behavior against temporal formulas [10].

4 Property Specification and Compositional Verification

In this section, we define the two main specification languages PROMOVER uses, namely *Linear–time Temporal Logic (LTL)* and *Safety Automata*, and introduce our compositional verification principles for both specification languages.

4.1 Property Specification

Safety properties can be expressed in a variety of formalisms. In this paper, we use two property specification languages: *safety LTL* which is the safety–fragment of *Linear–time Temporal Logic* (LTL) [23] that uses only the weak until–operator, and *Safety Automata* which are based on Schneider's *Security Automata* [27], but where states are additionally tagged with atomic propositions. Both specification languages demand a different treatment regarding verification. This subsection defines the syntax and semantics of the two specification languages, while the following one explains compositional verification for each case.

Linear–time Temporal Logic. One of the standard logics to express safety and liveness temporal properties is LTL. In our work, we focus on safety properties and therefore, we only use the safety fragment of LTL based on the weak version of until. The fragment is parameterized on a set of atomic propositions A as induced by a given flow graph $\mathcal{G}$, augmented with a special atomic proposition `entry` that holds at the entry nodes of $\mathcal{G}$.

Definition 5 (Safety LTL). *The formulae of* Safety LTL *are inductively defined by:*

$$\phi ::= p \mid \neg p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \mathtt{X}\,\phi \mid \mathtt{G}\,\phi \mid \phi_1\,\mathtt{W}\,\phi_2$$

where p ranges over $A \cup \{\texttt{entry}\}$. For convenience, we sometimes use $p \Rightarrow \phi$ to abbreviate $\neg p \vee \phi$.

Satisfaction on states $(\mathcal{M}_b, s) \models \phi$ for LTL formulae is defined in the standard fashion [30]: formula $\texttt{X}\,\phi$ holds of state s in model $\mathcal{M}_b$ if ϕ holds in the next state of every run starting in s; $\texttt{G}\,\phi$ holds if for every run starting in s, ϕ holds in all states of the run; and $\phi\ \texttt{W}\ \psi$ holds in s if for every run starting in s, either ϕ holds in all states of the run, or ψ holds in some state and ϕ holds in all previous states.

Example 5. Consider the global property of class `EvenOdd` in Figure 1 (where `!`, `&&`, `||`, and `->` are ASCII notations for $\neg$, $\wedge$, $\vee$, and $\Rightarrow$, respectively) and its intuitive meaning discussed in Example 1. Flow graph extraction and construction ensure that entry nodes are only accessible via calls; hence, if control starts and remains in method `even`, execution can be at an entry node only as the result of a self–call. The formula thus states that "if program execution starts in method `even`, method `even` is not called until method `odd` is reached", which coincides with the interpretation given in Example 1.

Internally, the verification machinery for local LTL formulae is based on the safety fragment of the modal μ-calculus (that is, excluding diamond modalities and least fixed point recursion). Safety LTL is somewhat less expressive than the latter and can be uniformly encoded in it [7]. This translation is implemented as part of ProMoVer. As a technical detail, the additional atomic proposition `entry` that can appear in LTL formulae is removed during the translation.

Safety Automata. Alternatively, safety properties can be specified by means of safety automata, which are closely related to the notion of security automata [27].

Definition 6 (Safety Automaton). *A* safety automaton $\mathcal{A}$ *is an instance of an initialized model, where the set of labels is* $L_b = \{m_1\ k\ m_2 \mid k \in \{\mathsf{call}, \mathsf{ret}\}, m_1, m_2 \in I^+\} \cup \{m_1\ \mathsf{caret}\ m_2 \mid m_1 \in I^+ \wedge m_2 \in I^-\} \cup \{\tau\}$ *and the set of atomic propositions is* A.

Notice that since a safety automaton is an instance of the general notion of initialized model, the composition of two safety automata $\mathcal{A}_1$ and $\mathcal{A}_2$ is defined as their disjoint union $\mathcal{A}_1 \uplus \mathcal{A}_2$.

If a safety automaton $\mathcal{A}$ is used for specifying a method specification, then it can be translated in a straightforward manner into a flow graph $FG(\mathcal{A})$ that simulates exactly those flow graphs that are simulated by $\mathcal{A}$. Safety automaton $\mathcal{A}$ *simulates* a flow graph $\mathcal{G}$ if $\mathcal{G} \leqslant FG(\mathcal{A})$ as initialized models, as defined in Definition 2 (extended to initialized models).

The language of safety automata is equally expressive as μ-calculus and thus safety automata can be translated into μ-calculus formulae.

Example 6. Consider the local specification of method `odd` in Example 1, expressing "method `odd` can only call method `even`, and after returning from the

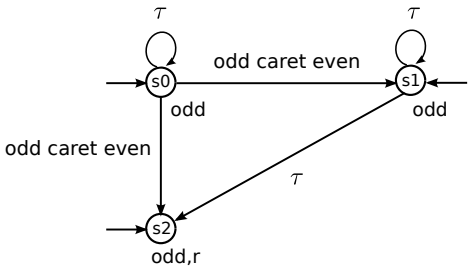


Fig. 3: Safety automaton for the local specification of method `odd`

call, no other method can be called". Figure 3 contains a graphical representation of this property.

The textual ASCII representation of the safety automaton is shown in Figure 1. In the ASCII representation, the `node` keyword defines a state of the automaton, followed by a list of comma–separated atomic propositions that hold in the state, while the `edge` keyword defines a transition of the automaton by starting state, target state, and the transition label, respectively. The atomic propositions `entry` and `ret` specify entry and return states, respectively, while label `tau` is the ASCII representation of τ.

Syntactic Sugar. Safety automata as defined above can become rather large in case of large interfaces. There are a variety of conventions one can use to facilitate a less verbose and more compact representation of an automaton. At present, we support negated labels to abbreviate that a particular action cannot be present on a transition between two states; for example, a label $\neg(a \ \mathsf{call} \ b)$ on a transition from an automaton state s_1 to state s_2 means that all labels from the label set L are present on the transition except for label $a \ \mathsf{call} \ b$. As another useful shorthand, it is often convenient to be able to express that the atomic proposition r may have any value in a particular state; for this we provide the "wild–card" atomic proposition $r*$.

Automata described with the above shorthands are easily translated into ordinary safety automata.

Example 7. The safety automaton from Figure 3 can be represented more compactly by the automaton illustrated in Figure 4. The latter automaton can be transformed (back) to the automaton of Figure 3 by duplicating state `s1` to states `s1` and `s2`, tagging only state `s2` with r, and eliminating all outgoing edges from state `s2`.

4.2 Compositional Verification

Next, we describe the compositional verification principles for the two specification languages. First, we describe compositional verification based on the construction of maximal flow graphs from the component's local specifications,

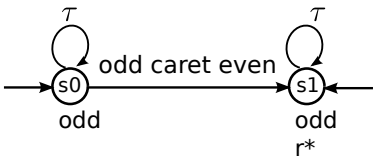


Fig. 4: Compact safety automaton for the local specification of method `odd`

when the latter are expressed in temporal logic: safety LTL, safety μ-calculus, or as modal equation systems (as defined by Larsen [20]). A modal equation system Σ is a finite set of defining equations of the shape $X = \phi_X$, where X is a propositional variable and ϕ_X is a formula of propositional modal logic without diamond modalities (recall that a modal formula $[l]\phi$ holds in a state s of a model if ϕ holds in all states accessible from s via transitions labeled with l). The defined variables X are pairwise distinct and bound in Σ, while all other variables are free. Its meaning is defined as its greatest solution. Modal equation systems are equivalent to the safety μ-calculus. In fact, we use this presentation of temporal properties in our maximal model construction and when automatically extracting local temporal specifications from method implementations (see Section 5).

The second part of this section discusses compositional verification when properties are expressed as safety automata.

Compositional Verification for Safety LTL. Our method for *algorithmic compositional verification* for LTL specifications is based on the construction of maximal flow graphs from component properties. For a given property ψ and interface I, consider the set of all flow graphs with interface I satisfying ψ. A *maximal flow graph* for ψ and I, denoted $\mathcal{M}ax(\psi, I)$, satisfies exactly those properties that hold for all members of the set. Thus, the maximal flow graph can be used as a representative of the set for the purpose of property verification. For details the reader is referred to [13].

For a system with k components, our principle of compositional verification based on maximal flow graphs can be presented as a proof rule with $k + 1$ premises.

$$\frac{\mathcal{G}_1 \models \psi_1 \;\; \cdots \;\; \mathcal{G}_k \models \psi_k \qquad \biguplus_{i=1,\ldots,k} \mathcal{M}ax(\psi_i, I_i) \models \phi}{\biguplus_{i=1,\ldots,k} \mathcal{G}_i \models \phi} \tag{1}$$

The rule states that the composition of components $\mathcal{G}_1 : I_1, ..., \mathcal{G}_k : I_k$ satisfies a global property ϕ if there are local properties ψ_i such that (i) each component $\mathcal{G}_i$ satisfies its local property ψ_i, and (ii) the composition of the k maximal flow graphs $\mathcal{M}ax(\psi_i, I_i)$ satisfies ϕ. This principle is proved *sound* and *complete* in [13].

In the context of ProMoVer, we consider individual program methods as components. If we instantiate the above compositional verification principle to procedure–modular verification, we obtain the verification tasks stated informally in Section 2 (where M is the set of program methods, with $k = |M|$, and ψ_i and $\mathcal{C}_i$ are the specification and the implementation of method m_i, respectively):

(i) **Checking $\mathcal{C}_i \models \psi_i$ for $i = 1, ..., k$:** For each method $m_i \in M$, (a) extract the method graph $\mathcal{G}_i$ from $\mathcal{C}_i$, and (b) model check $\mathcal{G}_i$ against ψ_i. For the latter, we exploit the fact that flow graphs are *Kripke structures*, and apply standard finite–state model checking.

(ii) **Checking $\biguplus_{i=1,...,k} \mathcal{M}ax(\psi_i, I_i) \models \phi$:** ($a$) Construct maximal flow graphs $\mathcal{M}ax(\psi_i, I_i)$ for all method specifications ψ_i and interfaces I_i, then (b) compose the graphs, resulting in flow graph $\mathcal{G}_{\mathcal{M}ax}$, and finally ($c$) model check $\mathcal{G}_{\mathcal{M}ax}$ against global property ϕ. For the latter, represent the behavior of $\mathcal{G}_{\mathcal{M}ax}$ as a PDS and use a standard PDS model checker.

Compositional Verification for Safety Automata. When all specifications are specified by safety automata, we check (i) whether the safety automaton of each method simulates its method graph, and (ii) whether the composition of the flow graphs of all local automata is simulated by the global automaton. Notice that in (ii) the flow graphs of the local safety automata serve as "maximal" flow graphs. This is due to that fact that, by definition, the safety automaton specification of a method simulates exactly those method graphs that satisfy the specification. Thus, the general compositional verification principle in this case for a system with k methods can be presented as the following proof rule.

$$\frac{\mathcal{G}_1 \leqslant \mathcal{A}_1 \ \cdots \ \mathcal{G}_k \leqslant \mathcal{A}_k \qquad \biguplus_{i=1,...,k} FG(\mathcal{A}_i) \leqslant \mathcal{A}}{\biguplus_{i=1,...,k} \mathcal{G}_i \leqslant \mathcal{A}} \tag{2}$$

The principle states that the composition of method graphs $\mathcal{G}_1 : I_1, ..., \mathcal{G}_k : I_k$ satisfies a global property expressed by a safety automaton $\mathcal{A}$ if there are local properties expressed by safety automata $\mathcal{A}_i$ such that (i) each method graph $\mathcal{G}_i$ is simulated by its local property $\mathcal{A}_i$, and (ii) the composition of the k flow graphs of the local safety automata $\mathcal{A}_i$ is simulated by $\mathcal{A}$. *Soundness* and *completeness* of this principle is established similarly as soundness and completeness of Principle 1 (in [13]).

In ProMoVer, for safety automata specifications, the verification tasks stated informally in Section 2, are achieved based on Principle 2 by:

(i) **Checking $\mathcal{C}_i \leqslant \mathcal{A}_i$ for $i = 1, ..., k$:** For each method $m_i \in M$, (a) extract the method graph $\mathcal{G}_i$ from $\mathcal{C}_i$, and (b) check that $\mathcal{G}_i$ is simulated by $\mathcal{A}_i$. For the latter, we exploit the fact that flow graphs and safety automata are initialized models, and check for simulation accordingly.

(ii) **Checking $\biguplus_{i=1,\ldots,k} FG(\mathcal{A}_i) \leqslant \mathcal{A}$:** ($a$) compose the flow graphs of the safety automata specifications of all methods, resulting in safety automaton FG_{comp}, and then (b) model check FG_{comp} against global automaton $\mathcal{A}$. For the latter, represent the behavior of FG_{comp} as a context free process, and use a CFP model checker (on the temporal formula translation of the automaton).

The two principles can be combined freely, so that local specifications and global properties can be written in either formalism. In task (i), if method m is specified in LTL, the flow graph extracted from method m is model checked against the specification, while if method m is specified with a safety automaton, simulation of the flow graph by the safety automaton is checked instead. In task (ii), maximal flow graphs are constructed for all methods with LTL specifications, and are then composed with the flow graphs of all safety automata specifications. Finally, if the global property is specified in LTL, the composition result is model checked against the property, while if the global property is specified by a safety automaton, the composition result is model checked against the automaton instead.

Example 8. Consider again the annotated Java program from Example 1. In the example, the global property and the local specification of method `even` are specified in LTL, while the local specification of method `odd` is given as a safety automaton. ProMoVer first extracts the method graphs of methods `even` and `odd`, denoted $\mathcal{G}_{even}$ and $\mathcal{G}_{odd}$, respectively. Next, ProMoVer checks $\mathcal{G}_{even} \models \psi_{even}$ and $\mathcal{G}_{odd} \leqslant \mathcal{A}_{odd}$. Independently, it constructs the maximal flow graph of method `even` denoted $\mathcal{M}ax(\psi_{even}, I_{even})$ and composes it with the flow graph of the safety automaton of method `odd` denoted FG_{odd} to obtain the flow graph $FG_{even-odd} = \mathcal{M}ax(\psi_{even}, I_{even}) \uplus FG_{odd}$. Finally, ProMoVer translates $FG_{even-odd}$ to a PDS and model checks the latter against the global LTL property.

5 The ProMoVer Tool

Next we describe the internals of ProMoVer. As mentioned above, ProMoVer essentially is a wrapper for cvpp [17], with extra features such as specification extraction, private method abstraction, a property specification library and support for proof reuse. All features are implemented in Python. ProMoVer can be tested via a web interface [28].

CVPP Wrapper. Figure 5 shows schematically how ProMoVer combines the individual cvpp tools. An annotated Java program, as exemplified in Section 2, is given as input. The *pre–processor* parses the annotations, using the Java Doclet API [9], and then passes properties and interfaces on to the different cvpp tools.

Task (i) first invokes the Analyzer tool described in [4] to extract the method graphs of the program. This tool builds on Sawja [15] to extract flow graphs from Java bytecode. Then our Graph tool is used. This implements several algorithms on flow graphs and safety automata, including composition $\uplus$

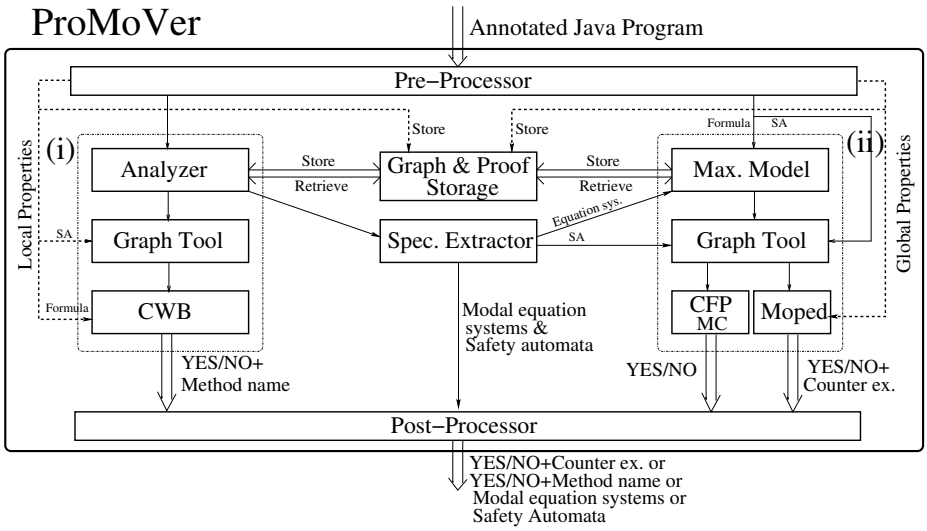


Fig. 5: Overview of ProMoVer and its underlying tool set

and translations of flow graphs and safety automata into different formats. Here the Graph tool is used to translate the flow graph of each method into a CCS model. These are then checked against the respective local method specifications using the *Concurrency Workbench* (cwb) [6]. If the specification is specified by LTL then it is translated to a μ-calculus formula and cwb is used to model check the CCS model against the formula. In case the specification is given in safety automaton, it is also translated into a CCS model and language inclusion is checked by cwb.

Task (*ii*) first constructs a maximal flow graph for every method specified with LTL by using the Maximal Model tool, and for methods specified with safety automata translates the specifications to flow graphs by the Graph tool. Then the Graph tool composes the generated flow graphs and converts the result into a PDS (for a global property expressed with LTL) or CFP (for a global property expressed as a safety automaton). Finally Moped [18] is used to model check the PDS against the LTL global property or CFP MC [10] is used to model check the CFP against the μ-calculus translation of the global safety automaton. The latter is a model checker implemented as part of the toolset.

The *post–processor* collects all model checking results and converts these into a user–understandable format. It only returns a positive result if all collected model checking tasks succeed. If one of the local model checking tasks fails, the name of the method that violates its specification is returned. If the global model checking task fails, for LTL global properties, a counterexample is provided by Moped and translated into a program execution and returned, however, for safety automata global properties, CFP MC does not provide a counterexample and therefore, no counterexample is returned.

Specification Extraction. To reduce the effort needed to write specifications, ProMoVer provides support to extract a specification from a given method implementation, resulting in the (over–approximated) order of method invocations for this method. The user might then want to remove some superfluous dependencies, in order not to be overly restrictive on possible evolution of the code.

ProMoVer extracts specifications in two different formats: modal equation systems and safety automata. Modal equation systems have the advantage that in cvpp they can serve directly as input for the construction of maximal flow graphs. On the other hand, the extracted safety automata specifications bypass the expensive maximal flow graph construction process, are often more intuitive, and can be modified graphically.

Consider again Figure 1. Specification extraction for method `odd` results in the following modal equation system (where `eps` is ASCII notation for ε, and `ff` denotes *false*):

```
@local_eq_prop: (X0){ X0  =  [even]X1 /\ [odd]ff  /\ [eps]X0;
                      X1  =  [odd]ff  /\ [even]ff /\ [eps]X1; }
```

The formula (which refers to the denotation of `X0` in the greatest solution of the equation system) essentially specifies that method `even` may be called at most once: initially `X0` holds, and method `even` may be called or an internal step (labeled `eps`) may be made. After calling `even`, `X1` should hold and only internal steps are allowed.

Using the specification extractor to extract the safety automaton specification for the same method results in the safety automaton depicted in Figure 3.

As a more involved example, consider the following method `m` together with its specification, extracted as a modal equation system:

```
@local_eq_prop:
(X0){ X0  =  [m4]ff /\ [m1]X1 /\ [m3]ff /\ [m2]ff /\ [m]ff /\ [eps]X0;
      X1  =  [m4]ff /\ [m1]ff /\ [m3]ff /\ [m2]X2 /\ [m]ff /\ [eps]X1;
      X2  =  [m4]X3 /\ [m1]ff /\ [m3]X4 /\ [m2]ff /\ [m]ff /\ [eps]X2;
      X3  =  [m4]ff /\ [m1]ff /\ [m3]ff /\ [m2]ff /\ [m]ff /\ [eps]X3;
      X4  =  [m4]ff /\ [m1]ff /\ [m3]ff /\ [m2]ff /\ [m]ff /\ [eps]X4;
    }

public void m() { int i = m1(); int j = m2();
                  if (i < j) { m3(); } else { m4(); } }
```

The formula captures that first only `m1` can be called, then only `m2`, and then either `m3` or `m4`, and no further calls can be made. Suppose that the order of invoking `m1` and `m2` is immaterial for this program. In that case a designer may choose to change the equations defining `X0` and `X1` to allow the two methods to be called in any order (whereas the defining equations for `X2` to `X4` remain unchanged):

```
X0  =  [m4]ff /\ [m1]X10 /\ [m3]ff /\ [m2]X11 /\ [m]ff /\ [eps]X0;
X10 =  [m4]ff /\ [m1]ff  /\ [m3]ff /\ [m2]X2  /\ [m]ff /\ [eps]X10;
X11 =  [m4]ff /\ [m1]X2  /\ [m3]ff /\ [m2]ff  /\ [m]ff /\ [eps]X11;
```

Using the specification extractor to extract the safety automaton specification of method `m` above will result in the following safety automaton, illustrated graphically in Figure 6.

```
node s1 m,entry      edge s1 s1 tau      edge s1 s2 m caret m1
node s2 m            edge s2 s2 tau      edge s2 s3 m caret m2
node s3 m            edge s3 s3 tau      edge s3 s4 m caret m3
node s4 m,r*         edge s4 s4 tau      edge s3 s5 m caret m4
node s5 m,r*         edge s5 s5 tau
```

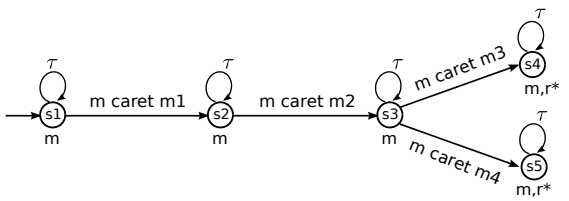


Fig. 6: Extracted safety automaton

As above, also the safety automaton can be relaxed for the case that the order in which the methods `m1` and `m2` are invoked is immaterial, as shown in Figure 7.

```
node s1 m,entry      edge s1 s1 tau      edge s1 s2_1 m caret m1
node s2_1 m          edge s2_1 s2_1 tau  edge s2_1 s3 m caret m2
node s2_2 m          edge s2_2 s2_2 tau  edge s1 s2_2 m caret m2
node s3 m            edge s3 s3 tau      edge s2_2 s3 m caret m1
node s4 m,r*         edge s4 s4 tau      edge s3 s4 m caret m3
node s5 m,r*         edge s5 s5 tau      edge s3 s5 m caret m4
```

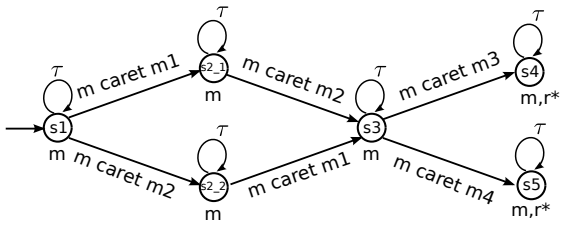


Fig. 7: Relaxed safety automaton

Private Method Abstraction. Since private methods are used as a means of implementation for public methods, at the flow graph level, all calls to private methods can be inlined into the flow graph of the public methods. The resulting method flow graphs thus only describe the public behavior, and users only have to specify the public methods. For details the reader is referred to [13].

Property Specification Library. ProMoVer's web interface provides a collection of pre–formalized global properties. These describe platform–specific security properties, restricting calls to API methods. Currently, the library contains several Java Card and voting system properties.

Proof Storage and Reuse. All extracted method flow graphs and constructed maximal flow graphs are stored when a program is verified by ProMoVer. If later the implementation of method m changes, a new method flow graph is extracted and checked against m's local specification. If m's local specification ϕ_m changes, the existing flow graph of method m is model checked against ϕ_m. In addition a new maximal flow graph for m is constructed from ϕ_m. This is composed with the other maximal flow graphs (recovered from storage), and the composed flow graph is model checked against the global property.

6 Experimental Results for ProMoVer

We use ProMoVer to verify standard control flow safety properties on a number of applications from two application domains where code evolution is important, namely Java Card and web-based applications.

6.1 Experiments on Java Card Applications

Java Card is one of the leading interoperable platforms for smart cards. Many smart card applications are security–critical. As mentioned above, for platforms such as Java Card, collections of control flow safety properties exist that programs should adhere to in order to provide minimal security requirements. We focus on such a property of the Java Card transaction mechanism. This mechanism ensures that data remains consistent upon power loss, however, careful use of it sometimes demands that certain methods are not used within a transaction. We show how this global safety property can be expressed in our setting, and be verified with ProMoVer for several applications, where we apply specification extraction to annotate the public methods of the applications.

As a side remark, control flow of Java Card programs might be different from control flow of a standard Java application, for example the Java Card firewall can cause an object field to raise an exception. Handling these differences correctly is an issue for the control flow graph extraction algorithm. However, for the properties and case study discussed here, this difference in control flow is not relevant, and we do not discuss it further here.

Application	#LoC	#Methods (Public)	#Calls (Relevant)
`AccountAccessor`	190	9 (7)	38 (4)
`TransitApplet`	918	18 (5)	106 (5)
`JavaPurse`	884	19 (9)	190 (25)

Table 1: Applications details

The Java Card Transaction Mechanism. Smart cards have two types of writable memory, *persistent memory* (EEPROM or Flash) and *transient memory* (RAM). Transient memory needs constant power supply to store information, while persistent memory can store data without power. Smart cards do not have their own power supply; they depend on the external source that comes from the card reader device. Therefore, a problem known as *card tear* may occur: a power loss when the card is suddenly disconnected from the card reader. If a card tear occurs in the middle of updating data from transient to persistent memory, the data stored in transient memory is lost and may cause the smart card to be in an inconsistent state.

To prevent this, the *transaction mechanism* is provided. It can be used to ensure that several updates are executed as a single *atomic* operation, *i.e.*, either all updates are performed or none. The mechanism is provided through methods `beginTransaction` for beginning a transaction, `commitTransaction` for ending a transaction with performed updates, and `abortTransaction` for ending a transaction with discarded updates [14] – all declared in class `JCSystem` of the Java Card API.

However, the Java Card API also contains some *non–atomic* methods that are better not used when a transaction is in progress. Notably, the class `javacard.framework.Util` that provides functionality to store and update byte arrays, contains methods `arrayCopyNonAtomic` and `arrayFillNonAtomic`. Careful use of the transaction mechanism can require that these methods should not be used within a transaction. We use ProMoVer to verify that applications comply with this *Transaction Policy*.

The Applications. For this experiment we use several public examples of Java Card applications. All are realistic e-commerce applications developed by Sun Microsystems to demonstrate the use of the Java Card environment for developing e-commerce applications. `AccountAccessor` is an application to keep track of account information. It is to be used by a wireless device connected via a network service. It contains methods to look up and modify the account balance. `TransitApplet` implements the on–card part of a system that connects to an authenticated terminal and provides account information and operations to modify the account balance. `JavaPurse` is a smart card electronic purse application providing secure money transfers. It contains a balance record denoting the user's current and maximum credits, and methods to initialize, perform and complete a secure transaction. Further, it also contains methods to update in-

formation related to a loyalty program, and to validate and update the values of transactions, balance and PIN code.

Table 1 shows information about the size, number of methods (total and public), and number of method invocations (total and relevant for the global property) of these applications.

Specification of the Transaction Policy. As discussed above, we want to ensure formally that the non-atomic methods `arrayCopyNonAtomic` and `arrayFillNonAtomic` are not invoked within a transaction. Hence, applications have to adhere to the following global control flow safety property:

> In every program execution, after a transaction begins, methods `arrayCopyNonAtomic` and `arrayFillNonAtomic` are not called until the transaction ends.

This safety property can be expressed formally with the following LTL formula:

$$\mathtt{G}\,(\mathtt{beginTransaction} \Rightarrow ((\neg\mathtt{arrayCopyNonAtomic} \wedge \neg\mathtt{arrayFillNonAtomic})\,\mathtt{W}\,\mathtt{commitTransaction}))$$

The property could also have been specified, though more verbosely, as a safety automaton.

Local Method Specifications. In order to compare the efficiency of verification for the different formalisms for writing local specifications, we annotated the methods of each application once in LTL and once with safety automata. For this we used the assistance of the specification extraction facility of ProMoVer.

The specification extractor is used to obtain local specifications for every public method, either as an equation system or as a safety automaton. The extracted specifications describe the actual order of method invocations in the code. We then inspect the specifications for immaterial orderings and remove these, with the intention that local method specifications should only restrict unwanted sequences of method calls made from within the specified method.

Writing specifications abstractly allows for possible evolution of the method implementations. Comparing the two formalisms, it can be observed that using temporal logic allows in general for more compact specifications, since only explicitly prohibited method invocations have to be mentioned.

Verification Results. After annotating the applications with global properties and local specifications, ProMoVer extracts the flow graphs of the applications and partitions these into the individual method graphs to verify adherence to the local specifications. Further, for applications with local specifications given in LTL, the maximal method graphs are constructed from the specifications, and their composition is verified *w.r.t.* the global property above. For applications with local specifications given as safety automata, the corresponding flow graphs of the automata are composed and verified *w.r.t.* the global property.

The statistics for these verifications are summarized in Table 2 and 3. The tables show: the time spent by the pre–processor (PPT) and the graph extractor

Application	PPT	GE	#NEF	LMC	MFC	#NMF	GMC	TT
`AccountAccessor`	1.4	3.8	435	0.5	0.7	20	0.9	8.7
`TransitApplet`	1.4	4.7	897	0.5	0.9	30	0.9	13.2
`JavaPurse`	1.5	6.5	1543	0.5	13.0	48	1.1	22.5

Table 2: Verification Results with LTL Local Specifications

Application	PPT	GE	#NEF	LMC	GMC	TT
`AccountAccessor`	1.4	3.8	435	0.6	0.9	8.1
`TransitApplet`	1.4	4.7	897	4.0	0.9	12.2
`JavaPurse`	1.5	6.5	1543	4.8	1.0	14.8

Table 3: Verification Results with Safety Automata Local Specifications

(GE) (all times here and below are in seconds), the number of nodes in the extracted flow graphs (#NEF), the time spent for local model checking (LMC) and for constructing maximal flow graphs (MFC), the number of nodes in the maximal flow graph composition (#NMF), the time spent for global model checking (GMC), and the total time spent for the whole verification task including conversions between formats and post–processing (TT). All results are obtained on a SUN SPARC machine. Notice that the pre–processing time (PPT), the graph extraction time (GE), and the number of nodes in the extracted flow graphs (#NEF) are the same for applications with local specifications given in LTL and safety automata, but in the case of safety automata the expensive process of maximal flow graph construction is bypassed.

As can be observed from the tables, local model checking takes longer for applications with local specifications given as safety automata. This is due to the higher verbosity of local specifications with automata, compared with temporal logic formulae, as discussed above. However, the increased local model checking time is compensated for by the translation from automata into method graphs, which just renames transition labels and is thus much less expensive than the corresponding maximal model construction for temporal logic specifications.

Proof Reuse. We also evaluate experimentally the advantages of exploiting the proof storage and reuse mechanism. After the first verification, when all method and maximal flow graphs have been stored, we changed, for each application, once the source code and once the local specification of a public method, and used ProMoVer to re–verify the applications.

The changes in the source code imitate a typical code evolution scenario, where a method's body is changed, for example, for the purpose of maintenance. The changes in the local specifications are motivated by the scenario where the (automatically extracted) specifications are weakened to support code evolution.

The results of proof reuse are shown in Table 4. The table shows: maximal flow graph construction time (MFC), the time spent by ProMoVer to re–verify

	Code Change		Local Specification Change		
Application	New TT	% TT	MFC	New TT	% TT
`AccountAccessor`	6.0	68	0.1	4.6	52
`TransitApplet`	7.2	54	0.1	5.0	37
`JavaPurse`	9.0	40	0.1	5.4	24

Table 4: Proof Reuse Results

the program after the change (new TT), and its percentage in relation with the original verification time (%TT). The numbers indicate that proof reuse can significantly reduce the verification time, especially for larger applications.

6.2 Experiments on a Web Application

Web applications are client–server programs intended to be used over the Internet. Typically, clients are *web browsers* and servers are *web servers*. Such web applications are of major importance in the ICT business, and therefore it is crucial to check that they function correctly, without any unexpected errors.

To minimize errors, various coding standards exist that components of web applications should respect. Based on these standards, we identify several requirements for database connections and transactions of the *Java Enterprise* platform that can be expressed as control flow safety properties. We show how ProMoVer is used to verify such control flow database connection properties in the presence of code evolution. Concretely, we verify the *Single DataBase Connection Policy* for an incomplete and prototype version of the Sail–Web application (both property and application are discussed in more detail below). First, we verify the incomplete program with the specifications of the missing components. Later, we import the missing code from the prototype into the incomplete code and re–verify the program. By this we mimic the code evolution scenario discussed above and how it is supported by ProMoVer.

Java Enterprise Platform (J2EE). J2EE is a popular platform to develop Java web applications. It provides an API and specification of the runtime environment to develop and run typical enterprise applications. In J2EE, a web application consists of a set of components running on a web server. These components are typically used by the web server to extend its capabilities for generating responses to clients' requests.

A commonly used technology to develop such components is *Java Servlets*. Technically, servlets are Java classes that conform to the Java–Servlet API model. They may be used by developers to provide web–pages containing dynamic contents (e.g., HTML or XML) using the Java platform. Servlets are typically invoked via the methods `doPost` and `doGet`. The web server creates instances of the servlets at boot time and maintains these objects throughout the execution. When a request arrives from a client, the web server assigns a thread from a thread–pool to the request and forwards the request to the `doPost` or

`doGet` methods of the suitable servlet. The servlet computes a response for the request and returns it back to the web server. Then, this response is sent back to the client and the allocated thread is returned back to the thread–pool.

Web servers use multi–threading to be able to respond to simultaneous requests; however, each request is handled by a single thread. Hence, control flow properties for processing a single request can be analyzed in a non–concurrent setting.

J2EE Database Connection. Web applications often use databases to manipulate data and store information. For example, almost all web applications that provide support for user accounts store user information (such as user name and password) in a database.

Typical examples of control flow properties for database connections are the *safe database transaction policy* that states that "a database transaction should be either committed or rolled–back if an exception is raised", and the *database connection policy* that states that "only a single database connection should be created for each request and it should be properly closed". In the remainder of this section, we focus on the second property. The first property can be expressed and verified similarly to the Java Card transaction policy presented above, and therefore we do not discuss its verification here.

To understand why the *database connection policy* is important, one should realize that each database system is capable of handling a limited number of simultaneous connections only. Therefore, if a single request opens more than one connection to a database, it is using these limited resources inefficiently. Moreover, such a practice significantly increases the likelihood of coding–errors caused by not closing the open connections properly. Therefore, the database connection policy demands that web applications obtain only a single database connection per request, and moreover, that this connection is closed before the assigned thread is returned back to the pool.

Various strategies and frameworks exist that ensure that the policy is respected, such as using filters or frameworks like JBoss Seam and Spring. However, many web programmers do not use any of these facilities. Therefore, it is highly desirable to have a tool that can check such properties of web applications.

Formal Specification of the Single Database Connection Policy. If no special framework is used, Java applications typically communicate with a database via the Java DataBase Connectivity (JDBC) API. In this API, the methods `java.sql.DriverManager.getConnection` and `java.sql.Connection.close` are used to create and close database connections, respectively. Therefore, in order to check the database connection property explained above, we check the absence of consecutive calls to the method `java.sql.DriverManager.getConnection` unless the method `java.sql.Connection.close` is called in between.

More precisely, this means that applications should respect the following global control flow safety property:

Sail-Web App.	#LoC	#Classes (Servlets)	#Public Methods
Limited Package	3038	20 (16)	28
Extended Package	10844	32 (28)	94

Table 5: Sail-Web application details

> In every thread execution, after a connection to a database is created, the method `java.sql.DriverManager.getConnection` is not called until the connection is closed.

This safety property can be formally expressed by the following safety LTL formula:

```
G (p.DriverManager.getConnection ⇒
    X (¬p.DriverManager.getConnection W p.Connection.close))
```

where `p` abbreviates the `java.sql` package.

The Sail–Web Application. For our experiments, we use the Sail–Web (Scalable Architecture for Interactive Learning on the Web) application, which is available in Google Codes [25]. Sail–Web is an ongoing project that aims at developing a web–based content management system for interactive learning. This application uses a MySql database through the JDBC API to manipulate data. The application is divided into two separate packages, here called *limited* and *complete*. The complete package is an extended version of the limited one, supporting several additional features.

Table 5 shows information about size, number of classes (total and servlets), and number of public methods of the limited package and its extension with some features imported from the complete one. The extended package includes 12 more classes, here called *additional classes*. These classes extend the limited package by adding new features such as file management, URL connection, and security utilities. We begin our verification experiment with the code of the limited package, with additional annotations specifying the control flow of the methods of additional classes. This resembles systems with unavailable code, e.g., mobile code. Then, to imitate the code evolution scenario, we import the code of the additional classes into the limited package (which forms extended package) and re–verify the program.

Focusing on the database connection policy, private methods `createConnection` and `shutdown` of the servlets are used to create and close database connections, respectively. The code of these methods is shown in Figure 8.

These two methods are invoked by the `doGet` and `doPost` methods of servlets. As an example, the code of method `doGet` of class `VLEGetAnnotations` is shown in Figure 9. Methods `doGet` and `doPost` of other servlets use similar code to respond to the requests. Method `getData` is a private method to process requests; it has a different implementation in each servlet.

```
private static void createConnection() {
  try {
    //create a connection to the mysql db
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    conn = DriverManager.getConnection(DBurl, "sailuser", "sailpass");
  } catch (Exception ex) { ex.printStackTrace(); }
}

private static void shutdown() {
  try {
    conn.close();
  } catch(SQLException ex) { ex.printStackTrace(); }
}
```

Fig. 8: The private methods to create and close database connections

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
                  throws ServletException, IOException {
  createConnection();
  getData(request, response);
  shutdown();
}
```

Fig. 9: The code of method `doGet` of `VLEGetAnnotations` class

As explained above, the web server invokes the objects of the servlets based on the input request. We have modeled the behaviour of the web server by implementing a method that iteratively forwards random requests to random Servlets in a loop. This method is called `dispatch`.

Verification Results. We used the specification extractor to extract safety automata specifications of the methods of the Sail–Web application. The extracted safety automata represent the actual order of method invocations in the program. As mentioned above, we also annotated the specifications of the methods of the additional classes into the application and used these for verification of the global safety control flow property expressing the database single connection policy. ProMoVer constructs maximal models of the annotated specifications, combines these with the extracted specifications into a PDS and model checks the result against the global property. The statistics for the verification are given in the first row of Table 6. In the table, we show the time spent by the pre–processor (PPT), graph extractor (GE), local model checking (LMC), maximal flow graph construction (MFC), global model checking (GMC), and the whole verification (TT). Notice that in this version of the program local model checking is not used because the local specifications are extracted from the code and need not be checked. The verification result is "NO" and the following counterexample execution in the form of a program behaviour is returned[2].

[2] To simplify the presentation, the package names are removed from the configurations.

Sail-Web App.	PPT	CG	LMC	MFC	GMC	TT
Limited Package	43	19	–	8	1	71
Limited Package (with improvements)	2	19	–	–	1	22
Extended Package	–	–	32	–	–	32

Table 6: Verification results of the Sail-Web application

...
(`dispatch`, ε)
`dispatch call VLEGetAnnotations.doGet`
(`VLEGetAnnotations.doGet`, `dispatch`)
`VLEGetAnnotations.doGet caret getConnection`
(`VLEGetAnnotations.doGet`, `dispatch`)
`VLEGetAnnotations.doGet call VLEGetAnnotations.getData`
(`VLEGetAnnotations.getData`, `VLEGetAnnotations.doGet . dispatch`)
`VLEGetAnnotations.getData ret VLEGetAnnotations.doGet` exp
(`VLEGetAnnotations.doGet`, `dispatch`)
`VLEGetAnnotations.doGet ret dispatch` exp
(`dispatch`, ε)
`dispatch call VLEPostAnnotations.doPost`
(`VLEPostAnnotations.doPost`, `dispatch`)
`VLEPostAnnotations.doPost caret getConnection`
...

where the exceptional transitions are labeled by *exp*.

The counterexample shows an execution starting in method `dispatch` that results in two simultaneous connections to the database. The reason is that after creating the first connection, if an unhandled runtime exception (e.g., NullPointerException) is raised in method `getData` of class `VLEGetAnnotations`, then the normal execution path of the program changes. In the counterexample, the first unhandled exception in method `VLEGetAnnotations.getData` brings the program pointer back to method `VLEGetAnnotations.doGet`, and then this method propagates the exception to method `dispatch`. Usually in these situations, the web server sends the stack trace to the client and continues responding to other requests. While the database connection remains open, the next request arrives and opens a second connection by calling method `VLEPostAnnotations.doPost`.

We eliminate the behaviour in the counterexample by changing the `doGet` method of class `VLEGetAnnotations` to the code shown in Figure 10. In the new implementation of method `doGet` we added a try–catch block to catch any exception that may be raised during the execution of method `getData` and to close the connection to the database.

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
                  throws ServletException, IOException {
  createConnection();
  try {
    getData(request, response);
  } catch (Exception ex) { }
  shutdown();
}
```

Fig. 10: The new implementation of method `doGet` of `VLEGetAnnotations` class

We use ProMoVer to re–verify the new code. ProMoVer detects the small change and therefore only re-extracts the specification of method `doGet` of class `VLEGetAnnotations`. It uses this new specification to construct a new PDS and to perform the global model checking. The statistics of the verification are shown in the second row of Table 6. Again, the result of verification is "NO". The provided counterexample is analogous to the previous one; this time, however, the first connection is obtained in method `VLEPostAnnotations.doPost` and the second one in `VLEPostFlag.doPost`. This shows that the same problem of unhandled runtime exceptions exists in the `VLEPostAnnotations.doPost` method, too. In fact, we realized that the same problem exists in some other classes as well. After changing all of them, we use ProMoVer to verify the program, and finally the verification result is "YES".

Code Evolution Scenario. As mentioned above, the Sail–Web project is an ongoing project divided into two main packages. One of the packages includes more features and utilities that probably will be imported into the smaller package in the future. We act proactively and import the missing features and utilities, which we have specified before, into the limited package and form the so called extended package. As mentioned above, this new part consists of 12 servlet classes (called additional classes), which are implemented by 66 public methods with 4806 lines of Java code.

We now use ProMoVer to verify the code of the extended package. The proof storage and reuse mechanism detects the new code, and for each method of the additional classes, checks that its implementation matches the corresponding specification. As shown in the third row of Table 6, the verification takes 32 seconds. This time is spent for local model checking only. The result shows that all new methods respect their specifications. Therefore the verification result is "YES".

7 Conclusion

This paper describes ProMoVer, a tool that supports automatic procedure–modular verification of control flow safety properties of sequences of method invocations. It essentially implements a particular verification scenario for the

CVPP tool set that supports compositional verification of programs with procedures [13]. ProMoVer takes as input a Java program annotated with temporal correctness assertions. The assertions can be written in different specification formalisms. Currently LTL and safety automata are supported.

Modularity is understood here as the relativization of global program correctness properties on the correctness of its components. This is seen as the key to program verification in the presence of static and/or dynamic variability due to code evolution, code customization for many users such as in software product lines (as illustrated in [26]), or as yet unknown or unavailable code such as mobile code. We illustrate two important points: (*i*) temporal safety properties provide a meaningful abstraction for individual methods; and (*ii*) procedure–modular verification of temporal safety properties can be performed automatically. Different specification formalisms can be used to specify those temporal safety properties. Moreover, ProMoVer implements a mechanism for proof storage and reuse, so that only relevant parts have to be reverified after a system change. This makes the verification method advocated by ProMoVer suitable to be used in a context where systems evolve frequently, as is the case e.g., for mobile code. The modularity of the verification allows an independent evolution of the implementations of the individual methods, only requiring the re–establishment of their local correctness.

We believe that writing properties at the procedure–level is intuitive for a programmer. Still, to decrease the effort of annotating programs, we provide support for specification extraction in the case of post–hoc specification of already implemented methods, an inlining–based private method abstraction that requires only public methods to be specified, and a library of standard global safety properties.

Experiments with realistic Java Card and web–based applications show that useful safety properties of such programs can be conveniently expressed in a light–weight notation and verified automatically with ProMoVer. Moreover, proof storage and reuse provide appropriate support for the modular nature of the verification work: local changes in the specification or code require only local re–verification, with significant reduction in verification time.

The addition of safety automata as a specification formalisms has proven to be convenient, and moreover, it also results in more efficient maximal flow graph construction. Still, some issues remain to be resolved in order to increase the utility of ProMoVer. In the future, we plan to also experiment with other temporal logics and notations, or to use patterns to abbreviate common specification idioms. The tool set will be extended with further translations into the underlying uniform logic, which is currently the safety fragment of the modal μ-calculus.

Many important safety properties require program *data* to be taken into account. As a first step towards handling data, work has begun on extending our verification framework and tool set to Boolean programs. We are also currently investigating how to generalize our method for the program model of Rot *et al.* that models object references in the presence of unbounded object creation [24].

Acknowledgments We are indebted to Wojciech Mostowski, Erik Poll and Roberto Guanciale for their help in finding suitable case studies, to Afshin Amighi and Pedro de Carvalho Gomes for helping with the implementation of CVPP and ProMoVer, and to Stefan Schwoon for adapting the input language of Moped to our needs.

References

1. R. Alur, M. Arenas, P. Barcelo, K. Etessami, N. Immerman, and L. Libkin. First-order and temporal logics for nested words. In *Logic in Computer Science (LICS '07)*, pages 151–160, Washington, DC, USA, 2007. IEEE Computer Society.
2. R. Alur and S. Chaudhuri. Temporal reasoning for procedural programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI '10)*, volume 5944 of *LNCS*, pages 45–60. Springer, 2010.
3. R. Alur, K. Etessami, and P. Madhusudan. A temporal logic for nested calls and returns. In *Tools and Algorithms for the Analysis and Construction of Software (TACAS '04)*, volume 2998 of *LNCS*, pages 467–481. Springer, 2004.
4. A. Amighi, P. de Carvalho Gomes, D. Gurov, and M. Huisman. Sound control-flow graph extraction for Java programs with exceptions. In *Software Engineering and Formal Methods (SEFM '12)*, volume 7504 of *LNCS*, pages 33–47, 2012. QC 20121213.
5. O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 545–623. North Holland, 2000.
6. R. Cleaveland, J. Parrow, and B. Steffen. A semantics based verification tool for finite state systems. In *International Symposium on Protocol Specification, Testing and Verification*, pages 287–302. North-Holland Publishing Co., 1990.
7. M. Dam. CTL* and ECTL* as fragments of the modal μ-calculus. In *Colloquium on Trees in Algebra and Programming, (CAAP '92)*, volume 581 of *LNCS*, pages 145–164. Springer, 1992.
8. M. Das, S. Lerner, and M. Seigle. ESP: Path–sensitive program verification in polynomial time. In *Programming Language Design and Implementation (PLDI '02)*, pages 57–68. ACM, 2002.
9. Doclet overview. `http://java.sun.com/j2se/1.3/docs/tooldocs/javadoc/overview.html`.
10. N. Gawell. Automatic verification of applet interaction properties. Master's thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2009. Ref.: TRITA-CSC-E 2009:128.
11. M. Goldman and S. Katz. MAVEN: Modular aspect verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '07)*, volume 4424 of *LNCS*, pages 308–322. Springer, 2007.
12. D. Gurov and M. Huisman. Reducing behavioural to structural properties of programs with procedures. In *Verification, Model Checking, and Abstract Interpretation (VMCAI '09)*, volume 5403 of *LNCS*, pages 136–150. Springer, 2009.
13. D. Gurov, M. Huisman, and C. Sprenger. Compositional verification of sequential programs with procedures. *Information and Computation*, 206(7):840–868, 2008.
14. E. Hubbers and E. Poll. Transactions and non-atomic API methods in Java Card: specification ambiguity and strange implementation behaviours. Technical Report NIII-R0438, Radboud University Nijmegen, 2004.

15. L. Hubert, N. Barré, F. Besson, D. Demange, T. Jensen, V. Monfort, D. Pichardie, and T. Turpin. Sawja: Static Analysis Workshop for Java. In *Formal Verification of Object–Oriented Software (FoVeOOS '10)*, volume 6528 of *LNCS*. Springer, 2010.
16. M. Huisman, I. Aktug, and D. Gurov. Program models for compositional verification. In *International Conference on Formal Engineering Methods (ICFEM '08)*, volume 5256 of *LNCS*, pages 147–166. Springer, 2008.
17. M. Huisman and D. Gurov. CVPP: A tool set for compositonal verification of control-flow safety properties. In *Formal Verification of Object–Oriented Software (FoVeOOS '10)*, volume 6528 of *LNCS*, pages 107–121. Springer, 2010.
18. S. Kiefer, S. Schwoon, and D. Suwimonteerabuth. Moped - a model-checker for pushdown systems. http://www.informatik.uni-stuttgart.de/fmi/szs/tools/moped/.
19. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
20. K. Larsen. Modal specifications. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 232–246. Springer, 1989.
21. G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual*, Feb. 2007. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.
22. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2002.
23. A. Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science (FOCS '77)*, pages 46–57. IEEE Computer Society, 1977.
24. J. Rot, F. de Boer, and M. Bonsangue. A pushdown system representation for unbounded object creation. In *Informal pre-proceedings of Formal Verification of Object–Oriented Software (FoVeOOS '10)*, 2010.
25. Sail-web application, 2012. `https://code.google.com/p/sail-web/`.
26. I. Schaefer, D. Gurov, and S. Soleimanifard. Compositional algorithmic verification of software product lines. In *Formal Methods for Components and Objects (FMCO '10)*, volume 6957 of *LNCS*, pages 184–203. Springer, 2011.
27. F. B. Schneider. Enforceable security policies. *ACM Trans. Infinite Systems Security*, 3(1):30–50, 2000.
28. S. Soleimanifard, D. Gurov, and M. Huisman. ProMoVer web interface. `http://www.csc.kth.se/~siavashs/ProMoVer`.
29. S. Soleimanifard, D. Gurov, and M. Huisman. ProMoVer: Modular verification of temporal safety properties. In G. Barthe, A. Pardo, and G. Schneider, editors, *Software Engineering and Formal Methods (SEFM '11)*, volume 7041 of *LNCS*, pages 366–381. Springer, 2011.
30. C. Stirling. *Modal and Temporal Logics of Processes*. Springer, 2001.

Appendix B

Paper II: Model Mining and Efficient Verification of Software Product Lines

Model Mining and Efficient Verification of Software Product Lines

Siavash Soleimanifard[1], Dilian Gurov[1], Bjarte M. Østvold[2], and Minko Markov[3]

[1] Royal Institute of Technology, Stockholm, Sweden
{siavashs,dilian}@csc.kth.se
[2] Norwegian Computing Center, Oslo, Norway
bjarte@nr.no
[3] University of Sofia, Sofia, Bulgaria
minkom@fmi.uni-sofia.bg

Abstract. Software product line modeling aims at capturing a set of software products in an economic yet meaningful way. We introduce a class of variability models that capture the sharing between the software artifacts forming the products of a software product line (SPL) in a hierarchical fashion, in terms of *commonalities* and *orthogonalities*. Such models are useful when analyzing and verifying *all* products of an SPL, since they provide a scheme for divide-and-conquer-style decomposition of the analysis or verification problem at hand. We define an abstract class of SPLs for which variability models can be constructed that are optimal w.r.t. the chosen representation of sharing. We show how the constructed models can be fed into a previously developed algorithmic technique for compositional verification of control-flow temporal safety properties, so that the properties to be verified are iteratively decomposed into simpler ones over orthogonal parts of the SPL, and are not re-verified over the shared parts. We provide tool support for our technique, and evaluate our tool on a small but realistic SPL of cash desks.

1 Introduction

Software Product Lines. System diversity is prevalent in modern software. In order to comply with the varying requirements of a potentially large number of customers, software systems often exist simultaneously in many different variants. *Software product line* engineering aims at planning for and developing a family of system variants through managed reuse, in order to decrease time to market and improve software quality [36].

The variability of the different products in a software product line can be represented at different levels [11]. *Problem-space variability* describes product variation in terms of so-called features, that is user-visible product characteristics. The set of valid feature configurations defines the set of possible products. However, features are not necessarily related to the actual *artifacts* that are used to realize the products. Problem-space variability based on features is at

the requirements level, while *solution-space variability* is at the design and implementation level. Solution-space variability describes product variation in terms of artifacts that are used to build the actual products of the product line.

In this paper, we aim to capture solution-space variability in terms of software artifacts that implement various functionalities. In the present context, an artifact is a software component at a suitable level of granularity, such as a Java method, a class, or a module.

Hierarchical Modeling. In order to describe the solution space variability in a software product line, we propose a *hierarchical variability model*, or HVM. Such a model represents, in a hierarchical manner, the artifacts that are common to all products, and the artifact variations that can occur between different products. On each hierarchical level, there is a *common set* of artifacts that represent parts shared by all products, while *variation points* represent parts that can vary from product to product. Every variation point is associated with a set of variants that represents choices for realizing the variation point in different ways. A variant is itself represented by a hierarchical variability model, potentially introducing a new level of hierarchy. A *product* described by a hierarchical variability model is obtained by selecting a variant at every variation point. The *product line*, or *family*, described by the model is the set of all its products.

Consider as an example a product line of a web-based social network application, shown graphically in Figure 1. This social network is to be used for audio or video sharing and communication between users. It provides basic user account support, content sharing facilities, and two communication environments, namely chat and email. The commonality of all social networks of the product line is that they all have user account support. This is modeled by the common artifact *userAccount* at the first level of hierarchy. The social networks, however, differ in the content they allow to share and the facilities they provide for communication: some allow only audio sharing, while others only allow video. In the model, this is represented by the variation point *content* (depicted as a diamond node) with the variants *Audio* and *Video* at the second level of hierarchy. Similarly, users of the social networks can either communicate via email or a chat system. Common for all social networks supporting chat is the text chat functionality, which only allows text exchange between users while at a third level of hierarchy, two alternative chat systems are realized, namely *AudioChat* and *VideoChat*. This hierarchical variability model gives rise to 6 products, corresponding to the 6 ways of resolving the variabilities.

Analysis and Verification of Software Product Lines. For any given program analysis, analyzing all products of a family individually may be infeasible for larger families. However, the number of products generated by a hierarchical variability model is at worst exponential in the size of the model; or equivalently, the model can be *exponentially more succinct* than the family. Exploiting the artifact commonalities at the different levels of hierarchy—as revealed by the model—is the key to achieving *scalability* of any analysis.

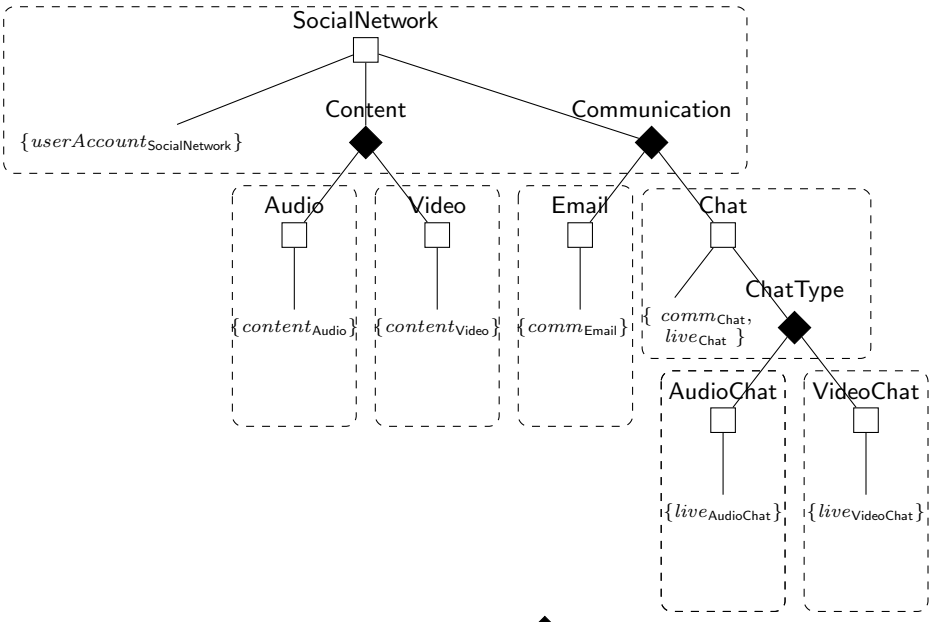


Fig. 1. The Social Network hierarchical variability model.

Factoring out common artifacts naturally reduces redundancy in the analysis: At variants with more than one variation point, the analysis problem is *decomposed* into simpler subproblems, as long as variation points at the same level of hierarchy expose *orthogonality*, while at variation points with more than one variant, the same problem is solved *independently* for each variant, as a case analysis, as long as variants at the same level of hierarchy expose *alternative* implementations. Thus, a hierarchical variability model can be viewed as a *divide-and-conquer scheme* for decomposing and splitting an analysis over a family of products.

In this paper, we develop the above idea by relativizing the correctness of the properties that are to hold for all products of a family on local specifications associated with the variation points. Thus, the number of verification tasks is reduced to the number of *regions* in the model (indicated by dotted lines in Figure 1), which is *linear* in its size rather than exponential. The associated overhead is that the designer has to provide specifications for the variation points. Here, we adapt our previously developed compositional verification technique for temporal safety properties [44] and its automated tool, ProMoVer [43], for this scenario.

Model Mining. The above considerations lead to the natural problem of constructing a hierarchical variability model from an already realized software prod-

uct line. The problem where a model is inferred from a set of programs is sometimes referred to as *model mining*.

In general, the HVMs giving rise to a particular software product line are not unique. We would like to measure how amenable a hierarchical variability model is to analysis by means of divide-and-conquer reasoning as suggested above. To this end we define a quality measure, called the *separation degree* of a model, as the ratio between the total number of artifacts from which products are constructed and the total number of artifact occurrences in the leaves of the model. High-quality models capture repetitions of products in a family without repetition in the model. The maximum theoretically possible separation degree of one is only reached in models where artifacts occur exactly once.

The problem then becomes to construct, from a given software product line, an HVM with maximum separation degree. We introduce a natural class of software product lines termed *simple* for which the optimal HVMs are unique and have separation degree one. We present a model mining transformation that constructs the unique optimal HVM from a given simple family.

Contributions. This paper combines and extends two of our earlier results: The hierarchical variability model for software product lines [20] and a technique for verification of families modeled in this way [39]. The combination essentially provides an efficient verification technique for simple families that have either been originally described in a modeling language that does not capture solution space variability, or families that have been produced in an ad hoc manner, for instance as a result of evolving and adapting a piece of software for different customers. For such families, the technique of the first paper allows the algorithmic extraction of a variability model, which is then used by the technique of the second paper to drive the verification of all products of the family. Thus, the main technical contributions of this paper are:

- A formal definition of *simple hierarchical variability models (SHVM)*, together with a quality measure called *separation degree* and a set of *well-formedness constraints* yielding (by construction) models with maximal measure (Subsection 2.1).
- A formal semantics for hierarchical variability models in terms of *family generation*, and a proof that, for every well-formed variability model, the generated family is simple (Subsection 2.2).
- A *characterization result* stating that, for well-formed hierarchical variability models and simple families, family generation and hierarchical variability model construction are *inverses* of each other, thus implying correctness of model construction and uniqueness of well-formed models with respect to the families they generate (Subsection 2.2).
- A procedure to construct hierarchical variability models from simple families that produces well-formed models (Subsections 2.2 and 2.3).
- An adaptation of a previously developed compositional verification framework and its tool support, ProMoVer, for verifying control flow temporal

safety properties of all products of simple families represented through (constructed) SHVMs (Section 3).
– Evaluation of the tools on a small but realistic case study (Section 4).

The proofs of all results presented in the paper can be found in the Appendix.

2 Hierarchical Variability Models

In this section, we present our variability models and their semantics, and relate them with families of products. We also illustrate our construction of variability models from families, by an example.

2.1 Families and Variability Models

Here, we first present product families as a semantic domain for our hierarchical variability models and then define formally these models.

We develop our formalization using a straightforward notation. However, the formalization can also be carried out in the terminology of relational algebra, or the one of regular languages. We choose a neutral notation here since our intended application domains are of a general nature.

Families. We consider products realized by a set of artifact implementations for a given set of artifact names. An artifact can be thought of as, e.g., a component or a method. We fix a countably infinite set of artifact names Art.

Definition 1 (Product, family). *An* artifact implementation *is an indexed artifact name; let a_i denote the i-th implementation of artifact name a. A* product *P is a finite set of artifact implementations, where for each artifact name there is at most one implementation. A* family *$\mathcal{F}$ is a finite non-empty set of products.*

Thus, products can be seen as partial maps from artifact names to natural numbers, having a finite domain; we use Nat^{Art} to denote the set of all products over Art. We refer to singleton set families as *core* families, or simply cores. The family consisting of the empty product is denoted $1_{\mathcal{F}}$.

Example 1. Here are some families that are used later to illustrate various notions.

$$\begin{aligned}\mathcal{F}_A = \{\ & \{a_1, b_1, c_1, d_1, e_1\}, \{a_1, b_1, c_1, d_1, e_2\}, \{a_1, b_1, c_2, d_2, e_1\}, \\ & \{a_1, b_1, c_2, d_2, e_2\}, \{a_1, b_1, c_2, d_3, e_1\}, \{a_1, b_1, c_2, d_3, e_2\}\ \} \\ \mathcal{F}_B = \{\ & \{a_1, b_1\}, \{a_1, b_2\}, \{a_2, b_1\}\ \}\end{aligned}$$

Next, we define two mappings for identifying the artifact names and artifact implementations that occur in a family.

Definition 2 (Family names and implementations). *The mapping names* $(\mathcal{F})$ *from families to sets of artifact names and the mapping impls*$(\mathcal{F})$ *from families to sets of artifact implementations are defined as follows, where* $a^1, \ldots, a^n \in Art$ *and* $i_1, \ldots, i_n \in Nat$*:*

$$\begin{aligned} names\,(\mathcal{F}) &\stackrel{\mathsf{def}}{=} \textstyle\bigcup_{P\in\mathcal{F}} names\,(P) \\ where\ names\left(\{a^1_{i_1}, \ldots, a^n_{i_n}\}\right) &\stackrel{\mathsf{def}}{=} \{a^1, \ldots, a^n\} \\ impls(\mathcal{F}) &\stackrel{\mathsf{def}}{=} \textstyle\bigcup_{P\in\mathcal{F}} impls(P) \\ where\ impls(\{a^1_{i_1}, \ldots, a^n_{i_n}\}) &\stackrel{\mathsf{def}}{=} \{a^1_{i_1}, \ldots, a^n_{i_n}\} \end{aligned}$$

In this definition we abuse notation by also defining mappings with the same names from products to the same co-domains.

We use two binary operations on families, the usual set union operation $\cup$ and the *product union* operation $\bowtie$ over families with disjoint sets of artifact names defined by:

$$\mathcal{F}_1 \bowtie \mathcal{F}_2 \stackrel{\mathsf{def}}{=} \{P_1 \cup P_2 \,|\, P_1 \in \mathcal{F}_1 \wedge P_2 \in \mathcal{F}_2\}$$

and generalized through $\prod_{i\in I} \mathcal{F}_i$ to non-empty sets of families[4]. Intuitively, the product union of two families is the family having as products all possible combinations of products of the original families. Both operations are commutative and associative.

We now define a distinct class of families that we later relate to a specific class of hierarchical variability models. The class of families contains all single-product families consisting of a single artifact implementation, and is closed under product union of families over disjoint sets of artifact names, and under union of families over the same set of artifact names, but having disjoint implementations.

Definition 3 (Simple family). *The class* $\boldsymbol{F}$ *of* simple *families is the least set of families closed under the formation rules:*

(F1) $\{\{a_i\}\} \in \boldsymbol{F}$ *for any* $a \in Art$ *and* $i \in Nat$*.*
(F2) $\mathcal{F}_1 \bowtie \mathcal{F}_2 \in \boldsymbol{F}$ *for any* $\mathcal{F}_1, \mathcal{F}_2 \in \boldsymbol{F}$ *such that* $names\,(\mathcal{F}_1) \cap names\,(\mathcal{F}_2) = \emptyset$*.*
(F3) $\mathcal{F}_1 \cup \mathcal{F}_2 \in \boldsymbol{F}$ *for any* $\mathcal{F}_1, \mathcal{F}_2 \in \boldsymbol{F}$ *such that* $names\,(\mathcal{F}_1) = names\,(\mathcal{F}_2)$ *and* $impls(\mathcal{F}_1) \cap impls(\mathcal{F}_2) = \emptyset$*.*

Example 2. The family $\{\{a_1, b_1\}, \{a_1, b_2\}\}$ is simple, as it can be presented as $\{\{a_1\}\} \bowtie (\{\{b_1\}\} \cup \{\{b_2\}\})$ which follows the above formation rules. Family $\mathcal{F}_A$ of Example 1 is also simple (as we shall see later in Example 6), while family $\mathcal{F}_B$ of Example 1 is not: there is no way of building this family with the above formation rules.

[4] In relational algebra these are the usual union $\cup$ and Cartesian product $\times$ on relations with disjoint sets of attributes, a partial case of the more general join operation $\bowtie$.

Simplicity of families expresses that different functionalities in a product line are always orthogonal, and that alternative realizations of the same functionality have always disjoint implementations. These assumptions are rather heavy and may not always hold in practice. But only under such severe constraints can one hope for such a (strong) uniqueness result as the one obtained later (Section 2.1).

To characterize the applicability of the formation rules, we introduce the concept of correlation between artifact names as a restriction on the possible combinations of their implementations. If two artifact names are correlated, then not all possible combinations of artifact implementations occur in the family which means that the artifact implementations depend on each other.

Two distinct artifact names $a, b \in \mathit{names}(\mathcal{F})$ are termed *correlated* in a family $\mathcal{F}$, denoted $a\, C_{\mathcal{F}}\, b$, if there are implementations $a_i, b_j \in \mathit{impls}(\mathcal{F})$ such that no product in $\mathcal{F}$ contains both implementations simultaneously. Otherwise, names a and b are termed *uncorrelated* or *orthogonal*. The correlation relation $C_{\mathcal{F}}$ on $\mathit{names}(\mathcal{F})$ is symmetric, and hence, its reflexive and transitive closure $C^*_{\mathcal{F}}$ is an equivalence relation. As usual, we denote the partitioning induced by $C^*_{\mathcal{F}}$ on $\mathit{names}(\mathcal{F})$ by $\mathit{names}(\mathcal{F})/C^*_{\mathcal{F}}$ (quotient set).

Example 3. Consider family $\mathcal{F}_A$ of Example 1. The only two correlated names are c and d, evidenced by the lack of a product containing, for instance, c_1 and d_2. Thus, we have $\mathit{names}(\mathcal{F}_A)/C^*_{\mathcal{F}_A} = \{\{a\}, \{b\}, \{c, d\}, \{e\}\}$.

Correlation (and orthogonality) extends naturally to products in a family: Products P and P' are correlated in $\mathcal{F}$ if some artifact name occurring in P is correlated to some artifact name occurring in P'.

Similarly, we define the sharing relation $N_{\mathcal{F}}$ on $\mathcal{F}$ as $P_1\, N_{\mathcal{F}}\, P_2 \overset{\mathsf{def}}{\Leftrightarrow} P_1 \cap P_2 \neq \emptyset$, and use its reflexive and transitive closure $N^*_{\mathcal{F}}$ to partition the family $\mathcal{F}$.

The following result provides sufficient conditions for the applicability of the three formation rules for simple families from Definition 3. The proof of this proposition, as all other proofs can be found in the appendix. As usual, $\overline{A}$ denotes the complement of set A in a given universe of elements.

Proposition 1. *Let family $\mathcal{F}$ be simple. The following holds.*

(i) Let $a_i \in \mathit{impls}(\mathcal{F})$, and let $\mathcal{F}'$ be the projection of $\mathcal{F}$ on $\mathit{names}(\mathcal{F})\backslash\{a\}$. The name a_i occurs in all products of $\mathcal{F}$, i.e., *$a_i \in \bigcap_{P\in\mathcal{F}} P$, iff $\mathcal{F} = \big\{\{a_i\}\big\} \bowtie \mathcal{F}'$. Then either $\mathcal{F}' = 1_{\mathcal{F}}$ and thus rule (F1) applies, or else $\mathcal{F}'$ is simple and rule (F2) applies.*

(ii) Let $\{A_1, A_2\}$ be a non-trivial partitioning of $\mathit{names}(\mathcal{F})$, and let $\mathcal{F}_1$ and $\mathcal{F}_2$ be the projections of $\mathcal{F}$ on A_1 and A_2, respectively. Every name in A_1 is orthogonal to every name in A_2 in $\mathcal{F}$, i.e., *$A_1 \times A_2 \subseteq \overline{C_{\mathcal{F}}}$, iff $\mathcal{F} = \mathcal{F}_1 \bowtie \mathcal{F}_2$ and $\mathcal{F}_1$ and $\mathcal{F}_2$ are simple. Formation rule (F2) applies in this case.*

(iii) Let $\{\mathcal{F}_1, \mathcal{F}_2\}$ be a non-trivial partitioning of $\mathcal{F}$. No product of $\mathcal{F}_1$ shares an artifact implementation with any product of $\mathcal{F}_2$, i.e., *$\mathcal{F}_1 \times \mathcal{F}_2 \subseteq \overline{N_{\mathcal{F}}}$, iff $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$ and $\mathcal{F}_1$ and $\mathcal{F}_2$ are simple. Formation rule (F3) applies in this case.*

The following important property of simple families follows from the above result: if a simple family $\mathcal{F}$ can be formed by formation rule ($\mathcal{F}2$) with some suitable $\mathcal{F}_1$ and $\mathcal{F}_2$ satisfying the rule's condition, then it cannot be formed by formation rule ($\mathcal{F}3$), and *vice versa*.

When restricted to simple families, the two operations on families do not distribute over each other. This entails that simple families have *unique* formation trees modulo commutativity and associativity of the two operations associated with the rules.

Variability Models. In order to represent solution space variability of families in terms of shared artifact implementations, we consider simple hierarchical variability models.

Definition 4 (Simple hierarchical variability model). *A* simple hierarchical variability model (SHVM) $\mathcal{S}$ *is inductively defined as:*

(i) a (possibly empty) common set *of artifact implementations* M_C*, or*

(ii) a pair $(M_C, \{VP_1, \ldots, VP_n\})$ *where* M_C *is defined as above and the set* $\{VP_1, \ldots, VP_n\}$ *of* variation points *is non-empty. A variation point* $VP_i = \{\mathcal{S}_{i,j} \mid 1 \leq j \leq k_i\}$*, where* $k_i \geq 2$*, is a set of (at least two) SHVMs called* variants*.*

We sometimes refer to an SHVM simply as a variability model. An SHVM consisting of only a common set of artifact implementations is called *ground model*. An SHVM generates a family $\mathcal{F}$ through all possible ways of resolving the variabilities of the SHVM. This process recursively selects exactly one variant for each variation point. We defer a formal definition of such a semantics for SHVMs to Section 2.2. Variability models can be naturally depicted as trees, where the leaves are common sets of artifact implementations, and the internal nodes are the roots of SHVMs or variation points.

Example 4. Figure 2 and Figure 3 show four variability models named $\mathcal{S}_{A1}$, $\mathcal{S}_{A2}$, $\mathcal{S}_{B1}$, and $\mathcal{S}_{B2}$. In these figures, (sub)trees showing variability models are rooted with boxes, and subtrees showing variation points are rooted with diamonds.

Analogously to Definition 2, we define two mappings for identifying the artifact names and artifact implementations that occur in SHVMs.

Definition 5 (SHVM names and implementations). *The mapping names* $(\mathcal{S})$ *from SHVMs to sets of artifact names and the mapping impls*$(\mathcal{S})$ *from SHVMs to sets of artifact implementations are defined as follows, where* $a^1, \ldots, a^n \in Art$

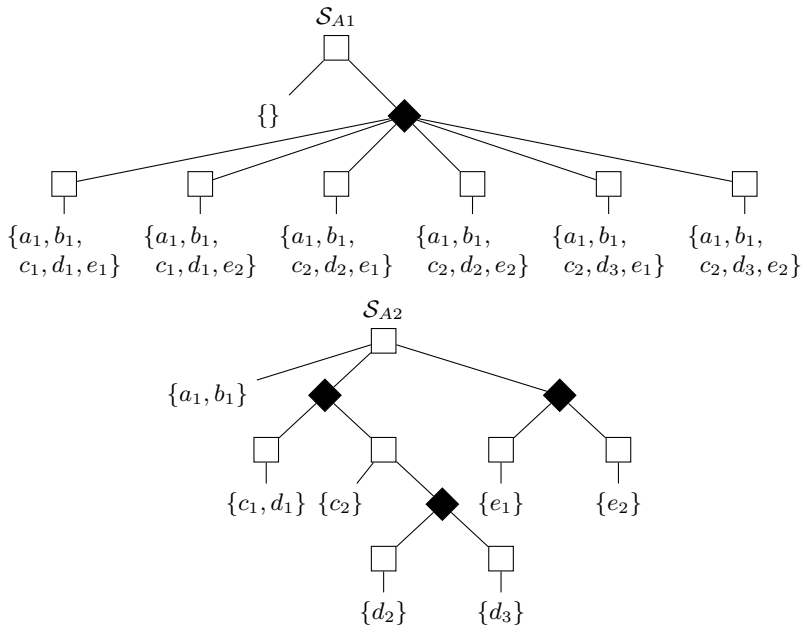


Fig. 2. SHVMs $\mathcal{S}_{A1}$ and $\mathcal{S}_{A2}$ for the family $\mathcal{F}_A$ in Example 1.

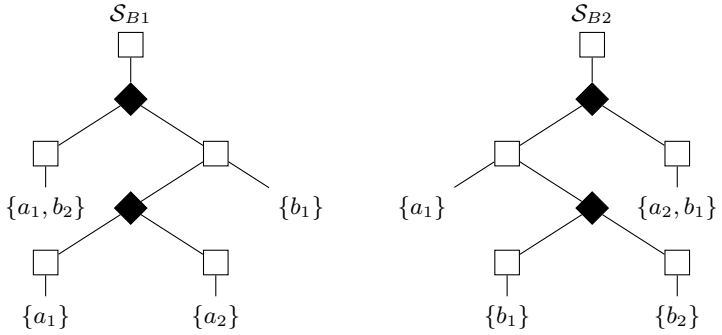


Fig. 3. SHVMs $\mathcal{S}_{B1}$ and $\mathcal{S}_{B2}$ for the family $\mathcal{F}_B$ in Example 1.

and $i_1, \ldots, i_n \in Nat$*:*

$$names\left(\{a^1_{i_1}, \ldots, a^n_{i_n}\}\right) \stackrel{\text{def}}{=} \{a^1, \ldots, a^n\}$$
$$names\left((M_C, \{VP_1, \ldots, VP_n\})\right) \stackrel{\text{def}}{=} names\left(M_C\right) \cup \bigcup_{1 \leq i \leq n} names\left(VP_i\right)$$
$$where\ names\left(VP\right) \stackrel{\text{def}}{=} \bigcup_{\mathcal{S} \in VP} names\left(\mathcal{S}\right)$$
$$impls(\{a^1_{i_1}, \ldots, a^n_{i_n}\}) \stackrel{\text{def}}{=} \{a^1_{i_1}, \ldots, a^n_{i_n}\}$$
$$impls((M_C, \{VP_1, \ldots, VP_n\})) \stackrel{\text{def}}{=} impls(M_C) \cup \bigcup_{1 \leq i \leq n} impls(VP_i)$$
$$where\ impls(VP) \stackrel{\text{def}}{=} \bigcup_{\mathcal{S} \in VP} impls(\mathcal{S})$$

Again we abuse notation by also defining mappings with the same names from variation points to the same co-domains.

Next we define a measure of the degree of separation in a variability model as the ratio between the cardinality of the set of artifact implementations and the sum of the cardinalities of the leaves of the SHVM tree. The separation degree is, thus, a number in the interval $(0, 1]$ that captures the degree to which the commonalities and orthogonalities of products are factored out as common sets and variation points in a variability model, respectively: the higher this degree, the less artifact implementations occur repeatedly in more than one leaf. The maximum value of 1 holds when every artifact implementation occurs in exactly one leaf; this is trivially the case for ground models.

Definition 6 (Separation degree). *The* separation degree $sd(\mathcal{S})$ *of a variability model* $\mathcal{S}$ *is defined as:*

$$sd(\{\}) \stackrel{\text{def}}{=} 1$$
$$sd(\mathcal{S}) \stackrel{\text{def}}{=} \frac{|impls(\mathcal{S})|}{sd'(\mathcal{S})} \qquad if\ \mathcal{S} \neq \{\}$$

where $|S|$ *denotes the cardinality of set* S*, and* $sd'(\mathcal{S})$ *is inductively defined as follows:*

$$sd'(M_C) \stackrel{\text{def}}{=} |M_C|$$
$$sd'((M_C, \{VP_1, \ldots, VP_n\})) \stackrel{\text{def}}{=} sd'(M_C) + \Sigma_{1 \leq i \leq n} sd'(VP_i)$$
$$where\ sd'(VP) \stackrel{\text{def}}{=} \Sigma_{\mathcal{S} \in VP} sd'(\mathcal{S})$$

Intuitively this definition captures the extent to which orthogonal artifact implementations are delegated to separate variation points, and the extent to which disjointness of artifact implementations is delegated to separate variants. Since this is the original intention of variation points and variants in our model, separation degree is an obvious quality measure indicating how well the model is used for the purpose of hierarchically representing a software family.

The following definition provides a set of well-formedness constraints on SHVMs. Variability models satisfying these constraints always have separation degree one, as we show in Proposition 2.

Definition 7 (Well-formed variability model). *A ground variability model* $\mathcal{S} = M_C$ *is* well-formed *if constraint (*$\mathcal{S}1$*) below is satisfied. A variability model* $\mathcal{S} = (M_C, \{VP_1, \ldots, VP_n\})$ *with variation points* $VP_i = \{\mathcal{S}_{i,j} \mid 1 \leq j \leq k_i\}$ *is* well-formed *if all variants* $\mathcal{S}_{i,j}$ *are well-formed, and furthermore, the following constraints are satisfied:*

($\mathcal{S}1$*)* M_C *implements artifact names at most once.*
($\mathcal{S}2$*)* $names(M_C) \cap names(VP_i) = \emptyset$ *for all* i*, and*
$names(VP_{i_1}) \cap names(VP_{i_2}) = \emptyset$ *whenever* $i_1 \neq i_2$*.*
($\mathcal{S}3$*)* $names(\mathcal{S}_{i,j_1}) = names(\mathcal{S}_{i,j_2})$ *for all* i, j_1, j_2*, and*
$impls(\mathcal{S}_{i,j_1}) \cap impls(\mathcal{S}_{i,j_2}) = \emptyset$ *whenever* $j_1 \neq j_2$*.*

Example 5. Consider the SHVMs $\mathcal{S}_{A1}$ and $\mathcal{S}_{A2}$ depicted in Figure 2. $\mathcal{S}_{A1}$ is not well-formed whereas $\mathcal{S}_{A2}$ is. The separation degrees are $sd(\mathcal{S}_{A1}) = \frac{9}{6 \cdot 5} = 0.3$ and $sd(\mathcal{S}_{A2}) = \frac{9}{9} = 1$. Figure 3 depicts two other SHVMs, $\mathcal{S}_{B1}$ and $\mathcal{S}_{B2}$. Neither of these are well-formed and both have separation degree $\frac{4}{5} = 0.8$.

The constraints in Definition 6 ensure that the separation degree of a well-formed SHVM is equal to 1 and is thus maximum.

Proposition 2. *If variability model* $\mathcal{S}$ *is well-formed then* $sd(\mathcal{S}) = 1$*.*

Note that the converse of Proposition 2 does not hold in general: The variability model $\{a_1, a_2\}$ has separation degree 1, but well-formedness constraint ($\mathcal{S}1$) is not satisfied.

Proposition 3. *For a given SHVM, let AND and OR denote the maximum branching factors at SHVM and variation point nodes, respectively, and let ND be its nesting depth. The number of products generated by the SHVM is bound by* $OR^{\frac{AND \cdot (AND^{ND} - 1)}{AND - 1}}$ *and is thus exponential in the size of the SHVM, which is bound by* $\frac{(OR \cdot AND)^{(ND+1)} - 1}{OR \cdot AND - 1}$*.*

Inversely stated, SHVMs can be exponentially more succinct than the underlying family.

2.2 Relating Families and Variability Models

In this subsection we present translations between well-formed variability models and simple families and show that they are inverses of each other. In particular, this entails that the translation from simple families to variability models produces the unique well-formed model generating the respective family, thus giving a procedure for constructing a variability model from a given family.

From Variability Models to Families. The set of products generated by a ground model is the singleton set comprising the set of common artifact implementations and, thus, representing one product. The set of products generated by a variation point is the union of the product sets generated by its variants. Finally, the set of products generated by an SHVM with a non-empty set of variation points is the set of all products consisting of the common artifact implementations and of exactly one product from the set generated by each variation point.

Definition 8 (Family generation). *The mapping family($\mathcal{S}$) from variability models to families is inductively defined as follows:*

$$\mathit{family}(M_C) \stackrel{\mathsf{def}}{=} \{M_C\}$$
$$\mathit{family}((M_C, \{VP_1, \ldots, VP_n\})) \stackrel{\mathsf{def}}{=} \{M_C\} \bowtie \textstyle\prod_{1 \leq i \leq n} \mathit{family}(VP_i)$$
$$\text{where } \mathit{family}(VP) \stackrel{\mathsf{def}}{=} \textstyle\bigcup_{\mathcal{S} \in VP} \mathit{family}(\mathcal{S})$$

We say that variability model $\mathcal{S}$ generates *family($\mathcal{S}$).*

Here we again abuse notation by also defining a mapping with the same name from variation points to the same co-domain. Family generation is well-defined in the sense that well-formed variability models generate simple families.

Proposition 4. *If variability model $\mathcal{S}$ is well-formed, then family($\mathcal{S}$) is simple.*

Example 6. SHVMs $\mathcal{S}_{A1}$ and $\mathcal{S}_{A2}$ in Figure 2 both generate family $\mathcal{F}_A$ in Example 1, implying that family $\mathcal{F}_A$ is simple since $\mathcal{S}_{A2}$ is well-formed. SHVMs $\mathcal{S}_{B1}$ and $\mathcal{S}_{B2}$ in Figure 2 both generate family $\mathcal{F}_B$ in Example 1. Among these four SHVMs, $\mathcal{S}_{A2}$, $\mathcal{S}_{B1}$ and $\mathcal{S}_{B2}$ have maximum separation degree in the sense that, for each of the families $\mathcal{F}_A$ and $\mathcal{F}_B$, no other SHVMs for the same family have higher separation degree.

From Families to Variability Models. We now present a reverse transformation from simple families to well-formed variability models. Recall that simple families have unique formation trees modulo commutativity and associativity of the two operations. Well-formed SHVMs can thus be seen as a uniform way of grouping the formation terms. Every family $\mathcal{F}$ can be decomposed into the form:

$$\mathcal{F} = \{P\} \bowtie \mathcal{F}_V, \quad \mathcal{F}_V = \textstyle\prod_{1 \leq i \leq n} \mathcal{F}_i, \quad \mathcal{F}_i = \bigcup_{1 \leq j \leq k_i} \mathcal{F}_{i,j}$$

where P is a product, or equivalently, as a single equation:

$$\mathcal{F} = \{P\} \bowtie \textstyle\prod_{1 \leq i \leq n} \bigcup_{1 \leq j \leq k_i} \mathcal{F}_{i,j} \qquad (*)$$

The existence of the decomposition is ensured since every family $\mathcal{F}$ can be trivially decomposed as $\{\emptyset\} \bowtie \prod \bigcup \mathcal{F}$, *i.e.*, with product P being empty and $n = k_1 = 1$. Decomposition $(*)$ is only unique under additional constraints, under which the decomposition is called canonical.

Definition 9 (Canonical form of family). *A family $\mathcal{F}$, decomposed as equation $(*)$ above, is in* canonical form *if the following conditions hold:*

(C1) The product P is the set of artifact implementations that are common to all products in $\mathcal{F}$.
(C2) The set of artifact names in $\mathcal{F}_V$ has n equivalence classes w.r.t. correlated artifact names $C^*_{\mathcal{F}_V}$, and for the i-th equivalence class, the family $\mathcal{F}_i$ is the projection of $\mathcal{F}_V$ onto the artifact names of the class.
(C3) For all i, $1 \leq i \leq n$, $\mathcal{F}_{i,j}$ are the k_i equivalence classes of $\mathcal{F}_i$ w.r.t. implementation sharing $N^*_{\mathcal{F}_i}$.

A consequence of the following proposition is that definitions and proofs may exploit the canonical form to proceed by induction on the size of simple families.

Proposition 5. *If $\mathcal{F}$ is a simple non-core family in canonical form then for all i, $1 \leq i \leq n$, and $k_i \geq 2$ all $\mathcal{F}_{i,j}$ are simple and of strictly smaller size than $\mathcal{F}$.*

The decomposition into canonical form is clearly unique for a simple family, and exposes one level of hierarchy. Thus, by iterative application of the decomposition, we obtain a mapping from families to hierarchical variability models.

Definition 10 (Variability model generation). *The mapping $shvm(\mathcal{F})$ from simple families presented in canonical form to variability models is inductively defined as follows:*

$$shvm(\{P\}) \stackrel{\mathsf{def}}{=} P$$

$$shvm\big(\{P\} \bowtie \textstyle\prod_{1\leq i\leq n} \bigcup_{1\leq j\leq k_i} \mathcal{F}_{i,j}\big) \stackrel{\mathsf{def}}{=} (P, \{VP_1, \ldots, VP_n\})$$
$$\text{where } VP_i \stackrel{\mathsf{def}}{=} \{shvm(\mathcal{F}_{i,j}) \mid 1 \leq j \leq k_i\}$$

We say that family $\mathcal{F}$ generates *variability model $shvm(\mathcal{F})$.*

Proposition 5 guarantees that the above mapping is well-defined, in the sense that $shvm(\mathcal{F})$ is indeed an SHVM. Furthermore, as the next result shows, the generated variability model is well-formed.

Proposition 6. *If family $\mathcal{F}$ is simple, then $shvm(\mathcal{F})$ is well-formed.*

Example 7. Consider the family $\mathcal{F}_A$ from Example 1.

- In the first step of the decomposition of $\mathcal{F}_A$ into canonical form we obtain the common set $P = \{a_1, b_1\}$ and the family $\mathcal{F}_V = \{\{c_1, d_1, e_1\}, \{c_1, d_1, e_2\}, \{c_2, d_2, e_1\}, \{c_2, d_2, e_2\}, \{c_2, d_3, e_1\}, \{c_2, d_3, e_2\}\}$.
- In the next step, we analyze $\mathcal{F}_V$ to find that only artifact names c and d are correlated. Projecting $\mathcal{F}_V$ onto the two resulting equivalence classes $\{c, d\}$ and $\{e\}$ we obtain the two variation points $\mathcal{F}_1 = \{\{c_1, d_1\}, \{c_2, d_2\}, \{c_2, d_3\}\}$ and $\mathcal{F}_2 = \{\{e_1\}, \{e_2\}\}$.

– In the third step, we analyze $\mathcal{F}_1$ and see that two products share the artifact implementation c_2, which gives us the variants $\mathcal{F}_{1,1} = \{\{c_1, d_1\}\}$ and $\mathcal{F}_{1,2} = \{\{c_2, d_2\}, \{c_2, d_3\}\}$, and then analyze $\mathcal{F}_2$ to obtain the variants $\mathcal{F}_{2,1} = \{\{e_1\}\}$ and $\mathcal{F}_{2,2} = \{\{e_2\}\}$.

Only $\mathcal{F}_{1,2}$ is not a ground model. Applying the above steps decomposes it into a common set $\{c_2\}$ and a single variation point with two variants consisting of the common sets $\{d_2\}$ and $\{d_3\}$. It is easy to see that $\mathit{shvm}(\mathcal{F}_A)$ is the variability model $\mathcal{S}_{A2}$ in Figure 2.

Characterization Results. Our first result establishes *correctness* of model extraction.

Lemma 1. *For every simple family $\mathcal{F}$ we have:*

$$\mathit{family}(\mathit{shvm}(\mathcal{F})) = \mathcal{F}$$

The second result establishes *uniqueness* of well-formed models w.r.t. the generated (simple) family.

Lemma 2. *For every well-formed variability model $\mathcal{S}$ we have:*

$$\mathit{shvm}(\mathit{family}(\mathcal{S})) = \mathcal{S}$$

An immediate consequence of the above two lemmas is our main characterization result, which essentially states that the two transformations relating variability models and families are inverses of each other.

Theorem 1 (Characterization Theorem). *For every simple family $\mathcal{F}$ and every well-formed variability model $\mathcal{S}$ we have:*

$$\mathit{family}(\mathcal{S}) = \mathcal{F} \iff \mathit{shvm}(\mathcal{F}) = \mathcal{S}$$

2.3 Model Extraction from Code

Here we explain, using an example, how to extract variability models from program code of simple product families. The example is written in Java, but our method is independent of the programming language.

Example 8. As a running example in the rest of this paper, we consider a product line of cash desks that is a simplified version of a case study from the HATS project [37]. A cash desk processes purchases by retrieving the prices for all items to be purchased and calculates the total price. After the customer has paid, a receipt is printed and the stock is updated. All cash desks have in common that every purchase is processed following the same process. However, the cash desks differ in how items are entered. Some cash desks allow entering products using a keyboard, others only provide a scanner, and a third group provides both options. Payment at some cash desks can only be made in cash. Other cash

```
public class CashDesk {

    public void sale() {
        int prodNu = 10;
        for (int i = 0; i < 10; i++) {
            int prod = enterProd();
            writeReceipt(prod);
            prodNu = updateStock(prodNu);
            payment();
        }
    }

    public int enterProd() {
        return useKeyboard();
    }

    public void payment() {
        cardPay(enterCard());
    }

    public static void main(String[] args) {
        (new CashDesk()).sale();
    }

    /* The implementation of the private
       methods, including methods
       writeReceipt, updateStock, cardPay,
       enterCard, and useKeyboard are not
       shown here.
     */
}
```

```
public class CashDesk {

    public void sale() {
        int prodNu = 10;
        for (int i = 0; i < 10; i++) {
            int prod = enterProd();
            writeReceipt(prod);
            prodNu = updateStock(prodNu);
            payment();
        }
    }

    public int enterProd() {
        return useScanner();
    }

    public void payment() {
        cashPay();
    }

    public static void main(String[] args) {
        (new CashDesk()).sale();
    }

    /* The implementation of the private
       methods, including methods
       writeReceipt, updateStock, cardPay,
       enterCard, and useScanner are not
       shown here.
     */
}
```

Fig. 4. Products P_2 (top) and P_4 (bottom) from the Cash Desk product line.

desks only accept credit cards, while a third group allows the choice between cash and credit card payment.

Figure 4 shows two of nine products from the product line where each product takes the form of a Java class called CashDesk. At the top is product code for a cash desk for entering with keyboard and paying with credit card only. At the bottom of the figure is product code for a cash desk that scans products and accepts cash payment only. These nine Java classes can be converted into a family of products in the sense of Definition 1 by considering public method names as artifact names and the corresponding method bodies as artifact implementations. This yields the following simple family:

$$\mathcal{F}_{\mathsf{CashDesk}} = \{P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9\}$$

where:

$$\begin{aligned}
P_1 &= \{sale_{\mathsf{CashDesk}}, enterProd_{\mathsf{Keyboard}}, payment_{\mathsf{Cash}}\}\\
P_2 &= \{sale_{\mathsf{CashDesk}}, enterProd_{\mathsf{Keyboard}}, payment_{\mathsf{Card}}\}\\
P_3 &= \{sale_{\mathsf{CashDesk}}, enterProd_{\mathsf{Keyboard}}, payment_{\mathsf{CashOrCard}}\}\\
P_4 &= \{sale_{\mathsf{CashDesk}}, enterProd_{\mathsf{Scanner}}, payment_{\mathsf{Cash}}\}\\
P_5 &= \{sale_{\mathsf{CashDesk}}, enterProd_{\mathsf{Scanner}}, payment_{\mathsf{Card}}\}
\end{aligned}$$

$$\begin{aligned}
P_6 &= \{sale_{\mathsf{CashDesk}}, enterProd_{\mathsf{Scanner}}, payment_{\mathsf{CashOrCard}}\}\\
P_7 &= \{sale_{\mathsf{CashDesk}}, enterProd_{\mathsf{KeyboardOrScanner}}, payment_{\mathsf{Cash}}\}\\
P_8 &= \{sale_{\mathsf{CashDesk}}, enterProd_{\mathsf{KeyboardOrScanner}}, payment_{\mathsf{Card}}\}\\
P_9 &= \{sale_{\mathsf{CashDesk}}, enterProd_{\mathsf{KeyboardOrScanner}}, payment_{\mathsf{CashOrCard}}\}
\end{aligned}$$

The common purchase process of all cash desks is modeled by the artifact name *sale* and implementation (subscript) CashDesk. The artifact names *enterProd* and *payment* are common to all products, but their implementations vary: Cash, Card, or CashOrCard. Starting form family $\mathcal{F}_{\mathsf{CashDesk}}$, and following steps similar to those of Example 7, gives the following SHVM.

$$shvm(\mathsf{CashDesk}) = \left(\{sale_{\mathsf{CashDesk}}\}, \{@\mathsf{EnterProducts}, @\mathsf{Payment}\}\right)$$

$$\begin{aligned}
\textit{where}\ @\mathsf{EnterProducts} &= \{\mathsf{Keyboard}, \mathsf{Scanner}, \mathsf{KeyboardOrScanner}\}\\
@\mathsf{Payment} &= \{\mathsf{Cash}, \mathsf{Card}, \mathsf{CashOrCard}\}
\end{aligned}$$

$$\begin{aligned}
\textit{and}\quad \mathsf{Keyboard} &= \{enterProd_{\mathsf{Keyboard}}\}\\
\mathsf{Scanner} &= \{enterProd_{\mathsf{Scanner}}\}\\
\mathsf{KeyboardOrScanner} &= \{enterProd_{\mathsf{KeyboardOrScanner}}\}\\
\mathsf{Cash} &= \{payment_{\mathsf{Cash}}\}\\
\mathsf{Card} &= \{payment_{\mathsf{Card}}\}\\
\mathsf{CashOrCard} &= \{payment_{\mathsf{CashOrCard}}\}
\end{aligned}$$

The two variation points @EnterProducts and @Payment represent the variabilities of the cash desks. Variation point @EnterProducts has associated variants Keyboard, Scanner and KeyboardOrScanner, while variation point @Payment has associated variants Cash, Card and CashOrCard. Figure 5 shows the model as a diagram.

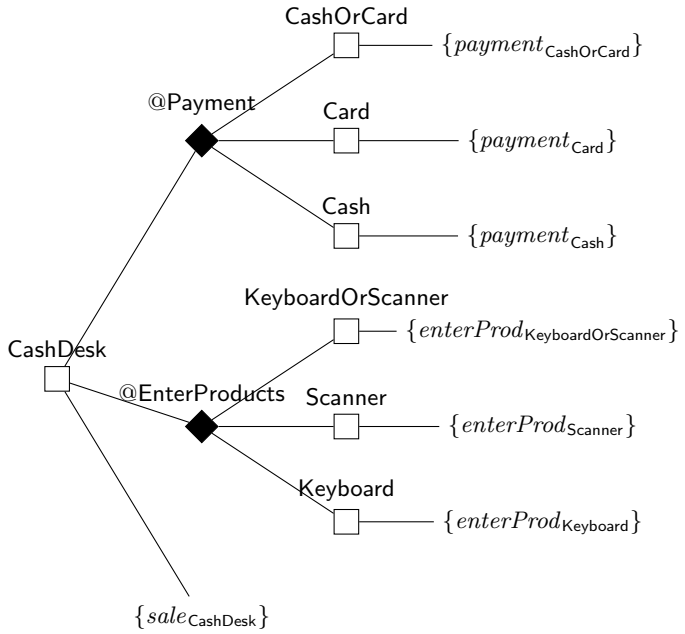


Fig. 5. The CashDesk hierarchical variability model (drawn sideways).

As we describe in Section 4.1, the extraction of SHVM models from an existing simple family of products (explained by the above example) is implemented as a part of our tool support. These models can be used for hierarchical analyses of product families. In the next section, we show how they facilitate efficient verification of temporal safety properties.

3 Verification of Temporal Safety Properties of Software Product Lines

Suppose we have a large software family that has either been produced in an ad hoc manner (for instance as a result of evolving and adapting a software product for different customers) or that has been developed by some methodology that does not capture solution space variability. Suppose also that we want

to apply some given standard static program analysis technique, such as formal verification, on the implementations (i.e., the code) of *all* products of the family. Naturally in such a case we should strive to minimize the overall effort by maximizing the *reuse* of partial verification results obtained for the shared artifacts. In the previous section we developed a technique to extract automatically SHVMs from the implementations of simple families. Since the extracted SHVMs capture the sharing of artifacts in the solution space, they contain, in a succinct representation (see Proposition 3), precisely the information that is needed to maximize the reuse of analysis results.

The exact way of utilizing SHVMs for software product line verification depends heavily on the concrete verification technique at hand. Especially suited for the task are *compositional* techniques, since they reduce the verification of whole products to the individual verification of their components (i.e., artifacts), and thus allow the reuse of the latter in case they are shared. In this paper, we illustrate this idea by adapting a previously developed compositional verification technique for temporal safety properties of Java programs, and its tool support [19,18], to the setting of software product lines. Let us first explain intuitively our original compositional verification framework, and then describe its adaptation for product families.

Our original framework for compositional verification is a realization of *assume-guarantee* reasoning for the verification of *incomplete* programs, i.e. programs where the implementation of some of their components are not available. Hence, such programs consist of so-called *concrete components* available through their implementations and of unavailable *abstract components*. To verify incomplete programs, we require a user provided *local* specification for each abstract component that describes its legal behavior (assumption). Our verification framework relativizes the correctness of *global properties* of such programs on the local specifications of their abstract components and the implementation of the concrete ones, thus dividing the verification task into the following two independent subtasks:

(a) a check that the composition of the local specifications of abstract components together with the implementation of concrete ones entails the global property, and
(b) a check that the implementation of each abstract component (once it becomes available) satisfies its local specification.

Technically, for subtask (b) a control flow graph is extracted from the code of each abstract component (once it becomes available), and is model checked against its local specification. A control flow graph, here called *flow graph*, is a collection of *method graphs*, each representing the control flow structure of the code of a method (see Definition 12 and Example 9). For subtask (a), however, so-called *maximal flow graphs* are constructed from the local specifications of abstract components. Intuitively, a maximal flow graph for a local specification ϕ is the most general flow graph satisfying ϕ. Thus it can be used, for the purposes of verification, as a representation of *any* implementation of the component that satisfies ϕ. These maximal models are composed with the flow graphs extracted

from the code of concrete ones, and then the behavior of the result represented as a pushdown automaton is model checked against the global property of the program.

To adapt our framework to the verification of temporal safety properties of SHVMs, we require user provided local properties at all variation points. These properties should abstractly express the legal behavior of all their underlying variants (see Example 11 for concrete properties). The idea is that for the verification of variants their underlying variation points and core methods (i.e., their children variation point and core nodes in the graph) can be viewed as abstract and concrete components, respectively. Then the verification of the variants is *relativized* on the properties of their underlying variation points, while the correctness of the variation points is *established* through verifying their underlying variants (i.e., their children variant nodes in the graph). This results in a hierarchical verification scheme that is realized by the following two steps:

1. Verify each variation point by checking, using step (2), that all its underlying variants satisfy its specification. This essentially means that underlying variants attached to a variation point inherit the property of their parent variation point.
2. Verify each variant by checking that the composition of maximal flow graphs constructed from the local specifications of its underlying variation points, together with the flow graphs extracted from its core methods, satisfy the property of the variant. By this, we basically verify that *all* sub-products constructed by composing the different artifact implementations below a variant satisfy its property.

Since the family corresponds to the root variant, the global property of the software family is the property of the top-level variant of its SHVM.

As we show in Section 3.2, this verification procedure is *sound*: If it succeeds for SHVM $\mathcal{S}$ and global property ϕ, then all products of $\mathcal{S}$ satisfy ϕ.

For example, to verify the CashDesk product line modeled by the SHVM in Figure 5, the variation points `@EnterProducts` and `@Payment` are locally specified, and the desired global property of all products would be the property of variant `CashDesk`. Then the verification procedure follows the steps below:

1. Verify that each individual variation point satisfies its property independently. This is achieved for instance for variation point `@EnterProducts` by independently checking that the variants `Keyboard`, `Scanner`, and `KeyboardOrScanner` satisfy the local specification of `@EnterProducts`.
2. Construct maximal flow graph for the variation points `@EnterProducts` and `@Payment`, compose these with the flow graphs extracted from the core method `sale`, and model check the result against the property of `CashDesk`.

In the remainder of this section, we first present our compositional verification framework formally, and then describe how it is adapted to the verification of software families represented by SHVMs.

3.1 A Framework for Compositional Verification

Here, we define our program models and specification language and present our compositional verification principle.

Program Model. In order to reason algorithmically about sequences of method invocations, we abstract the set of methods defining our program by ignoring all data. An initialized model serves as an abstract representation of a program's structure and behavior.

Definition 11 (Model). *A* model *is a (Kripke) structure* $\mathcal{M} = (S, L, \rightarrow, A, \lambda)$ *where* S *is a set of states,* L *a set of labels,* $\rightarrow \subseteq S \times L \times S$ *a labeled transition relation,* A *a set of atomic propositions, and* $\lambda : S \rightarrow \mathcal{P}(A)$ *a valuation, assigning to each state* s *the set of atomic propositions that hold in* s*. An* initialized model *is a pair* $(\mathcal{M}, E)$ *with* $\mathcal{M}$ *a model and* $E \subseteq S$ *a set of initial states.*

A *method graph* is an instance of an initialized model which is obtained by ignoring all data from a method implementation. A *flow graph* is a collection of *method graphs*, one for each method of the program. It is a standard model for the analysis of control flow based properties [6].

Definition 12 (Method graph). *Let Meth be a countably infinite set of methods names. A* method graph *for method* $m \in Meth$ *over a set of method names* $M \subseteq Meth$ *is an initialized model* $(\mathcal{M}_m, E_m)$ *where* $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$ *is a finite model and* $E_m \subseteq V_m$ *is a non-empty set of* entry points *of* m*.* V_m *is the set of* control nodes *of* m*,* $L_m = M \cup \{\varepsilon\}$*,* $A_m = \{m, r\}$*, and* $\lambda_m : V_m \rightarrow \mathcal{P}(A_m)$ *so that* $m \in \lambda_m(v)$ *for all* $v \in V_m$ *(i.e., each node is tagged with its method name). The nodes* $v \in V_m$ *with* $r \in \lambda_m(v)$ *are* return points*.*

Note that according to the above definition, methods can have multiple entry points. Flow graphs that are extracted from a program source have single entry points, but the maximal models that we generate for compositional verification can have multiple entry points.

Every flow graph $\mathcal{G}$ is equipped with an *interface* $I = (I^+, I^-)$, denoted $\mathcal{G} : I$, where $I^+, I^- \subseteq Meth$ are the *provided* and externally *required* methods, respectively. Interfaces are needed when constructing maximal flow graphs. A flow graph is *closed* if its interface does not require any methods (i.e., $I^- = \emptyset$) and it is *open* otherwise. Flow graph *composition* is defined as the disjoint union $\uplus$ of their method graphs.

Example 9. Figure 6 shows a simple Java class and the (simplified) flow graph it induces. It consists of two method graphs, for method `even` and method `odd`, respectively. Entry nodes are depicted as usual by incoming edges without source. Its interface is $(\{\texttt{even}, \texttt{odd}\}, \emptyset)$, thus the flow graph is closed.

The operational semantics of flow graphs, here called flow graph *behavior*, is also defined as an instance of an initialized model, induced through the flow graph structure. We use transition label τ for internal transfer of control,

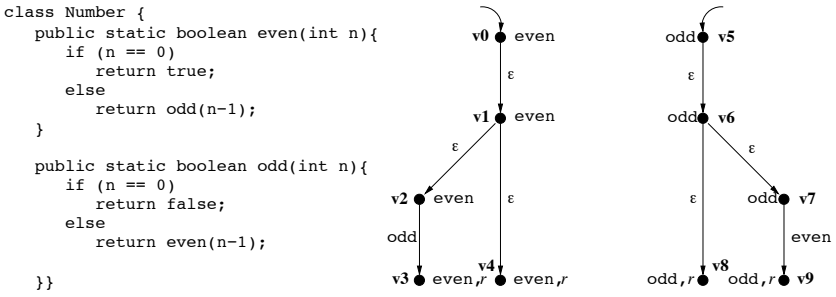


Fig. 6. A simple Java class and its flow graph.

m_1 `call` m_2 for the invocation of method m_2 by method m_1 when method m_2 is provided by the program and m_1 call! m_2 when method m_2 is external (e.g., API methods), and m_2 `ret` m_1 respectively m_2 ret? m_1 for the corresponding return from the call.

Definition 13 (Flow Graph Behavior). *Let $\mathcal{G} = (\mathcal{M}, E) : (I^+, I^-)$ be a flow graph such that $\mathcal{M} = (V, L, \rightarrow, A, \lambda)$. The behavior of $\mathcal{G}$ is defined as an initialized model $b(\mathcal{G}) = (\mathcal{M}_b, E_b)$, where $\mathcal{M}_b = (S_b, L_b, \rightarrow_b, A_b, \lambda_b)$, such that $S_b = (V \cup I^-) \times V^*$, i.e., states are pairs of control points v or required method names m, and stacks σ, $L_b = \{m_1\ k\ m_2 \mid k \in \{\mathsf{call}, \mathsf{ret}\}, m_1, m_2 \in I^+\} \cup \{m_1\ \mathsf{call!}\ m_2 \mid m_1 \in I^+, m_2 \notin I^+\} \cup \{m_2\ \mathsf{ret?}\ m_1 \mid m_1 \in I^+, m_2 \notin I^+\} \cup \{\tau\}$, $A_b = A$, $\lambda_b((v, \sigma)) = \lambda(v)$ and $\lambda_b((m, \sigma)) = m$, and $\rightarrow_b \subseteq S_b \times L_b \times S_b$ is defined by the following rules:*

[transfer] $(v, \sigma) \xrightarrow{\tau} (v', \sigma)$ if $m \in I^+,\ v \xrightarrow{\varepsilon}_m v',\ v \models \neg r$

[call] $(v_1, \sigma) \xrightarrow{m_1\ \mathtt{call}\ m_2} (v_2, v_1' \cdot \sigma)$ if $m_1, m_2 \in I^+, v_1 \xrightarrow{m_2}_{m_1} v_1', v_1 \models \neg r,$ $v_2 \models m_2, v_2 \in E$

[ret] $(v_2, v_1 \cdot \sigma) \xrightarrow{m_2\ \mathtt{ret}\ m_1} (v_1, \sigma)$ if $m_1, m_2 \in I^+, v_2 \models m_2 \wedge r, v_1 \models m_1$

[call!] $(v_1, \sigma) \xrightarrow{m_1\ \mathsf{call!}\ m_2} (m_2, v_1' \cdot \sigma)$ if $m_1 \in I^+, m_2 \in I^-, v_1 \xrightarrow{m_2}_{m_1} v_1', v_1 \models \neg r$

[ret?] $(m_2, v_1 \cdot \sigma) \xrightarrow{m_2\ \mathsf{ret?}\ m_1} (v_1, \sigma)$ if $m_1 \in I^+, m_2 \in I^-, v_1 \models m_1$

The set of initial states is defined by $E_b = E \times \{\varepsilon\}$, where ε denotes the empty sequence over $V \cup I^-$.

Notice that return transitions always hand back control to the caller of the method. Calls to external methods are modeled with an intermediate state, from which only an immediate return is possible. In this way possible callbacks from external methods are not captured in the behavior. This simplification is justified, since we abstract away from data in the model and the behavior is thus context-free, but has to be kept in mind when writing specifications; in particular one cannot specify that callbacks are not allowed.

Example 10. Consider the flow graph of Example 9. One example run through its (branching, infinite-state) behavior, from an initial to a final configuration, is:

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

Now, consider just the method graph of method `even` as an open flow graph, having interface $(\{\texttt{even}\},\{\texttt{odd}\})$. The *local contribution* of method `even` to the above global behavior is the following run:

$$(v_0,\varepsilon)\xrightarrow{\tau}(v_1,\varepsilon)\xrightarrow{\tau}(v_2,\varepsilon)\xrightarrow{\texttt{even call! odd}}(\texttt{odd},v_3)\xrightarrow{\texttt{odd ret? even}}(v_3,\varepsilon)$$

An alternative way to express flow graph behavior is by means of *pushdown systems* (PDS). We exploit this by using pushdown system model checking to verify behavioral properties [41].

Specification Language. To specify global and local properties we hare use the safety fragment of *linear temporal logic* (LTL) that uses the weak version of until[5].

Definition 14 (Safety LTL). *The formulas of* sLTL *are inductively defined by:*

$$\phi ::= p \mid \neg p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \texttt{X}\,\phi \mid \texttt{G}\,\phi \mid \phi_1\,\texttt{W}\,\phi_2$$

where $p \in A_b$ denotes the set of atomic propositions.

Satisfaction on states $(\mathcal{M}_b, s) \models \phi$ is defined in the standard fashion [45] as validity of ϕ over all runs starting from state $s \in S_b$ in model $\mathcal{M}_b$. For instance, formula $\texttt{X}\,\phi$ holds of state s in model $\mathcal{M}_b$ if ϕ holds in the second state of every run starting from s, while $\phi\ \texttt{W}\ \psi$ holds in s if for every run starting in s, either ϕ holds in all states of the run, or ψ holds in some state of the run and ϕ holds in all previous states. Satisfaction of a formula ϕ in flow graph $\mathcal{G}$ with behavior $b(\mathcal{G}) = (\mathcal{M}_b, E_b)$ is defined as satisfaction of ϕ on all initial states $s \in E_b$.

Satisfaction is generalized to product lines in the obvious way: A product line described by a variability model $\mathcal{S}$ satisfies a formula ϕ if the behavior $b(\mathcal{G}_p)$ of the flow graph $\mathcal{G}_p$ of every product $p \in \mathit{products}(\mathcal{S})$ satisfies ϕ.

Compositional Verification. As we mentioned, our method for compositional verification is based on the construction of *maximal flow graphs* for properties of sets of methods. For a given property ψ and interface I consisting of provided and required methods, consider the class of all flow graphs with interface I satisfying ψ. A maximal flow graph for ψ and I is a flow graph $\mathcal{M}ax(\psi, I)$ that

[5] The theoretical underpinnings of our compositional verification framework are actually based on a slightly more expressive specification language, namely *simulation logic*, the fragment of the modal μ-calculus [25] with boxes and greatest fixed-points only. For details see again our previous work [19].

satisfies exactly those properties that hold for all members of the class. Thus, the maximal flow graph can be used as a representative of the class for the purpose of checking properties. Using maximal models for compositional verification was first proposed by Grumberg and Long [17] for finite-state systems, and we generalized it for flow graphs [19,18].

Suppose a system with n components that are partitioned into two sets: The set of abstract components $\mathcal{G}_1, ..., \mathcal{G}_k$ specified with their local properties and interfaces $(\psi_1, I_1), ..., (\psi_k, I_k)$, and the set of concrete components $\mathcal{G}_{k+1}, ..., \mathcal{G}_n$. The main principle of compositional verification based on maximal flow graphs, can relativize the global correctness of such systems on the local specifications $(\psi_1, I_1), ..., (\psi_k, I_k)$, by the proof rule presented below.

$$\frac{\mathcal{G}_1 \models \psi_1 \ \cdots \ \mathcal{G}_k \models \psi_k \qquad \biguplus_{j=k+1,...,n} \mathcal{G}_j \ \uplus \ \biguplus_{i=1,...,k} \mathcal{M}ax(\psi_i, I_i) \models \phi}{\biguplus_{i=1,...,n} \mathcal{G}_i \models \phi} \tag{1}$$

The principle states that the composition of n components (here a set of methods), in which k of them are specified by their local specifications, satisfies global property ϕ if (i) each specified (abstract) component $\mathcal{G}_i$ satisfies its respective local property ψ_i and (ii) the composition of the k maximal flow graphs $\mathcal{M}ax(\psi_i, I_i)$ with the flow graphs extracted from the code of the other components (concrete components) $\mathcal{G}_{k+1}, ..., \mathcal{G}_n$ satisfies ϕ.

As we proved previously [19], the rule is sound and complete when interfaces describe all provided and required methods[6].

3.2 SHVM-driven Algorithmic Verification

For efficient verification of product families represented by SHVMs, we introduce the notion of *regions* in SHVMs, each of which is formed by an SHVM node (variant) and its underlying variation points and artifacts implementations, e.g., regions of the SHVM in Figure 1 are indicated by dotted lines. In this section, we propose a compositional reasoning approach that is linear in the number of regions in the SHVM description of the product line rather than linear in the number of generated products (which is exponential in the number of regions). This approach is an instantiation of the compositional verification principle presented above to SHVMs.

To show that all products generated from an SHVM satisfy global property Φ, the top-level region of the SHVM is specified with Φ, and also every variation point VP of the SHVM is specified by a behavioral property ψ_{VP} and its interface $I_{VP} = (I^+_{VP}, I^-_{VP})$ declaring the names of the provided and required

[6] Our proof [19] is for global properties ϕ written in behavioral simulation logic and local properties ψ_i in structural simulation logic; here in the context of sLTL we use translations into the respective logic.

methods. The underlying variants attached to a variation point inherit the corresponding variation point specification. Then, our verification procedure for SHVMs is as follows.

Verification Procedure. For every region $M = (M_C, \{VP_1, \ldots, VP_n\})$ of the SHVM with the property ϕ, perform the following two tasks:

(i) For every artifact name $a \in Art(M_C)$, extract the flow graph $\mathcal{G}_a$ from $Imp(a)$.

(ii) For all variation points VP_i with specification (ψ_{VP_i}, I_{VP_i}), construct the maximal flow graph $\mathcal{M}ax(\psi_{VP_i}, I_{VP_i})$. Then, compose the constructed graphs with the flow graphs of task (i), and model check the resulting flow graph against the region property ϕ, i.e.,

$$\biguplus_{a \in Art(M_C)} \mathcal{G}_a \quad \uplus \quad \biguplus_{1 \leq i \leq n} \mathcal{M}ax(\psi_{VP_i}, I_{VP_i}) \models \phi \tag{2}$$

For properties given in sLTL, the behavior of $\mathcal{G}_{\mathcal{M}ax}$ is represented as a PDS and standard PDS model checking is used.

The presented verification procedure is *sound*, as established by the following theorem.

Theorem 2. *Let $\mathcal{S}$ be an SHVM with global property ϕ. If the verification procedure succeeds for $\mathcal{S}$, then $p \models \phi$ for all its products $p \in products(\mathcal{S})$.*

The total number of verification tasks needed to establish the global product line property is, thus, equal to the number of regions, since we have to complete one verification task per region. In contrast, the number of products is exponential in the number of regions.

Example 11. To illustrate our compositional verification approach, we use the cash desk product line described in Example 8. The global behavioral property we want to verify is informally stated as follows:

> The entering of products has to be finished before the payment process has started.

Taking into account the distribution of functionality to artifact intended by the variability model from the example, the specification can be approximated as:

> If control starts in method `sale`, it cannot reach method `payment` before it has already been in method `enterProd` and then back in `sale`.

In terms of the (global) behavior of the flow graphs of the products induced by the product line, this property can be formalized in sLTL as follows:

$$\varphi_{CD} \;=\; sale \rightarrow (\neg payment \;\mathtt{W}\; (enterProd \;\wedge\; r \;\wedge\; \mathtt{X}\; sale))$$

where the subformula $enterProd \wedge r \wedge \mathtt{X}\; sale$ captures a return from `enterProd` to `sale`.

First, we have to specify all variation points of the cash desk SHVM. The specification of the `@EnterProd` and `@Payment` variation points are as follows:

– The interface of variation point `@EnterProducts` is $I_{EP} = (\{\texttt{enterProd}\}, \{\texttt{payment}\})$. The property required for the variation point is that the `enterProd` method never makes calls to `payment` method. Formally, this property can be expressed by the formula[7]:

$$\varphi_{EP} \;=\; \texttt{G}\; \neg \mathit{payment}$$

– The interface of variation point `@Payment` is $I_P = (\{\texttt{payment}\}, \{\texttt{enterProd}\})$. Similarly to the variation point above, the property required for this variation point is that the `payment` method never makes calls to the `enterProd` method:

$$\varphi_{P} \;=\; \texttt{G}\; \neg \mathit{enterProd}$$

The variants `Keyboard`, `Scanner`, and `KeyboardOrScanner` inherit their specifications from the `@EnterProducts` variation point, and the variants `Cash`, `Card` and `CashOrCard` from the `@Payment` variation point.

Finally, we have to establish that all regions satisfy their respective property. For the top-level region, we construct the maximal flow graphs for the specifications of the variation points `@EnterProducts` and `@Payment` and compose these with the flow graph of method `sale`, and model check φ_{CD} against the composition result. Then variants `Keyboard`, `Scanner`, `KeyboardOrScanner`, `Cash`, `Card` and `CashOrCard` are verified also by model checking the flow graph extracted from their implementation against their inherited verification point property.

4 Tool Support and Evaluation

Our tool support for the verification of product families consists of two tools: A tool that constructs SHVMs from families, and another one that automatically verifies temporal properties of SHVMs. Using these tools, we verify a simple family in two steps; first we construct the SHVM representation of the family and then we verify temporal safety properties of the constructed SHVM.

4.1 Construction of Simple Hierarchical Variability Models

We have implemented an algorithm that takes as input a simple family and produces its SHVM decomposition. The algorithm is not written explicitly in this paper but can be unambiguously inferred from Definition 9 and Definition 10. Our implementation is written in OCaml. Its input is a text file containing the products of the family. The constructed family is a list of sets, each set representing one product. The sets' elements are records, each record having two fields: name and number. Each record represents an implementation, the name being the name of the artifact and the number, the corresponding index. Having constructed the family $\mathcal{F}$ we proceed as dictated by Definition 10. First

[7] This and the following property would trivialize if we specified the set of required methods to be empty. For now, however, our tool does not check interfaces.

we factor out the common implementations, if any, and then we proceed with the remainder $\mathcal{F}_V$. We identify the equivalence classes of the $C^*_{\mathcal{F}_V}$ relation using Union-Find structures. For each equivalence class $\mathcal{F}_i$ we identify the equivalence classes of the $N^*_{\mathcal{F}_i}$ relation. If there are no common implementations and each of the two equivalence relations has a single equivalence class, by Proposition 5 the family $\mathcal{F}$ is not simple and the program exits with an appropriate message. Otherwise, recursive calls are made on each of the equivalence classes of the $N^*_{\mathcal{F}_i}$ relation. A very crude upper bound on the running time is $O(n^4)$, n being the size of the family.

4.2 Automated Modular Verification of SHVMs

ProMoVer [43] is a fully automated tool for the procedure-modular verification of control flow temporal safety properties of Java programs[8]. It supports compositional verification by relativizing the correctness of a global program property on properties of individual methods and their interfaces. All interfaces, variation points and global properties are provided to the tool as assertions in the form of program annotations. ProMoVer accepts a JML-like syntax for annotations (cf. [27]) as special comments called *pragmas*. For scalability, ProMoVer provides a proof storage and reuse mechanism which stores flow graphs, maximal models and model checking results and reuses these the next time the same program is verified. To reuse the stored information, ProMoVer checks for each method of the program: if the source code of the method has not changed, the stored flow graph of the method is used, if a local specification has not changed the stored maximal model for the specification is used. Further, it provides users with a library of global properties which contains platform as well as application specific properties. For details about ProMoVer, the reader is referred to [44].

We have adapted ProMoVer for verifying properties of SHVMs according to the compositionality principle described in Section 3.2. For this adaptation, we have extended the annotation language to support the definition of variants and variation points and the associated specifications by designated pragmas. The tool takes as input a source code file in which the SHVM to be analyzed is represented by annotations. The product property and the variation point properties are also provided by annotations. Figure 7 shows in the left column the annotation for the `@EnterProd` variation point, while the annotation for its `Keyboard` variant is shown in the right column. ProMoVer fully automatically extracts the SHVM modules and the corresponding flow graphs from the annotated source code and performs the associated model checking tasks.

For evaluating our compositional verification approach, we considered the verification of the safety property explained in Example 11 for different versions of the trading system product line [37]. The product lines of cash desks were described as SHVMs with different hierarchical depths and different total numbers of modules. As a basis, we used the product line described in Example 8

[8] ProMoVer is available via the web interface `www.csc.kth.se/~siavashs/ProMoVer`

```
/**
* @variation_point:
*   EnterProd
* @variation_point_interface:   /**@variant: Keyboard
*   provided enterProd          * @variant_interface:
* @variation_point_ltl_prop:    *   provided enterProd()
*   G ! payment                 * @variation_points:
* @variants:                    */
*     Keyboard,Scanner,
*     KeyboardOrScanner         public int enterProd(){
*/                              ...
```

Fig. 7. Annotations for variation point `@EnterProd` and its variant `Keyboard`

Product Line	Depth	# Modules	# Products	t_{ind}[s]	t_{comp}[s]
CD	1	7	9	79	9
CD/CH	1	9	18	177	10
CD/CT	2	15	27	278	11
CD/CH/CT	2	17	54	652	12

Table 1. Evaluation Results

and extended it by an optional coupon handling functionality within the `sale` method, and a variation point for accepting different card types as a hierarchical refinement of variant `Card`. For each product line, we compared the time required to verify all induced products individually with the time for compositional verification. The experiments were performed on a SUN SPARC machine[9].

The results are summarized in Table 1 where `CD` denotes the product line of Example 8, `CD/CH` the version with coupon handling, `CD/CT` the version with different card types and `CD/CH/CT` the version with coupon handling and different card types. As can be observed from the table, the processing time t_{ind} for verifying every product individually grows dramatically when new modules and levels of hierarchy are added to the SHVM. This is easily explained by the analytical bounds presented in Section 3.2. In contrast, the growth of the processing time t_{comp} for compositional SHVM verification is insignificant, since the preprocessing and flow graph extraction is only performed once by ProMoVer for the complete SHVM. The experiment suggests that for large software products comprising many products, the compositional verification technique based on the SHVM representation of the product line increases efficiency of verification dramatically.

Scalability of our method comes at the price of having to provide specifications for variation points. This additional effort is justified for large systems that render infeasible the verification of the product line by verifying all its products

[9] The focus of the evaluation is on comparing the times required for verification, and not on the total times themselves.

individually. Also, the specifications only need to be written once and are later reused when the code has been changed, or for proving other global properties.

SHVMs do not allow to express that a variant requires or excludes another variant. Without these constraints, the set of products that can be derived from an SHVM is larger than with requires/excludes constraints. If a desired property can be shown for the larger set of products defined by an SHVM, the property immediately holds for the original product set defined by the hierarchical variability model. However, this leaves the possibility that not all products defined by an SHVM satisfy a property such that verification procedure fails, while the property is satisfied by the products defined by an hierarchical variability model containing variant constraints. In this case, an additional check of the excluded products would be required.

5 Related Work

Variability Modeling. Hierarchical variability models represent solution space variability. The existing approaches to represent solution space product line variability can be divided into three directions [40]. First, annotative approaches consider one model representing all products of a product line. Variant annotations, *e.g.*, using UML stereotypes [52,15], presence conditions [10], or separate variability representations, such as orthogonal variability models [36], define which parts of the model have to be removed to generate the model of a concrete product. Second, compositional approaches [4,50,34,3] associate product fragments with product features which are composed for particular feature configurations, such as hierarchical variability models. Third, transformational approaches [22,8] represent variability by rules determining how a base model has to be changed for a particular product model. All these approaches consider a representation of artifact variability without any hierarchy.

Our hierarchical variability model generalizes the ideas of the Koala component model [49] for the implementation of variant-rich component-based systems. In Koala, the variability of a component is described by the variability of its subcomponents which can be selected by *switches* and explicit *diversity interfaces*. Diversity switches and interfaces in Koala can be understood as concrete language constructs at the implementation level targeted to express variation points and associated variants. Plastic partial components [35] are an architectural modeling approach where component variability is defined by extending partially defined components with variation points and associated variants. However, variants cannot contain variable components so this modeling approach is not truly hierarchical. Hierarchical variability modeling for software architectures [21] applies the modeling concepts for solution space variability presented in this paper to component-based software engineering and provides a concrete modeling language for variable software architectures that is truly hierarchical.

However, none of these approaches formally defines the semantics of hierarchical variability models, nor reasons about their well-formedness or uniqueness. Simple hierarchical variability models strike a balance between the expressive-

ness of the modeling formalism—no bindings and being grammar-like—and the desirable property of uniqueness of models: With a more expressive modeling formalism, uniqueness may not be achievable. To the best of our knowledge, this work is the first to provide a formal semantics for hierarchical variability models in the solution space, and to characterize a class of variability models through the class of generated product families.

Variability Model Mining. This paper presents the first approach for constructing a hierarchical variability model for solution space variability from a given product family. So far, there have only been approaches to construct feature models for representing problem space variability for a given set of products. Czarnecki *et al.* [12] re-construct a feature model from a set of sample feature combinations using data mining techniques [1]. Other approaches aim at constructing feature models from sample mappings between products and their features using formal concept analysis [14], for instance, to derive logical dependencies between code variants from pre-processor annotations [42], or to construct a feature model for function-block based systems after determining model variants by similarity [38]. Loesch and Ploedereder [29] use formal concept analysis to optimize feature models in case of product line evolution, *e.g.*, to remove unused features or to combine features that always occur together. Niu and Easterbrook [33] apply formal concept analysis to functional and non-functional product line requirements in order to construct a feature model as a more abstract representation of the requirements. Also, information retrieval techniques are applied to obtain a feature model from heterogeneous product line requirements [2]. Using hierarchical clustering, a tree structure of textually similar requirements is constructed. Requirement clusters in the leaves are more similar to each other than requirements clusters closer to the root giving rise to the structure of a feature model.

In our work, we abstract from the need to determine the different variants of the same conceptual entity by assuming fixed artifact names and corresponding artifact implementations. However, if we relax this assumption, techniques, such as similarity analysis [38] or formal concept analysis [14] could be applied to infer the relationship between different variants of the same conceptual entity, and thus make our approach applicable.

Regular expressions and relational algebras. Regular expressions (regexps) were introduced by Kleene [24]. Several variants of the original definition are known [46]. A certain analogy between simple families and regexps without Kleene star can be noticed, where individual implementations, the $\cup$ operation, and the $\bowtie$ operation on families correspond to alphabet symbols, the $+$ operation, and concatenation $\cdot$, respectively. There are two major differences, however: in our domain there is a two-level hierarchy names-implementations with no analogue in Formal Languages, and, since products are sets, there is no repetition of implementations in them, while strings can have arbitrary repetitions of symbols. Our goal to construct an optimal SHVM for a given family corresponds to constructing a smallest regexp for a given (finite) language. It is known that

regexp minimization is intractable: even without Kleene star or complement it is still co-NP-complete [46, problem INEQ($\{0,1\},\{\cup,\cdot\}$)] while in general it is PSPACE-complete [31,23]. That discouraging result, however, is with respect to languages that have no restriction of non-repeating symbols. It is worth investigating whether the problem still remains intractable after the said restriction.

Our problem domain bears substantial similarity to relational algebra as well. Our concepts of name, product, family, and product union translate to active domain, tuple, database relation, union, and join of relations, a minor difference being that database theory allows join of relations that share attributes. For a detailed introduction to relational databases and relational algebra, see [30]. Using database terminology, our goal is, given a database relation to deduce aspects of its design. That is, to perform some sort of model mining. Database decomposition has been intensely studied for the purposes of forward design. To the best of our knowledge there are no results on mining the relational database model from a given database.

Verification of Product Families. Most approaches to algorithmic verification of behavioral properties of software product lines rely on an annotative model of the product line comprising all possible product variants in the same model [51,48]. Existing model checking techniques are adapted to deal with optional behavior defined by variant annotations. For instance, in [13], modal transition systems are extended by variability operators from deontic logic. In [16], the process calculus CCS is extended with a variant operator to represent a family of processes. In [26], transitions of I/O-automata are related to variants. In [9], product families are modeled by transition systems where transitions are labeled with features, so that state reachability modulo a set of features can be computed. Also, in [5], safety specifications of features are identified and combined for the analysis of the products.

These approaches do not scale for large product lines since the used annotative product line models easily get very large. To counter this, Blundell et al. [7], Liu et al. [28], and Beek et al. [47] propose techniques for compositional verification of product features. In these approaches, the behavior of a feature is represented by a state machine to which other features may attach in designated states (interface states or variation points). For a temporal property of a feature, constraints for these states are generated which have to be satisfied by composed features. In another work, Millo et al. [32] check the conformance of variability information at the requirement and design level in a feature-based compositional fashion, but they do not address the reuse of verification results. In all these works, the compositionality results are based on the applied notion of features and feature composition, while SHVMs provide a more flexible means to define product variability.

The presented approach is one of the first compositional verification techniques for software product lines. It allows to guarantee efficiently that all products of a product line satisfy certain desired control-flow based safety properties. With respect to model checking behavioral properties of product lines, only Blundell et al. [7] and Liu et al. [28] propose compositional verification techniques

based on assume-guarantee style reasoning for product features. Other model checking approaches for product lines [13,16,26,9] use a monolithic model of the complete product line such that they face severe state-space explosion problems since all possible products are analyzed in the same analysis step.

6 Conclusion

In this article, we present hierarchical solution space variability models for software product lines and we generalize a previously developed compositional technique and tool set for the automatic verification of control-flow based temporal safety properties to software families that can be described by such models.

We give a formal semantics of hierarchical variability models in terms of sets—or families—of products, where each product is a set of artifact implementations. We introduce the separation degree as a quality measure of hierarchical variability models. We identify well-formed variability models as a class of models for which the measure is maximal (and equal to one) and which are unique for the family they generate; the class of families generated by such models is the class of simple families. Furthermore, we present an algorithm that accepts as input a simple family and outputs the unique well-formed model that generates it. We prove uniqueness by showing that family generation and model construction are inverses of each other for this class of models. While maximum separation degree and uniqueness of models with maximal measure are theoretically appealing, in practice, product families might not be simple. Still, the separation degree is a useful measure for hierarchical variability models, and, as Examples 5 and 6 suggest, searching for the set of models with a maximal measure (not necessarily equal to one) for a given family is equally meaningful.

Using the introduced variability model, we adapt a previously developed method and tool set for compositional verification of procedural programs, which allows to avoid the combinatorial explosion of verifying all products individually. The number of verification tasks resulting from our method is linear in the size of the variability model rather than in the number of products. This is achieved by introducing variation point specifications on which product properties are relativized, and by constructing maximal flow graphs that replace the specifications when model checking specifications on the next higher level of hierarchy. The class of properties that can be handled fully automatically is the class of control flow-based temporal safety properties, specifying illegal sequences of method calls. The input to our verification tool is the description of a product line in form of an annotated Java program defining the variability model and the necessary specifications. Our first experiments with the tool show a dramatic gain in performance even for models with a low hierarchical depth.

Future work. Future work will focus on the practical evaluation of the proposed method for variability model mining, considering in particular sets of (legacy code) products that have not been designed as a family from the outset. Further

effort is planned on generalizing the model with optional and multiple variant selections and with requires/excludes constraints between variants, and on adapting accordingly the model reconstruction transformation. Another generalization will deal with the more abstract domain of products over implementations only, where the names are not given in advance, but must be inferred. Additionally, the restriction that all variants associated to a variation point have to provide the same artifact names will be lifted.

References

1. Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD Conference*, pages 207–216, 1993.
2. Vander Alves, Christa Schwanninger, Luciano Barbosa, Awais Rashid, Peter Sawyer, Paul Rayson, Christoph Pohl, and Andreas Rummler. An exploratory study of information retrieval techniques in domain analysis. In *Software Product Line Conference (SPLC)*, pages 67–76, 2008.
3. Sven Apel, Florian Janda, Salvador Trujillo, and Christian Kästner. Model Superimposition in Software Product Lines. In *International Conference on Model Transformation (ICMT)*, volume 5563 of *LNCS*, pages 4–19. Springer, 2009.
4. Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transaction Software Engineering*, 30(6):355–371, 2004.
5. Sara Bessling and Michaela Huhn. Towards formal safety analysis in feature-oriented product line development. In Jeremy Gibbons and Wendy MacCaull, editors, *Foundations of Health Information Engineering and Systems*, volume 8315 of *Lecture Notes in Computer Science*, pages 217–235. Springer Berlin Heidelberg, 2014.
6. Frédéric Besson, Thomas Jensen, Daniel Le Métayer, and Tommy Thorn. Model checking security properties of control flow graphs. *J. of Computer Security*, 9(3):217–250, 2001.
7. Colin Blundell, Kathi Fisler, Shriram Krishnamurthi, and Pascal Van Hentenryck. Parameterized Interfaces for Open System Verification of Product Lines. In *Automated Software Engineering (ASE)*, pages 258–267. IEEE, 2004.
8. Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. Abstract delta modeling. In *Generative Programming and Component Engineering (GPCE)*. Springer, 2010.
9. Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *International Conference on Software Engineering (ICSE)*, pages 335–344. IEEE, 2010.
10. Krzysztof Czarnecki and Michal Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Generative Programming and Component Engineering (GPCE)*, volume 3676 of *LNCS*, pages 422 – 437. Springer, 2005.
11. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
12. Krzysztof Czarnecki, Steven She, and Andrzej Wasowski. Sample spaces and feature models: There and back again. In *Software Product Line Conference (SPLC)*, pages 22–31, 2008.

13. Alessandro Fantechi and Stefania Gnesi. Formal Modeling for Product Families Engineering. In *Software Product Line Conference (SPLC)*, pages 193–202. IEEE, 2008.
14. Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1996.
15. Hassan Gomaa. *Designing Software Product Lines with UML*. Addison Wesley, 2004.
16. Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. Modeling and model checking software product lines. In *Formal Methods for Open Object-based Distributed Systems (FMOODS)*, volume 5051 of *LNCS*, pages 113–131. Springer, 2008.
17. Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.
18. Dilian Gurov and Marieke Huisman. Reducing behavioural to structural properties of programs with procedures. *Theoretical Computer Science*, 480:69–103, 2013.
19. Dilian Gurov, Marieke Huisman, and Christoph Sprenger. Compositional verification of sequential programs with procedures. *Information and Computation*, 206(7):840–868, 2008.
20. Dilian Gurov, Bjarte M. Østvold, and Ina Schaefer. A hierarchical variability model for software product lines. In *Post-proceedings of: ISoLA 2011*, volume 336, pages 181–199, 2012.
21. Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical variability modeling for software architectures. In *Software Product Line Conference (SPLC)*, pages 150–159. IEEE, 2011.
22. Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. Adding Standardized Variability to Domain Specific Languages. In *Software Product Line Conference (SPLC)*, pages 139–148. IEEE, 2008.
23. Tao Jiang and Bala Ravikumar. Minimal nfa problems are hard. *SIAM J. Comput.*, 22(6):1117–1141, 1993.
24. Stephen Cole Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, 1956.
25. Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
26. Kim Lauenroth, Klaus Pohl, and Simon Toehning. Model checking of domain artifacts in product line engineering. In *Automated Software Engineering (ASE)*, pages 467–481. IEEE, 2009.
27. Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. *JML Reference Manual*, February 2007. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.
28. Jing Liu, Samik Basu, and Robyn R. Lutz. Compositional model checking of software product lines using variation point obligations. *Automated Software Engineering (ASE)*, 18(1):39–76, 2011.
29. Felix Loesch and Erhard Ploedereder. Optimization of variability in software product lines. In *Software Product Line Conference (SPLC)*, pages 151–162, 2007.
30. David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.
31. Albert R. MeyerLeavens and Larry J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Proceedings of the 13th Annual Symposium on Switching and Automata Theory (swat 1972)*, SWAT '72, pages 125–129. IEEE Computer Society, 1972.

32. Jean-Vivien Millo, S. Ramesh, ShankaraNarayanan Krishna, and GaneshKhandu Narwane. Compositional verification of software product lines. In EinarBroch Johnsen and Luigia Petre, editors, *Integrated Formal Methods*, volume 7940 of *Lecture Notes in Computer Science*, pages 109–123. Springer Berlin Heidelberg, 2013.
33. Nan Niu and Steve Easterbrook. Concept analysis for product line requirements. In *ACM International Conference on Aspect-Oriented Software Development (AOSD)*, pages 137–148, 2009.
34. Natsuko Noda and Tomoji Kishi. Aspect-Oriented Modeling for Variability Management. In *Software Product Line Conference (SPLC)*, pages 213–222. IEEE, 2008.
35. Jennifer Pérez, Jessica Díaz, Cristóbal Costa Soria, and Juan Garbajosa. Plastic Partial Components: A solution to support variability in architectural components. In *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 221–230, 2009.
36. Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, 2005.
37. Requirement Elicitation, August 2009. Deliverable 5.1 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.
38. Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. Automatic variation-point identification in function-block-based models. In *Generative Programming and Component Engineering (GPCE)*, pages 23–32, New York, NY, USA, 2010. ACM.
39. Ina Schaefer, Dilian Gurov, and Siavash Soleimanifard. Compositional algorithmic verification of software product lines. In *Postproceedings of Intlernational Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 184–203. Springer, 2011.
40. Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software diversity: state of the art and perspectives. *Software Tools for Technology Transfer (STTT)*, 14(5):477–495, 2012.
41. Stefan Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002.
42. Gregor Snelting. Reengineering of configurations based on mathematical concept analysis. *ACM Transaction on Software Engineering and Methodology*, 5:146–189, April 1996.
43. Siavash Soleimanifard, Dilian Gurov, and Marieke Huisman. ProMoVer: Modular verification of temporal safety properties. In *Software Engineering and Formal Methods (SEFM)*, volume 7041 of *LNCS*, pages 366–381, 2011.
44. Siavash Soleimanifard, Dilian Gurov, and Marieke Huisman. Procedure-modular specification and verification of temporal safety properties. *Software and Systems Modeling*, pages 1–18, 2013.
45. Colin Stirling. *Modal and Temporal Logics of Processes*. Springer, 2001.
46. Larry J. Stockmeyer and Albert R. Meyer. Word problems requiring exponential time: Preliminary report. In *ACM Symposium on the Theory of Computing (STOC)*, pages 1–9, 1973.
47. Maurice H ter Beek and Erik P de Vink. Towards modular verification of software product lines with mcrl2. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'14)*, LNCS. Springer, 2014. To appear.

48. Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):6:1–6:45, June 2014.
49. Rob C. van Ommering. Software reuse in product populations. *IEEE Transaction on Software Engineering*, 31(7):537–550, 2005.
50. Markus Völter and Iris Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *Software Product Line Conference (SPLC)*, pages 233–242. IEEE, 2007.
51. Alexander von Rhein, Sven Apel, Christian Kästner, Thomas Thüm, and Ina Schaefer. The pla model: on the combination of product-line analyses. In *Variability Modelling of Software-intensive Systems (VAMOS)*, pages 14–24, 2013.
52. Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel. Towards a UML Profile for Software Product Lines. In *Software Product Familiy Engineering (PFE)*, volume 3014 of *LNCS*, pages 129–139. Springer, 2003.

A Proofs

Proposition 1. *Let family $\mathcal{F}$ be simple. The following holds.*

(i) Let $a_i \in impls(\mathcal{F})$, and let $\mathcal{F}'$ be the projection of $\mathcal{F}$ on $names\,(\mathcal{F}) \setminus \{a\}$. a_i occurs in all products of $\mathcal{F}$, i.e., *$a_i \in \bigcap_{P\in\mathcal{F}} P$, iff $\mathcal{F} = \big\{\{a_i\}\big\} \bowtie \mathcal{F}'$. Then either $\mathcal{F}' = 1_{\mathcal{F}}$ and thus rule (F1) applies, or else $\mathcal{F}'$ is simple and rule (F2) applies.*

(ii) Let $\{A_1, A_2\}$ be a non-trivial partitioning of $names\,(\mathcal{F})$, and let $\mathcal{F}_1$ and $\mathcal{F}_2$ be the projections of $\mathcal{F}$ on A_1 and A_2, respectively. Every name in A_1 is orthogonal to every name in A_2 in $\mathcal{F}$, i.e., *$A_1 \times A_2 \subseteq \overline{C_{\mathcal{F}}}$, iff $\mathcal{F} = \mathcal{F}_1 \bowtie \mathcal{F}_2$ and $\mathcal{F}_1$ and $\mathcal{F}_2$ are simple. Formation rule (F2) applies in this case.*

(iii) Let $\{\mathcal{F}_1, \mathcal{F}_2\}$ be a non-trivial partitioning of $\mathcal{F}$. No product of $\mathcal{F}_1$ shares an artifact implementation with any product of $\mathcal{F}_2$, i.e., *$\mathcal{F}_1 \times \mathcal{F}_2 \subseteq \overline{N_{\mathcal{F}}}$, iff $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$ and $\mathcal{F}_1$ and $\mathcal{F}_2$ are simple. Formation rule (F3) applies in this case.*

Proof. The if parts of each case are immediate from Def. 3. The only-if parts are established by structural induction on the formation of $\mathcal{F}$.

(i) Let $a_i \in impls(\mathcal{F})$, $\mathcal{F}'$ be the projection of $\mathcal{F}$ on $names\,(\mathcal{F}) \setminus \{a\}$, and let $a_i \in \bigcap_{P\in\mathcal{F}} P$. We consider the three possible ways of forming the simple family $\mathcal{F}$.
  (a) Let $\mathcal{F} = \big\{\{b_j\}\big\}$. Then $a_i = b_j$, and so $\mathcal{F} = \big\{\{a_i\}\big\} \bowtie 1_{\mathcal{F}}$.
  (b) Let $\mathcal{F} = \mathcal{F}_1 \bowtie \mathcal{F}_2$ for simple $\mathcal{F}_1$ and $\mathcal{F}_2$ such that $names\,(\mathcal{F}_1) \cap names\,(\mathcal{F}_2) = \emptyset$. Assume w.l.o.g. that $a \in names\,(\mathcal{F}_1)$. Then $a_i \in \bigcap_{P\in\mathcal{F}_1} P$ and, by the induction hypothesis, $\mathcal{F}_1 = \big\{\{a_i\}\big\} \bowtie \mathcal{F}_1'$ where either $\mathcal{F}_1' = 1_{\mathcal{F}}$ or else $\mathcal{F}_1'$ is simple. In both cases, by the associativity of $\bowtie$, $\mathcal{F} = \big\{\{a_i\}\big\} \bowtie \mathcal{F}'$ for $\mathcal{F}' = \mathcal{F}_1' \bowtie \mathcal{F}_2$, and hence $\mathcal{F}'$ is simple.
  (c) The case $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$ for simple $\mathcal{F}_1$ and $\mathcal{F}_2$ such that $names\,(\mathcal{F}_1) = names\,(\mathcal{F}_2)$ and $impls(\mathcal{F}_1) \cap impls(\mathcal{F}_2) = \emptyset$ is not possible when $a_i \in \bigcap_{P\in\mathcal{F}} P$.

(ii) Let $\{A_1, A_2\}$ be a non-trivial partitioning of $names\,(\mathcal{F})$, let $\mathcal{F}_1$ and $\mathcal{F}_2$ be the projections of $\mathcal{F}$ on A_1 and A_2, respectively, and let $A_1 \times A_2 \subseteq \overline{C_{\mathcal{F}}}$. Again we consider three cases.
  (a) The case $\mathcal{F} = \big\{\{b_j\}\big\}$ is not possible when $\{A_1, A_2\}$ is non-trivial.
  (b) Let $\mathcal{F} = \mathcal{F}_1' \bowtie \mathcal{F}_2'$ for simple $\mathcal{F}_1'$ and $\mathcal{F}_2'$ such that $names\,(\mathcal{F}_1') \cap names\,(\mathcal{F}_2') = \emptyset$. Let $A_1' = names\,(\mathcal{F}_1')$ and $A_2' = names\,(\mathcal{F}_2')$. If $A_1' = A_1$ then $A_2' = A_2$ and the result follows immediately. Otherwise, let $A_{1,1}' \stackrel{\mathsf{def}}{=} A_1' \cap A_1$, $A_{1,2}' \stackrel{\mathsf{def}}{=} A_1' \cap A_2$, $A_{2,1}' \stackrel{\mathsf{def}}{=} A_2' \cap A_1$ and $A_{2,2}' \stackrel{\mathsf{def}}{=} A_2' \cap A_2$. Then $\{A_{1,1}', A_{1,2}'\}$ and $\{A_{2,1}', A_{2,2}'\}$ are non-trivial partitionings of A_1' and A_2', respectively. Furthermore, $A_{1,1}' \times A_{1,2}' \subseteq \overline{C_{\mathcal{F}_1'}}$ and $A_{2,1}' \times A_{2,2}' \subseteq \overline{C_{\mathcal{F}_2'}}$. Then, by the induction hypothesis, $\mathcal{F}_1' = \mathcal{F}_{1,1}' \bowtie \mathcal{F}_{1,2}'$ and $\mathcal{F}_2' = \mathcal{F}_{2,1}' \bowtie \mathcal{F}_{2,2}'$ where, for all $i, j \in \{1, 2\}$, $\mathcal{F}_{i,j}$ is the projection of $\mathcal{F}_i$ on A_j and is simple. Then $\mathcal{F}_1 = \mathcal{F}_{1,1}' \bowtie \mathcal{F}_{2,1}'$ and $\mathcal{F}_2 = \mathcal{F}_{1,2}' \bowtie \mathcal{F}_{2,2}'$ are simple, and, by the associativity of $\bowtie$, $\mathcal{F} = \mathcal{F}_1 \bowtie \mathcal{F}_2$.

(c) The case $\mathcal{F} = \mathcal{F}'_1 \cup \mathcal{F}'_2$ for simple $\mathcal{F}'_1$ and $\mathcal{F}'_2$ such that $\mathit{names}(\mathcal{F}'_1) = \mathit{names}(\mathcal{F}'_2)$ and $\mathit{impls}(\mathcal{F}'_1) \cap \mathit{impls}(\mathcal{F}'_2) = \emptyset$ is not possible when $\{A_1, A_2\}$ is non-trivial and $A_1 \times A_2 \subseteq \overline{C_\mathcal{F}}$.

(iii) Let $\{\mathcal{F}_1, \mathcal{F}_2\}$ be a non-trivial partitioning of $\mathcal{F}$ and let $\mathcal{F}_1 \times \mathcal{F}_2 \subseteq \overline{N_\mathcal{F}}$. Again we consider three cases.

(a) The case $\mathcal{F} = \{\{b_j\}\}$ is not possible when $\{\mathcal{F}_1, \mathcal{F}_2\}$ is non-trivial.

(b) The case $\mathcal{F} = \mathcal{F}'_1 \bowtie \mathcal{F}'_2$ for simple $\mathcal{F}'_1$ and $\mathcal{F}'_2$ such that $\mathit{names}(\mathcal{F}'_1) \cap \mathit{names}(\mathcal{F}'_2) = \emptyset$ is also not possible when $\{\mathcal{F}_1, \mathcal{F}_2\}$ is non-trivial and $\mathcal{F}_1 \times \mathcal{F}_2 \subseteq \overline{N_\mathcal{F}}$.

(c) Let $\mathcal{F} = \mathcal{F}'_1 \cup \mathcal{F}'_2$ for simple $\mathcal{F}'_1$ and $\mathcal{F}'_2$ such that $\mathit{names}(\mathcal{F}'_1) = \mathit{names}(\mathcal{F}'_2)$ and $\mathit{impls}(\mathcal{F}'_1) \cap \mathit{impls}(\mathcal{F}'_2) = \emptyset$. If $\mathcal{F}'_1 = \mathcal{F}_1$ then $\mathcal{F}'_2 = \mathcal{F}_2$ and the result follows immediately. Otherwise, let $\mathcal{F}'_{1,1} \stackrel{\mathsf{def}}{=} \mathcal{F}'_1 \cap \mathcal{F}_1$, $\mathcal{F}'_{1,2} \stackrel{\mathsf{def}}{=} \mathcal{F}'_1 \cap \mathcal{F}_2$, $\mathcal{F}'_{2,1} \stackrel{\mathsf{def}}{=} \mathcal{F}'_2 \cap \mathcal{F}_1$ and $\mathcal{F}'_{2,2} \stackrel{\mathsf{def}}{=} \mathcal{F}'_2 \cap \mathcal{F}_2$. Then $\{\mathcal{F}'_{1,1}, \mathcal{F}'_{1,2}\}$ and $\{\mathcal{F}'_{2,1}, \mathcal{F}'_{2,2}\}$ are non-trivial partitionings of $\mathcal{F}'_1$ and $\mathcal{F}'_2$, respectively. Furthermore, $\mathcal{F}'_{1,1} \times \mathcal{F}'_{1,2} \subseteq \overline{N_{\mathcal{F}'_1}}$ and $\mathcal{F}'_{2,1} \times \mathcal{F}'_{2,2} \subseteq \overline{N_{\mathcal{F}'_2}}$. Then, by the induction hypothesis, $\mathcal{F}'_{1,1}$, $\mathcal{F}'_{1,2}$, $\mathcal{F}'_{2,1}$ and $\mathcal{F}'_{2,2}$ are all simple, and hence $\mathcal{F}_1 = \mathcal{F}'_{1,1} \cup \mathcal{F}'_{2,1}$ and $\mathcal{F}_2 = \mathcal{F}'_{1,2} \cup \mathcal{F}'_{2,2}$ are simple, too. Furthermore, since $\mathcal{F}_1 \times \mathcal{F}_2 \subseteq \overline{N_\mathcal{F}}$ implies $\mathit{impls}(\mathcal{F}_1) \cap \mathit{impls}(\mathcal{F}_2) = \emptyset$, rule ($\mathcal{F}3$) applies.

This concludes the proof. □

Proposition 2. *If variability model $\mathcal{S}$ is well-formed then $sd(\mathcal{S}) = 1$.*

Proof. We show $sd'(\mathcal{S}) = |\mathit{impls}(\mathcal{S})|$ by structural induction. First, let $\mathcal{S}$ be a ground model with common set M_C. We have:

$$\begin{aligned} & sd'(M_C) \\ &= |M_C| && \{\text{Def. 6}\} \\ &= |\mathit{impls}(M_C)| && \{\text{Def. 5}\} \end{aligned}$$

Next, let $\mathcal{S}$ be a variability model $(M_C, \{VP_1, \ldots, VP_n\})$ with variation points $VP_i = \{\mathcal{S}_{i,j} \mid 1 \leq j \leq k_i\}$. As the induction hypothesis, assume the result holds for all $\mathcal{S}_{i,j}$. We have:

$$\begin{aligned} & sd'((M_C, \{VP_1, \ldots, VP_n\})) \\ &= |M_C| + \Sigma_{1 \leq i \leq n} \Sigma_{1 \leq j \leq k_i} sd'(\mathcal{S}_{i,j}) && \{\text{Def. 6}\} \\ &= |M_C| + \Sigma_{1 \leq i \leq n} \Sigma_{1 \leq j \leq k_i} |\mathit{impls}(\mathcal{S}_{i,j})| && \{\text{Ind. hyp.}\} \\ &= |M_C \cup \textstyle\bigcup_{1 \leq i \leq n} \bigcup_{1 \leq j \leq k_i} \mathit{impls}(\mathcal{S}_{i,j})| && \{\text{Def. 7}\} \\ &= |\mathit{impls}((M_C, \{VP_1, \ldots, VP_n\}))| && \{\text{Def. 5}\} \end{aligned}$$

This concludes the proof. □

Proposition 3. *For a given SHVM, let AND and OR denote the maximum branching factors at SHVM and variation point nodes, respectively, and let ND*

be its nesting depth. The number of products induced by the SHVM is bound by $OR^{\frac{AND \cdot (AND^{ND}-1)}{AND-1}}$ *and is thus exponential in the size of the SHVM, which is bound by* $\frac{(OR \cdot AND)^{(ND+1)}-1}{OR \cdot AND-1}$.

Proof. The bounds on the number of products and size of an SHVM is obtained by solving the following recurrence equations in a routine fashion.

$$T(0) = T_0$$
$$T(n) = OR \cdot T(n-1)^{AND}$$

$$T(0) = T_0$$
$$T(n) = OR \cdot AND \cdot T(n-1) + 1$$

Proposition 4. *If variability model* $\mathcal{S}$ *is well-formed, then* $family(\mathcal{S})$ *is simple.*

Proof. By structural induction. First, let $\mathcal{S}$ be a well-formed ground model with common artifact implementations M_C. In that case $family(M_C)$ has a single product M_C that implements artifact names at most once. Then M_C can be represented as a product union over its artifact implementations taken as single-product families, and is hence simple.

Next, let $\mathcal{S}$ be a well-formed variability model $(M_C, \{VP_1, \ldots, VP_n\})$ with variation points $VP_i = \{\mathcal{S}_{i,j} \mid 1 \leq j \leq k_i\}$. As the induction hypothesis, assume the result holds for all $\mathcal{S}_{i,j}$. Since $\mathcal{S}$ is well-formed, so are all $\mathcal{S}_{i,j}$ by Definition 7, and hence, by the induction hypothesis, all families $family(\mathcal{S}_{i,j})$ are simple. For every variation point VP_i we have

$$family(VP_i) = \bigcup_{1 \leq j \leq k_i} family(\mathcal{S}_{i,j})$$

by Definition 8. Further, by well-formedness constraint ($\mathcal{S}3$) of Definition 7, we have that $names(\mathcal{S}_{i,j_1}) = names(\mathcal{S}_{i,j_2})$ for all i, j_1, j_2, and $impls(\mathcal{S}_{i,j_1}) \cap impls(\mathcal{S}_{i,j_2}) = \emptyset$ whenever $j_1 \neq j_2$. Hence, by formation rule ($\mathcal{F}3$) of Definition 3, all $family(VP_i)$ are simple. Furthermore, we have

$$family(\mathcal{S}) = \{M_C\} \bowtie \prod_{1 \leq i \leq n} family(VP_i)$$

by Definition 8. Further, by well-formedness constraint ($\mathcal{S}2$) of Definition 7, we have that $names(M_C) \cap names(VP_i) = \emptyset$ for all i, and $names(VP_{i_1}) \cap names(VP_{i_2}) = \emptyset$ whenever $i_1 \neq i_2$. Now, M_C is simple due to well-formedness constraint ($\mathcal{S}1$) of Definition 7 (see base case), and since all $family(VP_i)$ are simple, by formation rule ($\mathcal{F}2$) of Definition 3, $family(\mathcal{S})$ is also simple. □

Proposition 5. *If* $\mathcal{F}$ *is a simple non-core family in canonical form then for all* i, $1 \leq i \leq n$, *and* $k_i \geq 2$ *all* $\mathcal{F}_{i,j}$ *are simple and of strictly smaller size than* $\mathcal{F}$.

Proof. For every i, by Proposition 1, $\mathcal{F}_i$ is simple. Furthermore, by condition ($\mathcal{C}2$), all names of $\mathcal{F}_i$ are correlated, and hence, by Proposition 1, $\mathcal{F}_i$ is not

formed by rule ($\mathcal{F}2$). Since $\mathcal{F}$ is non-core, $\mathcal{F}_i$ is also non-core and is therefore formed by ($\mathcal{F}3$). Hence, again by Proposition 1, there are at least two equivalence classes of $\mathit{impls}(\mathcal{F}_i)$ w.r.t. implementation sharing $N^*_{\mathcal{F}_i}$, and thus $k_i \geq 2$.

That all $\mathcal{F}_{i,j}$ are simple is guaranteed by the three properties of simple families stated in Proposition 1 that match the three conditions in Definition 9.

That all $\mathcal{F}_{i,j}$ are strictly smaller is enforced through the formation rules for simple families from Definition 3: rule ($\mathcal{F}1$) requires the existence of a shared artifact implementation, rule ($\mathcal{F}2$) requires at least two equivalence classes on names, and rule ($\mathcal{F}3$) requires at least two equivalence classes on implementations, and thus the decomposition into canonical form is never trivial. □

Proposition 6. *If family $\mathcal{F}$ is simple, then $\mathit{shvm}(\mathcal{F})$ is well-formed.*

Proof. By induction on the size of $\mathcal{F}$. First, let $\mathcal{F}$ be a core $\{P\}$. Then, by Definition 10, $\mathit{shvm}(\mathcal{F}) = P$, which is a well-formed variability model.

Next, let $\mathcal{F}$ be a non-core family decomposed into canonical form. As the induction hypothesis, assume the result holds for all families smaller than $\mathcal{F}$. We have:

$$\begin{aligned} & \mathit{shvm}(\{P\} \bowtie \textstyle\prod_{1\leq i\leq n} \bigcup_{1\leq j\leq k_i} \mathcal{F}_{i,j}) \\ = {} & (P, \{\, VP_1, \ldots, VP_n\}) && \{\text{Def. 10}\} \\ & \text{where } VP_i = \{\mathit{shvm}(\mathcal{F}_{i,j}) \mid 1 \leq j \leq k_i\} \end{aligned}$$

By Proposition 5, all $\mathcal{F}_{i,j}$ are simple and strictly smaller than $\mathcal{F}$ and hence, by the induction hypothesis, all $\mathit{shvm}(\mathcal{F}_{i,j})$ are well-formed variability models. Now, since $\mathcal{F}$ is in canonical form, conditions ($\mathcal{C}1$) to ($\mathcal{C}3$) hold, ensuring the well-formedness constraints ($\mathcal{S}1$) to ($\mathcal{S}3$), respectively, and hence also $\mathit{shvm}(\mathcal{F})$ is a well-formed variability model. □

Lemma 1. *For every simple family $\mathcal{F}$ we have:*

$$\mathit{family}(\mathit{shvm}(\mathcal{F})) = \mathcal{F}$$

Proof. By induction on the size of $\mathcal{F}$. First, let $\mathcal{F}$ be a core $\{P\}$. We have:

$$\begin{aligned} & \mathit{family}(\mathit{shvm}(\{P\})) \\ = {} & \mathit{family}(P) && \{\text{Def. 10}\} \\ = {} & \{P\} && \{\text{Def. 8}\} \end{aligned}$$

Next, let $\mathcal{F}$ be a non-core family decomposed into canonical form presented as above. As the induction hypothesis, assume the result holds for all families

smaller that $\mathcal{F}$, and thus, by Proposition 5, for all $\mathcal{F}_{i,j}$. We have:

$$\begin{aligned}
& \mathit{family}(\mathit{shvm}(\{P\} \bowtie \textstyle\prod_{1\le i\le n} \bigcup_{1\le j\le k_i} \mathcal{F}_{i,j})) \\
&= \mathit{family}((P, \{VP_1, \ldots, VP_n\})) && \{\text{Def. 10}\} \\
&\quad \text{where } VP_i = \{\mathit{shvm}(\mathcal{F}_{i,j}) \mid 1 \le j \le k_i\} \\
&= \{P\} \bowtie \textstyle\prod_{1\le i\le n} \bigcup_{1\le j\le k_i} \mathit{family}(\mathit{shvm}(\mathcal{F}_{i,j})) && \{\text{Def. 8}\} \\
&= \{P\} \bowtie \textstyle\prod_{1\le i\le n} \bigcup_{1\le j\le k_i} \mathcal{F}_{i,j} && \{\text{Ind. hyp.}\}
\end{aligned}$$

This concludes the proof of the lemma. □

Lemma 2. *For every well-formed variability model $\mathcal{S}$ we have:*

$$\mathit{shvm}(\mathit{family}(\mathcal{S})) = \mathcal{S}$$

Proof. By structural induction. First, let $\mathcal{S}$ be a ground model with common set M_C. We have:

$$\begin{aligned}
& \mathit{shvm}(\mathit{family}(M_C)) \\
&= \mathit{shvm}(\{M_C\}) && \{\text{Def. 8}\} \\
&= M_C && \{\text{Def. 10}\}
\end{aligned}$$

Next, let $\mathcal{S}$ be a variability model $(M_C, \{VP_1, \ldots, VP_n\})$ with variation points $VP_i = \{\mathcal{S}_{i,j} \mid 1 \le j \le k_i\}$. As the induction hypothesis, assume the result holds for all $\mathcal{S}_{i,j}$. We have:

$$\begin{aligned}
& \mathit{shvm}(\mathit{family}((M_C, \{VP_1, \ldots, VP_n\}))) \\
&= \mathit{shvm}(\{M_C\} \bowtie \textstyle\prod_{1\le i\le n} \bigcup_{1\le j\le k_i} \mathit{family}(\mathcal{S}_{i,j})) && \{\text{Def. 8}\} \\
&= (M_C, \{VP'_1, \ldots, VP'_n\}) && \{\text{Def. 10}\} \\
&\quad \text{where } VP'_i = \{\mathit{shvm}(\mathit{family}(\mathcal{S}_{i,j})) \mid 1 \le j \le k_i\} \\
&= (M_C, \{VP'_1, \ldots, VP'_n\}) && \{\text{Ind. hyp.}\} \\
&\quad \text{where } VP'_i = \{\mathcal{S}_{i,j} \mid 1 \le j \le k_i\} \\
&= (M_C, \{VP_1, \ldots, VP_n\}) && \{\text{Def. } \mathcal{S}\}
\end{aligned}$$

To justify the second step above we need to show that

$$\{M_C\} \bowtie \textstyle\prod_{1\le i\le n} \bigcup_{1\le j\le k_i} \mathit{family}(\mathcal{S}_{i,j})$$

is in canonical form. This is established as follows, using that $\mathcal{S}$ is simple. First, the restriction that variation points have at least two variants and the constraint ($\mathcal{S}3$) guarantee that just the artifact implementations in M_C and no other artifact implementations are shared by all products of $\mathcal{S}$, and thus condition ($\mathcal{C}1$) is satisfied.

Next, constraint ($\mathcal{S}2$) guarantees that artifact names implemented by different variation points are orthogonal. On the other hand, the restriction that variation points have at least two variants and the constraint ($\mathcal{S}3$) guarantee that artifact names implemented by the same variation point must be correlated, and thus condition ($\mathcal{C}2$) is satisfied.

And finally, constraint ($\mathcal{S}3$) guarantees that variants do not share any artifact implementation. On the other hand, the restriction guarantees that any two products of the same variant share an artifact implementation, and thus condition ($\mathcal{C}3$) is satisfied. This concludes the proof of the lemma. □

Theorem 2. *Let $\mathcal{S}$ be an SHVM with global property ϕ. If the verification procedure succeeds for $\mathcal{S}$, then $p \models \phi$ for all its products $p \in \mathit{products}(\mathcal{S})$.*

Proof. The proof is by induction on the structure of $\mathcal{S}$. For the base case, let $\mathcal{S}$ be a ground model with common set M_C. Assume the verification procedure succeeds for $\mathcal{S}$. It has then established: $\biguplus_{a \in Art(M_C)} \mathcal{G}_a \models \phi$. From this, and by soundness of rule (1), it follows that $M_C \models \phi$. Since $\mathit{products}(\mathcal{S}) = \{M_C\}$ in this case, we have $p \models \phi$ for all $p \in \mathit{products}(\mathcal{S})$.

For the induction step, let $\mathcal{S}$ be a non-ground model $(M_C, \{VP_1, \ldots, VP_n\})$ with variation points $VP_i = \{\mathcal{S}_{i,j} \mid 1 \leq j \leq k_i\}$, where k_i is the number of variants of VP_i. Further, let (ψ_{VP_i}, I_{VP_i}) be the specification of VP_i. Assume the result for all $\mathcal{S}_{i,j}$ (induction hypothesis). Next, assume that the verification procedure succeeds for $\mathcal{S}$. The following has then been established for the top-level module:

(i) $\biguplus_{a \in Art(M_C)} \mathcal{G}_a \uplus \biguplus_{1 \leq i \leq n} \mathcal{M}ax(\psi_{VP_i}, I_{VP_i}) \models \phi$

By the assumption, the verification procedure has also succeeded for all $\mathcal{S}_{i,j}$. Thus, by the induction hypothesis, and since the SHVM nodes of variants attached to a variation point inherit the corresponding variation point specification, we have:

$$\forall i : 1 \leq i \leq n.\, \forall j : 1 \leq j \leq k_i.\, \forall p \in \mathit{products}(\mathcal{S}_{i,j}).\, p \models \psi_{VP_i}$$

By Definition 8 we have $\mathit{products}(VP_i) = \bigcup_{1 \leq j \leq k_i} \mathit{products}(\mathcal{S}_{i,j})$, and hence:

(ii) $\forall i : 1 \leq i \leq n.\, \forall p \in \mathit{products}(VP_i).\, p \models \psi_{VP_i}$

Also by Definition 8, we know that every product p of $\mathcal{S}$ is the union of the core M_C and exactly one subproduct from every variation point. Due to (ii), all subproducts meet their respective specifications. Also, by (i) and from soundness of rule (1) follows that $p \models \phi$. This concludes the proof. □

Appendix C

Paper III: Algorithmic Verification of Procedural Programs in the Presence of Code Variability

Algorithmic Verification of Procedural Programs in the Presence of Code Variability

Siavash Soleimanifard and Dilian Gurov

KTH Royal Institute of Technology, Stockholm, Sweden
{siavashs,dilian}@csc.kth.se

Abstract. We present a generic framework for verifying temporal safety properties of procedural programs that are dynamically or statically configured by replacing, adapting, or adding new components. To deal with such a variability of a program, we require programmers to provide local specifications for its variable components, and verify the global properties by replacing these specifications with maximal models. Our framework is a generalization of a previously developed framework that abstracts from all program data. In this work, we capture program data and thus significantly increase the range of properties that can be verified. Our framework is generic by being parametric on the set of observed program events and their semantics. We separate program structure from the behavior it induces to facilitate independent component specification and verification. To exemplify its use, we instantiate our framework to develop a compositional verification technique for programs written in a procedural language with pointers as the only datatype. We also adapt our previously developed toolset to provide support for compositional verification of such programs.

1 Introduction

In modern computing systems code changes frequently. Components evolve rapidly or exist in multiple versions customized for different users, and in open and mobile contexts a system may even automatically reconfigure itself. As a result, systems are no longer developed as monolithic applications; instead they are composed of ready-made off-the-shelf components, and each component may be dynamically replaced by a new one that provides improved or additional functionality. The design and implementation of systems with such static and dynamic *variability* has been attracting considerable attention over the past years. However, there has been less attention to their formal verification. In this paper, we develop a generic framework for the verification of temporal safety properties of such systems.

The verification of *variable systems* is challenging because the code of the variable components is either not available at verification time or changes frequently. Therefore, an ideal verification technique for such systems should (i) *localize* the verification of variable components, and (ii) *relativize* the *global* properties of the system on the correctness of its variable components. This can be

achieved through a compositional verification scheme where system components are specified *locally* and verified independently, while the correctness of its global properties is inferred from these local specifications. As a result, this allows an independent evolution of the implementations of individual components, only requiring the re-establishment of their local correctness. An algorithmic technique for realization of this verification scheme is to replace the local specifications by so-called *maximal models* [18]. These are most general models satisfying the specifications. Thus, if such models exist, they can replace the specifications of variable components in the verification of the global properties.

The work presented in this paper is the second, final, and conceptually more complicated phase of developing a compositional verification framework for temporal properties of procedural programs with variability exploiting maximal models. In the first phase, we developed a compositional verification technique that separates program structure from its operational semantics (behavior) to allow independent evolution of components [19,21]. The technique abstracts away all program data to achieve algorithmic and practical verification. Such a drastic abstraction, while allowing the verification of certain control flow safety properties [33], significantly reduces the range of properties that can be handled. For instance, properties of sequences of method invocations such as "method m_1 is not called after method m_2 is called" can be verified, but not properties that involve program data, such as "method m_1 is called only if variable `V` is not pointing to `null`". In this work, we generalize this technique to capture program data, and thus bring the usability of our work to a whole new level.

The two main limitations of any verification technique that is based on maximal models are (i) the computationally complex maximal model construction and (ii) the difficulty of producing component specifications. In our previous works, these limitations were softened by full data abstraction. As we show in Section 2, including program data (if done in the straightforward fashion) makes the maximal model construction and property specification impractical: the program models and properties become too detailed and large, maximal model construction becomes unmanageably complex, and the program models become overly specific to one programming language. Our present proposal captures program data without adding extra complexity to the maximal model construction, and keeps the complexity of property specification within practical limits.

We define a novel notion of program structure that is parametric on a set of *actions* that model single instructions of a selected type, and a set of Hoare-style state *assertions* that capture abstractly the effect of a series of statements between consecutive actions. We combine the abstraction provided by assertions with the precision provided by actions to define a uniform control flow graph representation of programs that can be tuned for the verification of the class of properties of interest. The abstraction provided by assertions prevents the local specifications from becoming overly verbose, and allows us to capture program data without adding extra complexity to the maximal model construction. From a wider perspective, by providing Hoare-style assertions and precise ordering of actions these models allow to combine Hoare-style with temporal logic reasoning.

Additionally, we present three instantiations of our generic verification framework. The first instantiation abstracts away all data as in the original CVPP framework. The second one is an instantiation for the verification of *Boolean programs* [10]. The capability of handling Boolean data shows that our generic framework can handle data from finite domains. In a third and most challenging instantiation we exemplify the use of our framework for the verification of programs written in a procedural language with pointers as the only datatype (PoP). Dealing with this language is challenging because, in addition to unbounded call stacks, it can give rise to infinite state spaces for yet another reason, namely unbounded pointer creation. This instantiation shows how our framework can cope with data from infinite domains.

To the extent of our knowledge, our previous framework and consequently the one presented in this paper are the only ones for algorithmic verification of temporal properties of procedural languages that allow the proofs to be relativized on component specifications. From a technical point of view, the main *contributions* of this paper compared to our previous works are: (i) a novel structural model that combines the precise ordering of selected instructions with abstract representation of the remaining ones, and its operational semantics (a behavioral model), (ii) a proof that the the original maximal model construction can be adapted for the case with data (possibly from infinite domains) with minimal additional cost, (iii) a proof of the correctness of the technique by (non-trivial) re-establishment of our previous results, (iv) an instantiation of the generic framework for a procedural language with pointer datatype (PoP), and (v) tool support for an instantiation of the framework to PoP programs.

In the remainder of this paper, Section 2 provides an overview of our technique and illustrates some variability scenarios on an example. Sections 3, 4, and 5 define our program models, specification languages, and maximal models. In Section 6 we spell-out our compositional verification principle. Sections 7, 8, and 9 present the instantiations of our framework with full data abstraction, for Boolean programs, and for PoP programs, respectively. Finally, in Section 10 we discuss related work and draw conclusions.

2 Overview of the Approach

This section provides an overview of our framework by demonstrating its use on an example that mimics the method invocation style of real-life web applications. Although the technique we propose applies to procedural languages in general, we illustrate it here on *Pointer Programs* (PoP), a language with *pointers* as the only datatype [31]. The language supports pointer creation and deletion, assignments and conditional statements, loops, and method-calls with call-by-reference parameter passing. The statement `new x` allocates a fresh chunk of memory and assigns its pointer to variable `x`, while `del x` deletes the memory that `x` is pointing to and assigns `null` to `x` (and all its aliases). The guards for the conditional statements and loops are equality (alias) and inequality checks on variables, and non-deterministic choice, denoted by `*`. Being able to deal

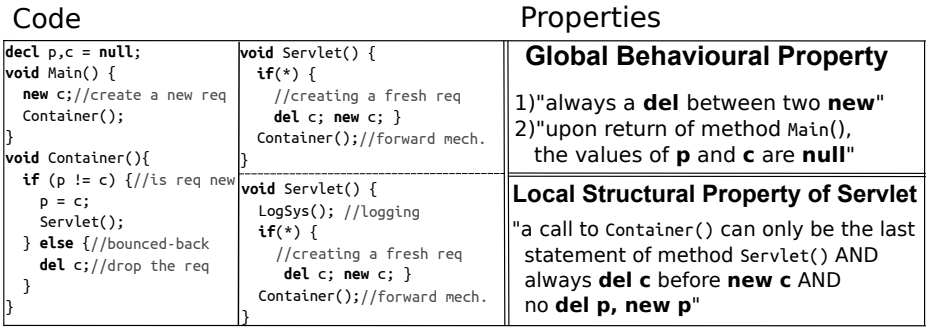


Fig. 1: Web Server Application

with this language is of interest, since it can give rise to infinite state spaces, for two reasons: unbounded stacks of procedure calls, and unbounded pointer creation. The formal instantiation of our generic verification framework to the PoP language is given in Section 9.

We use this language to implement a program that mimics the method invocation style of Java enterprise web applications. The execution of such applications starts in method `Container` where based on the current request a *Servlet* is called to prepare the output. As a coding standard [23], servlets should not call each other. Thus if for example servlet A needs to make use of servlet B, it forwards a request to the `Container` that triggers a call to B. We model this so-called forwarding mechanism by explicit invocation of `Container` in servlets.

The program in Figure 1 provides an implementation of a container and two implementations of a single servlet, in which the one at the bottom extends the one at the top by adding a logging facility through calling method `LogSys`. In the code, the variables are pointers to requests. The global variables `p` and `c` point to the previous (last-received) and current requests, respectively. At the beginning of the execution, the request `c` is initialized by `Main` and `Container` is called. In `Container` if the current request is different with the previous one, the current request is stored in `p` and method `Servlet` is called, which non-deterministically generates a fresh request and calls back `Container`. By this, we mimic the call-backs to `Container` (forwarding mechanism) in a real web-application when servlets call each other via the container. `Container` drops (i.e., deletes) the requests that are bounced back to it (when `p = c`) to avoid cycles in the computation. The code of method `LogSys` is not shown here, but we assume that it does not modify the global variables. Here, we consider each method as a component, but in general a component can consist of several methods.

In this example, we assume that the method `Servlet` is the variable part of the program. The structural local specification of method `Servlet` and two behavioral global properties are given in the figure. In the remainder of this section, we explain how to apply to this program the verification technique developed in the later sections, in different variability scenarios.

Verification Technique. In our framework, we divide the verification of variable programs into two independent sub-tasks:

(i) a check that the implementation of each variable component satisfies its local specification, and
(ii) a check that the composition of the local specifications together with the implementations of the non-variable components entails the global property.

By this division we localize the verification of variable components (with sub-task (i)), and relativize the correctness of global properties of the program on the local specifications of its variable components (with sub-task (ii)). Thus, adding or changing the implementation of a variable component does not require the global property to be re-verified, just its local specification (with sub-task (i)). Also notice that, if the local specifications are specified as completely as possible (*i.e.*, are not tailored toward particular global properties), once the local checks of sub-task (i) are performed, the verification of new global properties will not require the re-specification and verification of variable components. In fact, variable components are often implemented and specified as general-purpose libraries that can be used in arbitrary contexts and should thus not be specified toward specific global properties.

In most variability scenarios, variable systems would be verified once (with sub-tasks (i) and (ii)) before delivering the software to customers, and would be re-verified every time a variable component is modified by performing sub-task (i) on the customer's side. Ideally, sub-task (i) should be performable quickly and thus in isolation from the non-variable part of the system, which is (usually) significantly larger than the modified component. This is difficult to achieve for local specifications that express properties of the execution of programs, *i.e. behavioral* specifications, but is natural for those that express properties of the code (program text) itself, *i.e. structural* specifications. The reason is that the latter can be checked against the component's code rather than the execution of the whole program. For example, a behavioral specification of method `Servlet` would be "`c` points to `null` at any return point of method `Servlet`", which cannot be checked for method `Servlet` in isolation from `Container` and `LogSys`, while the structural specification given in Figure 1 can be checked against `Servlet`'s code, independent from the rest of the program. In practice, these local specifications should be provided by the developers. . This requires the knowledge of the safety requirements of the system.

Let us now mimic some dynamic variability scenarios. First assume that no implementation of `Servlet` is available, for example because it is not implemented yet or should be imported from a third-party library. Still, the incomplete program can be verified from the given structural local property of method `Servlet` and the implementation of methods `Main` and `Container` by performing sub-task (ii). Later, when the implementation of method `Servlet` at the top becomes available, it is only checked against its specification, as in sub-task (i). Now assume that, after a while, the implementation of `Servlet` is updated to the one at the bottom. Again, only the local check of sub-task (i) needs to be performed, this time for the new implementation.

For static variability scenarios, assume that the two implementations of `Servlet` are available and each of them together with `Container` make an application that is delivered to customers based on their needs and budget (as in product families). To verify the global property, the local specification of method `Servlet` is checked for each of the implementations separately (sub-task (i)). Independently, the composition of this local specification with the implementation of `Container` is checked against the global property (sub-task (ii)).

To verify programs in such variability scenarios, we model the structure of non-variable components with *flow graphs*, and convert local specifications of the variable components to *maximal flow graphs*. Here, we present these notions informally and describe how they are used in our verification framework.

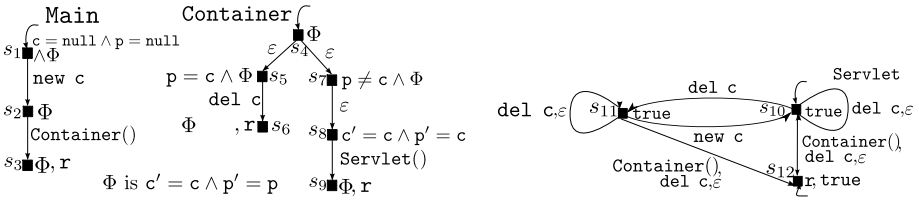
Flow Graphs. A flow graph is a finite collection of *method graphs*, each of which represents the control flow structure of a method. Our flow graphs are parametric on the class of program instructions that need to be explicitly represented for the verification of the properties of interest, while using an abstract representation of all other instructions. The rationale is that in temporal reasoning one is usually interested in the ordering of certain events of interest, here called *actions*. The exact ordering of the other events can be abstracted away; only their cumulative effect needs to be captured. We represent the effect of a series of consecutive events between two actions in a Hoare style, through logical *assertions*. The combination of the precise ordering of actions and abstract representation of data provided by assertions yields a flexible program model that potentially allows to combine Hoare-style with temporal logic reasoning. Here, however, we use these models only for the verification of control flow properties.

In our flow graphs, the actions have parameters and are represented by transition labels, while the assertions are assigned to control nodes[1]. Besides assertions, return nodes are tagged by the atomic proposition `r`. Entry nodes of method graphs represent the beginning of methods.

As an example, Figure 2a shows a flow graph of the code of methods `Main` and `Container`. We want to verify properties talking about order of `new` and `del` statements, e.g., global properties in the figure, thus in this example, actions are `new` and `del`. We add a neutral action ε to simplify the presentation of the flow graphs. Assertions are equality and inequality checks on the variables at the beginning and the end of a block of code between two actions. They express the cumulative effect of condition evaluation and assignments. We use variable names (such as `p` and `c`) and their primed version (`p`$'$ and `c`$'$) to refer to the values at the beginning and the end of blocks, respectively. For example, state s_8 in the figure represents the assignment statement `p := c` in the code of `Container`.

Maximal Flow Graphs. A maximal flow graph for a specification is a flow graph that represents the structure of *any* code satisfying it. To verify global properties, in our framework the variable components are replaced with maximal flow graphs

[1] This (maybe non-standard) design choice allows a clear distinction between actions and assertions, which is crucial for our framework.



(a) Flow Graphs of `Main` and `Container` (b) Maximal Flow Graph of `Servlet`

Fig. 2

constructed from their specifications (in sub-task (ii)). By this, we decouple the concrete implementations of variable components from the global correctness reasoning, thus allowing independent evolution of their code. In Section 5, we define formally maximal flow graphs, prove their existence and uniqueness for our specification logic, and provide an algorithm to construct them. Here, we only give an intuitive explanation of their specifics in the present setup.

Local specifications often specify constraints on a small subset of the program variables only, namely the variables whose values should be captured for the verification of the class of properties of interest. For example, the specification of method `Servlet` does not specify any constraint on the variables `p` and `c` since their values don't have any effect on the global properties. In such situations, there are (possibly infinitely) many implementations for a component that respect its specification. A maximal flow graph should capture the structure of all these implementations. It is therefore of size exponential in the number of unspecified variables and their values, and is thus infeasible to construct in practice with standard algorithms, e.g., [18,26,19], where data is represented concretely.

In our structural models, however, data is represented symbolically through logical assertions. We use a *semantic entailment* relation on assertions to reduce the size and complexity of the construction of the maximal flow graphs. The idea is that a control node with assertion ϕ can represent any set of nodes that are tagged with assertions entailing ϕ. For example, consider the maximal flow graph constructed from the local specification of `Servlet` shown in Figure 2b. In the graph the assertions (`true`) do not specify any constraints on the variables, so any similar flow graph that for example has $\mathtt{c}' = \mathtt{c}$ or $\mathtt{p}' = \mathtt{p}$ as assertions at its control nodes will be represented by the given maximal flow graph.

Verification. In our framework we support verification of structural and behavioral global properties by performing the sub-tasks (i) and (ii) as follows. (i) The flow graph extracted from the available implementation of `Servlet` is model checked against its local specification. (ii) The maximal flow graph of `Servlet` and the flow graph of `Container` are composed by means of set-theoretic union. This composition can be directly model checked against structural global properties. However, the verification of behavioral global properties requires that a behavioral model is induced from the composition. Intuitively this model (called *flow graph behavior*) should capture all possible runs (executions) of the flow

graph. Therefore, it should model the call stack and represent the values of variables at each point of the execution, in which the latter requires the semantics of the transition labels and state assertions. Also, to allow model checking of such models, values should be from finite domains. Then the model can be represented by means of pushdown automata. These models are defined in Section 3. As we shall see in Section 9, the second global behavioral property given in Figure 1 does not hold for the local specification of method `Servlet` and flow graph of methods `Container` and `Main`.

3 Program Model

We first define an abstract notion of *model* on which our representations of program structure and behavior are based. A model is a *Kripke* structure extended with transition labels and a set of state assertions.

Definition 1 (Model). *A* model *is a tuple* $\mathcal{M} = (S, L, \rightarrow, A, P, \lambda_A, \lambda_P)$ *where* S *is a set of states,* L *a set of labels,* $\rightarrow \subseteq S \times L \times S$ *a labeled transition relation,* A *a finite set of* atomic propositions *(or* atoms*),* P *a finite set of* state assertions*,* $\lambda_A : S \rightarrow 2^A$ *and* $\lambda_P : S \rightarrow P$ *valuations assigning to each state a set of atoms and a state assertion, respectively. An* initialized model $\mathcal{S}$ *is a pair* $(\mathcal{M}, E)$ *with* $\mathcal{M}$ *a model and* $E \subseteq S$ *a set of initial states.*

Models are composed through disjoint union $\uplus$. We assume the set of state assertions P to be equipped with a semantic entailment relation, denoted by $\sqsubseteq$. This relation is used to define simulation preorder, logical satisfaction, and maximal model construction.

In contrast to models without data, the states of models with data are additionally tagged with state assertions. As we shall see, these assertions together with the atomic propositions provide the basis for the symbolic and concrete representation of data, respectively. State assertions are used in structural models to capture how data may change at the states (nodes) of the model, while atomic propositions are used in behavioral models to represent the values of variables at each point of the program execution.

We mentioned that a maximal model is the most general model satisfying a property. The generality relation on models is technically defined w.r.t. a preorder relation called *simulation*. The definition of simulation preorder is parametric on the semantic entailment $\sqsubseteq$.

Definition 2 (Simulation). *A* simulation *on* S *is a binary relation* R *on* S *such that whenever* $(s,t) \in R$ *then* $\lambda_A(s) = \lambda_A(t)$*,* $\lambda_P(s) \sqsubseteq \lambda_P(t)$*, and whenever* $s \xrightarrow{a} s'$ *then there is some* $t' \in S$ *such that* $t \xrightarrow{a} t'$ *and* $(s', t') \in R$*. We say that* t *simulates* s*, written* $s \leqslant t$*, if there is a simulation* R *such that* $(s,t) \in R$*.*

Simulation on two models $\mathcal{M}_1$ and $\mathcal{M}_2$ is defined as simulation on their disjoint union $\mathcal{M}_1 \uplus \mathcal{M}_2$. The transitions of $\mathcal{M}_1 \uplus \mathcal{M}_2$ are defined by $in_i(s) \xrightarrow{a} in_i(s')$ if $s \xrightarrow{a} s'$ in $\mathcal{M}_i$ and its valuation by $\lambda(in_i(S)) = \lambda_i(S)$, where in_i (for $i \in \{1, 2\}$)

injects S_i into $S_1 \uplus S_2$. Simulation is extended to initialized models $(\mathcal{M}_1, E_1)$ by defining $(\mathcal{M}_1, E_1) \leqslant (\mathcal{M}_2, E_2)$ if there is a simulation R on $\mathcal{M}_1 \uplus \mathcal{M}_2$ such that for each $s \in E_1$ there is some $t \in E_2$ with $(in_1(s), in_2(t)) \in R$. Initialized model $\mathcal{S}_1$ is simulation equivalent to $\mathcal{S}_2$, written $\mathcal{S}_1 \simeq \mathcal{S}_2$ if $\mathcal{S}_1 \leqslant \mathcal{S}_2$ and $\mathcal{S}_2 \leqslant \mathcal{S}_1$. We extend disjoint union to initialized models (by $(\mathcal{M}_1, E_1) \uplus (\mathcal{M}_2, E_2) = (\mathcal{M}_1 \uplus \mathcal{M}_2, E_1 \uplus E_2)$).

As mentioned earlier, we compose models to verify global properties. The following theorem establishes that simulation is preserved by model composition.

Theorem 1 (Monotonicity). *If $\mathcal{S}_1 \leqslant \mathcal{S}_1'$ and $\mathcal{S}_2 \leqslant \mathcal{S}_2'$ then $\mathcal{S}_1 \uplus \mathcal{S}_2 \leqslant \mathcal{S}_1' \uplus \mathcal{S}_2'$.*

Proof. Suppose R_1 and R_2 are witnesses of $\mathcal{S}_1 \leqslant \mathcal{S}_1'$ and $\mathcal{S}_2 \leqslant \mathcal{S}_2'$, respectively. Then $R = \{((s,i),(t,i)) \mid i \in \{1,2\} \wedge (s,t) \in R_i\}$ is a simulation between $\mathcal{S}_1 \uplus \mathcal{S}_2$ and $\mathcal{S}_1' \uplus \mathcal{S}_2'$.

Next, we define formally our *flow graphs* for representing program structure. Here, for the sake of simplicity, we only consider one datatype in our formalizations. However, in a more general setting, where the program has different datatypes, a set of state assertions is defined for each datatype. The results easily generalize for this case.

3.1 Flow Graphs

Intuitively, a *flow graph* is a collection of *method graphs*, one for each method of the program, as illustrated in Figure 2a. W.l.o.g., we assume that method names are distinct and taken from a countably infinite set of method names *Meth*. The notion of method graph is an instance of the generic notion of initialized model defined above, with particular sets of assertions P and labels L. Let $\mathcal{A}$ be a set of *actions* with data parameters. The set of flow graph labels is $L = L_{\mathcal{A}} \cup L_{call}$, where $L_{\mathcal{A}} = \{\alpha(a_1, ..., a_n) \mid \alpha \in \mathcal{A}\}$ are action-induced labels and $L_{call} = \{m(a_1, \ldots, a_w) \mid m \in Meth\}$ are labels representing method invocations. where a_i is an actual parameter for formal parameter p_i

Definition 3 (Method Graph). *A* method graph *for method name $m \in Meth$ over a set $M \subseteq Meth$ of method names is an initialized model $(\mathcal{M}_m, E_m)$ where $\mathcal{M}_m = (S_m, L_m, \rightarrow_m, A_m, P_m, \lambda_{A_m}, \lambda_{P_m})$ is a finite model and $E_m \subseteq S_m$ is a non-empty set of* entry points *of m. S_m is the set of* control nodes *of m, $L_m \subseteq L$, $A_m = \{m, r\}$, $P_m \subseteq P$, $\lambda_{P_m} : S_m \rightarrow P_m$ is a valuation for transition propositions, and $\lambda_{A_m} : S_m \rightarrow \{\{m\}, \{m, r\}\}$ is a valuation for atoms so that each node is tagged with its method name, and* return nodes *are additionally tagged with r.*

We sometimes write $s \models m$ to denote $m \in \lambda_A(s)$. Notice that with the above definition, control nodes of flow graphs do not in general correspond to single program points in the actual program's code, but rather to sets of them.

In contrast to the flow graphs defined here, the ones without data do not have state assertions, because all variables and their values are abstracted away.

Every flow graph $\mathcal{G}$ is equipped with an *interface* I, denoted by $\mathcal{G} : I$. A *flow graph interface* consists of a triple $I = (M^+, M^-, \mathit{Modify})$, where $M^+, M^- \subseteq \mathit{Meth}$ are finite sets of *provided* and *required* methods, respectively, and *Modify* is the set of the global variables of the program that are modified in the code of the provided methods. The *composition* of two interfaces is defined as their disjoint union. As we shall see, interfaces are needed when constructing maximal flow graphs, which in turn are used for compositional verification.

The definition of *flow graph simulation*, denoted by $\leqslant_s$, is an instantiation of the general notion of simulation on models (see Definition 2) to flow graphs.

3.2 Flow Graph Behavior

Program states $\sigma \in \Sigma$ are defined as usual as mappings from the set of program variables V to their values taken from $\mathcal{D}$. Behavioral transitions conceptually capture the occurrence of actions together with data transformations as specified by assertions. An assertion ϕ is interpreted over pairs of program states, written $(\sigma, \sigma') \models \phi$, and is defined to hold when the closed formula $\phi[\sigma, \sigma']$ is logically valid (here σ, σ' are used as syntactic substitutions for the non-primed and primed variables, respectively). We define *behavioral states* $\langle s, \sigma, \sigma' \rangle$ as consisting of a control node and a pair of program states that satisfies the assertion of the node.

Next we define flow graph behavior. Behavioral transitions are labeled with "m_1 `call` $m_2(a_1, \ldots, a_w)$" for an invocation of method m_2 by method m_1 with parameters $a_1, \ldots, a_w$, "m_2 `ret` m_1" for the corresponding return from the call, or $\alpha(a_1, ..., a_n) \in L_{\mathcal{A}}$ for the (method-local) transfer of control by action α with parameters $a_1, ..., a_n$. The definition of flow graph behavior is parametric on externally provided (denotational) semantic mappings $[\![\cdot]\!]$ over states and state pairs that specify the (local) effect of actions, calls and returns.

Definition 4 (Flow Graph Behavior). *Let* $\mathcal{S} = (\mathcal{M}, E) : (M^+, M^-, \mathit{Modify})$ *be a flow graph s.t.* $\mathcal{M} = (S, L, \rightarrow, A, P, \lambda_A, \lambda_P)$. *The behavior of* $\mathcal{S}$ *is defined as the initialized model* $b(\mathcal{S}) = (\mathcal{M}_b, E_b)$ *where* $\mathcal{M}_b = (S_b, L_b, \rightarrow_b, A_b, P_b, \lambda_{A_b}, \lambda_{P_b})$, *with* $S_b \subseteq (S \times \Sigma \times \Sigma) \times (S \times \Sigma)^*$, *i.e., states (or configurations) are pairs of behavioral states* $\langle s, \sigma, \sigma' \rangle$ *and stacks* γ *over pairs of control nodes and program states,* $L_b = L_{\mathcal{A}} \cup \{m_1 \ \mathtt{call}\ l_{m_2} \mid m_1 \in M^+ \wedge l_{m_2} \in L_{call}\} \cup \{m_1 \ \mathtt{ret}\ m_2 \mid m_1, m_2 \in M^+\}$, $A_b = A \cup (\Sigma \times \Sigma)$, $P_b = \{\mathtt{tt}\}$, $\lambda_{A_b}(\langle s, \sigma, \sigma' \rangle, \gamma) = \lambda_A(s) \cup \{(\sigma, \sigma')\}$, *and* $\rightarrow_b \subseteq S_b \times L_b \times S_b$ *is defined by the following transition rules:*

$$[\alpha] \quad (\langle s_1, \sigma_1, \sigma_1' \rangle, \gamma) \xrightarrow{\alpha(\sigma'(a_1), \ldots, \sigma'(a_n))} (\langle s_2, \sigma_2, \sigma_2' \rangle, \gamma) \qquad \textit{if } s_1 \xrightarrow{\alpha(a_1, \ldots, a_n)} s_2 \wedge (\sigma_1, \sigma_1') \models \lambda_P(s_1) \wedge (\sigma_2, \sigma_2') \models \lambda_P(s_2) \wedge \sigma_2 = [\![\alpha]\!]\sigma_1'$$

$$[call] \ (\langle s_1, \sigma_1, \sigma_1' \rangle, \gamma) \xrightarrow{m \ \mathtt{call}\ m'(\sigma'(a_1), \ldots, \sigma'(a_w))} (\langle s, \sigma, \sigma' \rangle, \langle s_2, \sigma_1' \rangle \cdot \gamma) \qquad \textit{if } s_1 \xrightarrow{m'(a_1, \ldots, a_w)} s_2 \wedge s \models m' \wedge s_1, s_2 \models m \wedge m, m' \in M^+ \wedge (\sigma_1, \sigma_1') \models \lambda_P(s_1) \wedge (\sigma, \sigma') \models \lambda_P(s) \wedge \sigma_2 = [\![call]\!]\sigma_1'$$

$$\textit{[ret]}\ (\langle s_1,\sigma_1,\sigma_1'\rangle,\langle s_2,\sigma_2\rangle\cdot\gamma)\xrightarrow{m\ ret(ret)\ m'}(\langle s_2,\sigma_3,\sigma_3'\rangle,\gamma)\ \textbf{\textit{if}}\ \begin{array}{l} s_1\models r\wedge m\wedge \mathtt{ret}=v, m'\in M^+\wedge\\ s_2\models m'\wedge(\sigma_1,\sigma_1')\models\lambda_P(s_1)\wedge\\ s_1\models m\wedge(\sigma_3,\sigma_3')\models\lambda_P(s_2)\wedge\\ \sigma_3=[\![ret]\!](\sigma_1',\sigma_2)\end{array}$$

The initial configurations are $E_b=\{(\langle s,\sigma_0,\sigma_0'\rangle,\epsilon)\mid s\in E\wedge(\sigma_0,\sigma_0')\models\lambda_P(s)\}$, *where* σ_0 *and* ϵ *denote the initial program state and the empty stack, respectively.*

In the behavioral models variables are explicitly assigned to values and therefore the set of assertions P_b should be empty. However, to be faithful to Definition 1, we use the (dummy) value `tt` which we assign to all behavioral states. It should further be noted that if $\mathcal{D}$ is finite, flow graph behavior can also be defined by means of *pushdown automata*, as in [19].

In contrast to the above definition of behavior, the one without data does not have program states Σ, and the only action is ε. Thus, at calls control nodes are simply pushed to the stack and these are popped at returns. Also the set of atomic propositions A_b is equal to A, only consisting of method names and `r`.

Again, we instantiate the general definition of simulation (Definition 2) to flow graph behavior, and denote it by $\leqslant_b$. A result that we later exploit for compositional verification is that if two flow graphs are related by structural simulation, then their behaviors are related by behavioral simulation.

Theorem 2 (Simulation Correspondence). *For flow graphs* A *and* B, *if* $A\leqslant_s B$ *then* $b(A)\leqslant_b b(B)$.

Proof. Let R_s be a structural simulation between $A=(\mathcal{M}^A,E^A)$ and $B=(\mathcal{M}^B,E^B)$. We lift R_s on the structural level to R_b on the behavioral level by defining:

$$\begin{array}{l} R_b=\{((\langle s_1^A,\sigma_1^A,\sigma'^A\rangle,\gamma^A),(\langle s_1^B,\sigma_1^B,\sigma_1'^B\rangle,\gamma^B))\mid\\ \qquad (s_1^A,s_1^B)\in R_s\wedge\sigma_1^A=\sigma_1^B\wedge\sigma_1'^A=\sigma_1'^B\wedge\\ \qquad (\langle s_1^A,\sigma_1^A,\sigma'^A\rangle,\gamma^A)\in b(A)\wedge(\langle s_1^B,\sigma_1^B,\sigma_1'^B\rangle,\gamma^B)\in b(B)\wedge\\ \qquad |\gamma^A|=|\gamma^B|\wedge\forall 1\le i\le|\gamma^A|\,.\,\gamma_i^A=\langle s^A,\sigma^A\rangle\wedge\\ \qquad \gamma_i^B=\langle s^B,\sigma^B\rangle\wedge(s^A,s^B)\in R_s\wedge\sigma^A=\sigma^B\}\end{array}$$

We show that R_b is a behavioral simulation between $b(A)$ and $b(B)$.
For every entry point $s^A\in E^A$ there is an entry point $s^B\in E^B$ such that $(s^A,s^B)\in R_s$ and thus $\lambda_P(s^A)\sqsubseteq\lambda_P(s^B)$. Therefore, for each initial state $(\langle s^A,\sigma^A,\sigma'^A\rangle,\epsilon)$, there is an initial state $(\langle s^B,\sigma^B,\sigma'^B\rangle,\epsilon)$ such that

$$((\langle s^A,\sigma^A,\sigma'^A\rangle,\epsilon),(\langle s^B,\sigma^B,\sigma'^B\rangle,\epsilon))\in R_b.$$

Suppose that $((\langle s_1^A,\sigma_1^A,\sigma_1'^A\rangle,\gamma),(\langle s_1^B,\sigma_1^B,\sigma_1'^B\rangle,\gamma))\in R_b$. We now proceed by case analysis on the possible transitions from $(\langle s_1^A,\sigma_1^A,\sigma_1'^A\rangle,\gamma)$.

Case 1. (Actions) Suppose $(\langle s_1^A,\sigma_1^A,\sigma_1'^A\rangle,\gamma)\xrightarrow{\alpha(\sigma_1'^A(a_1),\ldots,\sigma_1'^A(a_n))}(\langle s_2^A,\sigma_2^A,\sigma_2'^A\rangle,\gamma)$. It follows that there is a state s_2^B such that $(s_2^A,s_2^B)\in R_s$, hence $\lambda_P(s_2^A)\sqsubseteq$

$\lambda_P(s_2^B)$ and $s_1^B \xrightarrow{\alpha(a_1,\ldots,a_n)} s_2^B$. Thus, for state $(\langle s_2^A, \sigma_2^A, \sigma_2^{'A}\rangle, \gamma)$ there is a state $(\langle s_2^B, \sigma_2^B, \sigma_2^{'B}\rangle, \gamma)$ such that

$$((\langle s_2^A, \sigma_2^A, \sigma_2^{'A}\rangle, \gamma), (\langle s_2^B, \sigma_2^B, \sigma_2^{'B}\rangle, \gamma)) \in R_b.$$

Case 2. (Call) Suppose

$$(\langle s_1^A, \sigma_1^A, \sigma_1^{'A}\rangle, \gamma) \xrightarrow{m' \texttt{ call } m(\sigma_1^{'A}(a_1),\ldots,\sigma_1^{'A}(a_n))} (\langle s_{1m}^A, \sigma_{1m}^A, \sigma_{1m}^{'A}\rangle, \langle s_2^A, \sigma_1^{'A}\rangle \cdot \gamma).$$

(i) We know that $s_{1_m}^A \in E_A$ and $s_{1_m}^A \models m$. Also since $s_{1_m}^A$ is an entry point, there is an entry point $s_{1_m}^B \in E_B$ such that $(s_{1_m}^A, s_{1_m}^B) \in R_s$.
(ii) We know $s_1^A \xrightarrow{m(a_1,\ldots,a_m)} s_2^A$ and that there is s_1^B such that $(s_1^A, s_1^B) \in R_s$. It follows that $s_1^B \xrightarrow{m(a_1,\ldots,a_m)} s_2^B$ such that $\lambda_P(s_1^A) \sqsubseteq \lambda_P(s_1^B)$ and $\lambda_A(s_1^A) = \lambda_A(s_1^B)$.
From (i) and (ii) we conclude that there is a transition

$$(\langle s_1^B, \sigma_1^B, \sigma_1^{'B}\rangle, \gamma) \xrightarrow{m' \texttt{ call } m(\sigma_1^{'B}(a_1),\ldots,\sigma_1^{'B}(a_n))} (\langle s_{1m}^B, \sigma_{1m}^B, \sigma_{1m}^{'B}\rangle, \langle s_2^B, \sigma_1^{'B}\rangle \cdot \gamma)$$

such that

$$((\langle s_{1m}^A, \sigma_{1m}^A, \sigma_{1m}^{'A}\rangle, \langle s_2^A, \sigma_1^{'A}\rangle \cdot \gamma), (\langle s_{1m}^B, \sigma_{1m}^B, \sigma_{1m}^{'B}\rangle, \langle s_2^B, \sigma_1^{'B}\rangle \cdot \gamma)) \in R_b$$

Case 3. (Ret) Suppose $(\langle s^A, \sigma^A, \sigma^{'A}\rangle, \gamma^A) \xrightarrow{m' \texttt{ret}(x) m} (\langle s_2^A, \sigma_2^A, \sigma_2^{'A}\rangle, \gamma^{'A})$. We know $s^A \models r \wedge m'$ and that there is s^B such that $(s^A, s^B) \in R_s$. Hence $s^B \models r \wedge m'$, $\lambda_P(s^A) \sqsubseteq \lambda_P(s^B)$, and $\lambda_A(s^A) = \lambda_A(s^B)$. We also know that $\gamma^A = (\langle s_2^A, \sigma^{''A}\rangle, \gamma^{'A})$ for some $\sigma^{''A}$. Since $((\langle s^A, \sigma^A, \sigma'^A\rangle, \gamma^A), (\langle s^B, \sigma^B, \sigma^{'B}\rangle, \gamma^B)) \in R_b$, there is a $\sigma^{''B} = \sigma^{''A}$ such that, $\gamma^B = (\langle s_2^B, \sigma^{''B}\rangle, \gamma^{'B})$, $(s_2^A, s_2^B) \in R_s$, and $\sigma^{''A} = \sigma^{''B}$. Thus, there exists a transition

$$(\langle s^B, \sigma^B, \sigma^{'B}\rangle, \gamma^B) \xrightarrow{m'' \texttt{ ret}(x) m} (\langle s_2^B, \sigma_2^B, \sigma_2^{'B}\rangle, \gamma^{'B})$$

such that

$$((\langle s_2^A, \sigma_2^A, \sigma_2^{'A}\rangle, \gamma^{'A}), (\langle s_2^B, \sigma_2^B, \sigma_2^{'B}\rangle, \gamma^{'B})) \in R_b$$

Thus we conclude that $A \leqslant_b B$. □

4 Logic

As a property specification language we use the safety fragment of Modal Equation Systems [27], that is without diamond modalities. This logic is equal in expressive power to the safety fragment of the modal μ-calculus [25]. Here, we employ the former logic for technical reasons that will become clear later, but a user is free to use either. The translation of μ-calculus to simulation logic defined in Definition 7 below is based on Bekič's principle described in [13,8].

The translation in the other direction is straightforward and done simply by replacing each fixed point by an equation.

Following Larsen [27], we define the syntax and semantics of the specification language in two steps: first we define a basic modal logic that is parametrized on a set of labels L, state assertions P, and atoms A, and then we add recursion by means of equation systems in Definition 7. *Basic simulation logic* is a variant of Hennessy-Milner logic [20] without diamond modalities.

Definition 5 (Basic Simulation Logic: Syntax). *The formulas of* basic simulation logic *are inductively defined by:*

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

where $a \in A$, $p \in P$, $l \in L$, and X ranges over a set of propositional variables $\mathcal{V}$. Formulas of the shape a, $\neg a$, p, and $\neg p$ are called atomic formulas.

Definition 6 (Basic Simulation Logic: Semantics). *The* semantics *of a formula ϕ of basic simulation logic over L, P, and A w.r.t. model M and an environment $\rho : \mathcal{V} \to 2^S$ is defined inductively by*

$$\begin{array}{ll}
\|a\|\rho & \stackrel{def}{=} \{s \in S \mid \lambda_A(s) = a\} \\
\|\neg a\|\rho & \stackrel{def}{=} S \setminus \|a\|\rho \\
\|p\|\rho & \stackrel{def}{=} \{s \in S \mid \lambda_P(s) \sqsubseteq p\} \\
\|\neg p\|\rho & \stackrel{def}{=} S \setminus \|p\|\rho \\
\|X\|\rho & \stackrel{def}{=} \rho(X) \\
\|\phi_1 \wedge \phi_2\|\rho & \stackrel{def}{=} \|\phi_1\|\rho \cap \|\phi_2\|\rho \\
\|\phi_1 \vee \phi_2\|\rho & \stackrel{def}{=} \|\phi_1\|\rho \cup \|\phi_2\|\rho \\
\|[l]\phi\|\rho & \stackrel{def}{=} \{s \in S \mid \forall t \in S. s \xrightarrow{l} t \text{ implies } t \in \|\phi\|\rho\}
\end{array}$$

where $\sqsubseteq$ is the preorder defined on the set of state assertions.

Definition 7 (Modal Equation System). *A* modal equation system $\Pi = \{X_i = \Phi_i \mid i \in J\}$ *over L and A for a set of indexes J is a finite set of* defining equations *such that the variables X_i are pairwise distinct and each Φ_i is a formula of basic simulation logic over L, P, and A. The set of variables occurring in Π is partitioned into the set of* bound *variables, defined by* $\mathtt{bv}(\Pi) = \{X_i \mid i \in J\}$*, and the set of* free *variables* $\mathtt{fv}(\Pi)$.

The semantics of a closed modal equation system $\|[\Pi]\|\rho$ is defined as its greatest fixed point. We use n-ary versions of conjunction and disjunction, setting $\bigwedge \emptyset = \mathtt{tt}$ (*true*) and $\bigvee \emptyset = \mathtt{ff}$ (*false*). We use *Labels*(X) and *Atoms*(X) to refer to the set of labels and atoms of the defining equation for X, respectively.

The semantics of a modal equation systems is defined in terms of its greatest solution. A solution of a modal equation system Π is a map $\eta : \mathtt{bv}(\Pi) \to 2^S$, assigning to each variable $X \in \mathtt{bv}(\Pi)$ a set of states, such that all equations in Π are satisfied. Maps η are ordered by point-wise inclusion. We first define the environment update $\rho[\eta]$, as $\rho[\eta](X) = \eta(X)$ if $X \in \mathtt{bv}(\Pi)$ and $\rho[\eta](X) = \rho(X)$ otherwise. Then we define the map $\Psi_{\Pi,\rho} \colon 2^S_{\mathtt{bv}(\Pi)} \to 2^S_{\mathtt{bv}(\Pi)}$ induced by the equations in Π by $\Psi_{\Pi,\rho}(\eta)(X) = \|\phi_X\|\rho[\eta]$.

Definition 8 (Solutions). *A* solution *of a modal equation system Π with respect to a model $\mathcal{M}$ and an environment ρ is a map $\eta : \mathtt{bv}(\Pi) \to 2^S$ such that $\Psi_{\Pi,\rho}(\eta) = \eta$. The* semantics *of a modal equation system Π with respect to $\mathcal{M}$ and ρ, denoted $\|\Pi\|\rho$, is its greatest solution.*

Note that by the well-known Knaster-Tarski fixed point theorem [34] the greatest solution of $\Psi_{\Pi,\rho}$ always exists, since $\Psi_{\Pi,\rho}$ is a monotone map on the lattice $2^S_{\mathtt{bv}(\Pi)}$ ordered by point-wise inclusion.

Definition 9 (Simulation Logic). *The formulas of* simulation logic *over L and A are defined by $\phi[\Pi]$, where ϕ is a formula of basic simulation logic and Π is a modal equation system. The set of free and bound variables are $\mathtt{fv}(\phi[\Pi]) = (\mathtt{fv}(\phi) \cup \mathtt{fv}(\Pi)) - \mathtt{bv}(\Pi)$ and $\mathtt{bv}(\phi[\Pi]) = \mathtt{bv}(\Pi)$, respectively.*

The semantics *of $\phi[\Pi]$ with respect to model $\mathcal{M}$ and environment ρ is defined by*

$$\|\phi[\Pi]\|\rho = \|\phi\|\rho[\|\Pi\|\rho].$$

We say that a state s of a model $\mathcal{M}$ satisfies $\phi[\Pi]$, written $(\mathcal{M}, s) \models \phi[\Pi]$, if $s \in \|\phi[\Pi]\|\rho$ for all ρ. For specifications $(\mathcal{M}, E)$ we define $(\mathcal{M}, E) \models \phi[\Pi]$ if $(\mathcal{M}, s) \models \phi[\Pi]$ for all $s \in E$.

Simulation logic is capable of expressing safety properties of sequences of observed actions, calls and returns. We use two instantiations of this logic to represent structural and behavioral properties. Structural logic expresses properties of flow graphs (Definition 3), therefore it is instantiated by $a \in A$, $p \in P$, and $l \in L$. Behavioral logic, however, expresses properties of flow graph behaviors (Definition 4), therefore it is instantiated by $a \in A_b$, $p \in P_b$, and $l \in L_b$.

Example 1. The structural local property "`Container` can only be called as the last statement of the method `Servlet`" in Figure 1 is specified by the structural formula $X[\Pi]$, where Π is

$$X = [\mathtt{Container()}]\mathtt{r} \wedge \bigwedge_{l \in L_{\mathtt{Servlet}} \setminus \mathtt{Container()}} [l]X$$

The second behavioral global property in Figure 1 is specified by the behavioral formula $X[\Pi]$, where Π is $X = ((\mathtt{Main} \wedge \mathtt{r}) \Rightarrow (\mathtt{p} = \mathtt{null} \wedge \mathtt{c} = \mathtt{null})) \wedge \bigwedge_{l \in L_b} [l]X$.

5 Maximal Models and Flow Graphs

To construct maximal models, we generalize our previous algorithm for models without program data [19], following closely the treatment there. We therefore only sketch our construction here, and refer the reader to [19] for the details. Our construction algorithm is defined on the general notion of model (Definition 1).

5.1 Maximal Model Construction

We define two auxiliary functions θ and χ which form a *Galois connection* between finite models and formulas in simulation logic. Function χ translates a *finite* model into a formula, while θ translates a formula into a (finite) model. Both functions are defined on formulas in a so-called *simulation normal form* (SNF). In this section, we define SNF and show that every formula of simulation logic has an SNF representation and provide an algorithm to convert a formula to its SNF. The construction of maximal models basically consists of translating a given formula into SNF and applying function θ on the result.

Definition 10 (χ). *Function χ maps a finite initialized model $(\mathcal{M}, E)$ into its* characteristic formula $\chi(\mathcal{M}, E) = \phi_E[\Pi_{\mathcal{M}}]$, *where $\phi_E = \bigvee_{s\in E} X_s$, and $\Pi_{\mathcal{M}}$ is defined by the equations:*

$$X_s = \bigwedge_{l\in L}[l]\bigvee_{s\xrightarrow{l}t} X_t \wedge \bigwedge_{a\in\lambda_A(s)} a \wedge \bigwedge_{b\notin\lambda_A(s)} \neg b \wedge \lambda_P(s)$$

The next result shows that function χ precisely translates an initialized model to a formula. This is a variation of an earlier result by Larsen [27].

Theorem 3. *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two initialized models and let $\mathcal{S}_2$ be finite. Then $\mathcal{S}_1 \leqslant \mathcal{S}_2$ if and only if $\mathcal{S}_1 \models \mathcal{X}(\mathcal{S}_2)$.*

Proof. (adapted from [27]; included here for completeness) Let $\mathcal{S}_1 = (\mathcal{M}_1, E_1)$ and $\mathcal{S}_2 = (\mathcal{M}_2, E_2)$.

"$\Rightarrow$" Let Ψ be the map on $2^S_{\mathtt{bv}(\Pi)}$ induced by the equations in Π (Ψ_Π before Definition 8). In order to prove that $(\mathcal{M}_1, E_1) \models (\bigvee_{s\in E_2} X_s)[\Pi_{\mathcal{M}_2}]$ it is sufficient to show that the map η defined by $\eta(X_s) = \{t \in S_1 \mid t \leqslant s\}$ is a post-fixed point of Ψ. It then follows by fixed point induction that $\eta \subseteq \|\Pi_{\mathcal{M}_2}\|$. Also, since $\mathcal{S}_1 \leqslant \mathcal{S}_2$, we have that for each $t \in E_1$ there is some $s \in E_2$ such that $t \in \eta(X_s)$. Hence $t \in \|\Pi\|(X_s)$ and therefore $t \models \chi(\mathcal{S}_2)$.

It remains to be shown that $\eta(X_s) \subseteq \Psi(\eta)(X_s)$ for all $s \in S$. Let $t \in \eta(X_s)$, hence $t \leqslant s$. We have to establish $t \in \Psi(\eta)(X_s)$, that is,

1. $t \in \|[a]\bigvee_{s\xrightarrow{a}s'} X_{s'}\|\rho[\eta]$ for all $a \in L$,
2. $t \in \|\bigwedge_{a\in\lambda_A(s)} a \wedge \bigwedge_{b\notin\lambda_A(s)} \neg b\|\rho[\eta]$, and
3. $t \in \|\lambda_P(s)\|\rho[\eta]$.

For (1) suppose $t\xrightarrow{a}t'$. Since $t \leqslant s$, there is a s' such that $s\xrightarrow{a}s'$ and $t' \leqslant s'$. Hence, $t' \in \eta(X_{s'})$. Points (2) and (3) follow from $t \leqslant s$ and the definition of simulation.

"$\Leftarrow$" Let $\chi(\mathcal{S}_2) = (\bigvee \mathcal{X})[\Pi]$ with $\mathcal{X} = \{X_s \mid s \in E_2\}$ and let $\eta = \|\Pi\|\rho$ for some (arbitrary) environment ρ. We show that $R = \{(s,t) \mid s \in \eta(X_t)\}$ is a simulation between $\mathcal{M}_1$ and $\mathcal{M}_2$. The result $\mathcal{S}_1 \leqslant \mathcal{S}_2$ then easily follows. Let $(s,t) \in R$, that is, $s \in \eta(X_t)$. Then s and t satisfy the same propositions, since $s \in \|\bigwedge_{a\in\lambda_A(t)} a \wedge \bigwedge_{b\notin\lambda_A(t)} \neg b\|\rho$, and $s \in \|\lambda_P(t)\|\rho$. Suppose now that $s\xrightarrow{a}s'$. Since $s \in \|[a]\bigvee_{t\xrightarrow{a}t'} X_{t'}\|\rho[\eta]$, we have $s' \in \eta(X_{t'})$ for some t' with $t\xrightarrow{a}t'$. This shows that R is a simulation between $\mathcal{M}_1$ and $\mathcal{M}_2$. □

Definition 11 (Simulation Normal Form). *A formula $\phi[\Pi]$ of simulation logic over L, A, and P is in* simulation normal form (SNF) *if ϕ has the form $\bigvee \mathcal{Z}$ for some finite set $\mathcal{Z} \subseteq \mathtt{bv}(\Pi)$ and all equations of Π have the following* state normal form

$$X = \bigwedge_{l \in L} [l] \bigvee \mathcal{Y}_{X,l} \wedge \bigwedge_{a \in B_X} a \wedge \bigwedge_{b \notin A \setminus B_X} \neg b \wedge p \tag{1}$$

where each $\mathcal{Y}_{X,l} \subseteq \mathtt{bv}(\Pi)$ is a finite set of variables, $B_X \subseteq A$ is a set of atomic propositions, and $p \in P$ is a state assertion.

To translate simulation logic formulas into SNF we generalize the algorithm of [19] that works as follows. For a given set of atoms A, labels L, and a formula $\phi[\Pi]$, it saturates each equation of Π by conjoining its missing labels as $\bigwedge_{l \in L \wedge l \notin Labels(X)} [l]\ \mathtt{tt}$, and atoms as $\bigwedge_{a \in A \wedge a \notin Atoms(X)} (a \vee \neg a)$, and then transforms the resulting formula to SNF by introducing new equations for disjunctions of formulas not guarded by any box. Our adaptation of this algorithm to formulas $\phi[\Pi]$ of Definition 7 proceeds in two steps. First, we apply the above algorithm to $\phi[\Pi]$, simply carrying over the assertions of the equations. In the second step, we conjoin the top element of the lattice of P to the resulting equations that do not have any assertions. In this way we simplify the saturation of assertions, that would otherwise be very inefficient or even impossible when the set of variables and their values is large or infinite.

Example 2. The table in Figure 3 illustrates the two steps of converting a formula of simulation logic to its SNF. The formula is of form $X1[\Pi]$ where Π is given in row 1. We let $L = \{l_1, l_2\}$, $A = \{a\}$, and $P = \{p, \top, \bot\}$ where $\top$ represents the top of the lattice ordered by $\sqsubseteq$. We show how the set of equations of this formula is changed at each step. The formula, in row 2 is translated to SNF with the data-less algorithm by carrying over the state assertions; in row $2a$ equations are saturated by conjoining the missing atomic propositions and labels, and in row $2b$ new equations are introduced for disjunctions not guarded by any box. Observe that $\mathtt{tt}$ in equations $X1$ and $X2$ in row $2a$ gives rise to two equations $X4$ and $X5$ in row $2b$. Finally in row 3, to the equations without any state assertions, $\top$ is added.

Theorem 4. *Every formula of simulation logic has an equivalent one in SNF.*

Proof. The correctness and termination of the algorithm for translating data-free simulation logic formulas to their SNF form are shown in [19]. We extend this algorithm by adding an additional step to the translation. However, this change does not effect the argument of the proof. □

Definition 12 (θ). *Function θ translates a formula $(\bigvee \mathcal{X})[\Pi]$ over L, A, and P, that is in SNF as in (1), to the (finite) initialized model $\theta((\bigvee \mathcal{X})[\Pi]) = ((S, L, \rightarrow, A, P, \lambda_A, \lambda_P), E)$ where $S = \mathtt{bv}(\Pi)$, $E = \mathcal{X}$ and for every $X \in \mathcal{X}$ the equation for X induces transitions $\{X \xrightarrow{l} Y \mid Y \in \mathcal{Y}_{X,l}\}$, $\lambda_A(X) = B_X$, and $\lambda_P(X) = p$.*

1)	$X1 = [l_1]X2 \wedge a \wedge p$, $X2 = [l_2]\ \mathtt{ff}$
2a)	$X1 = [l_1]X2 \wedge [l_2]\ \mathtt{tt} \wedge a \wedge p$, $X2 = [l_1]\ \mathtt{tt} \wedge [l_2]\ \mathtt{ff} \wedge (a \vee \neg a)$
2b)	$X1 = [l_1](X2 \vee X3) \wedge [l_2](X4 \vee X5) \wedge a \wedge p$, $X2 = [l_1](X4 \vee X5) \wedge [l_2]\ \mathtt{ff} \wedge a$, $X3 = [l_1](X4 \vee X5) \wedge [l_2]\ \mathtt{ff} \wedge \neg a$, $X4 = [l_1](X4 \vee X5) \wedge [l_2](X4 \vee X5) \wedge a$, $X5 = [l_1](X4 \vee X5) \wedge [l_2](X4 \vee X5) \wedge \neg a$
3)	$X1 = [l_1](X2 \vee X3) \wedge [l_2](X4 \vee X5) \wedge a \wedge p$, $X2 = [l_1](X4 \vee X5) \wedge [l_2]\ \mathtt{ff} \wedge a \wedge \top$, $X3 = [l_1](X4 \vee X5) \wedge [l_2]\ \mathtt{ff} \wedge \neg a \wedge \top$, $X4 = [l_1](X4 \vee X5) \wedge [l_2](X4 \vee X5) \wedge a \wedge \top$, $X5 = [l_1](X4 \vee X5) \wedge [l_2](X4 \vee X5) \wedge \neg a \wedge \top$

Fig. 3: Transformation of a formula to SNF

Lemma 1. *χ and θ are each others inverse up to equivalence, that is,*

1. *$\theta(\chi(\mathcal{S})) \cong \mathcal{S}$ ($\cong$ is isomorphism[2]) for finite $\mathcal{S}$, and*
2. *$\chi(\theta(\phi)) \equiv_\alpha \phi$ ($\equiv_\alpha$ is α-convertibility) for ϕ in SNF.*

Finally we are ready to relate simulation logic to simulation.

Theorem 5 (Maximal Model Theorem). *For any ϕ in SNF, we have $\mathcal{S} \leqslant \theta(\phi)$ if and only if $\mathcal{S} \models \phi$.*

Proof. Follows from Theorem 3 by Lemma 1(2). □

Thus, the model $\theta(\phi)$ is a maximal model for ϕ, in the sense that $\theta(\phi)$ is a model that satisfies ϕ and simulates all models satisfying it.

Consequences. We mention a few consequences of Theorems 3 and 5. Let $(\mathsf{S}, \leqslant)$ be the preorder of (isomorphism classes of) *finite* models over given L and A ordered by simulation and let $(\mathsf{L}, \models)$ be the preorder of formulas of simulation logic over L and A ordered by the logical consequence relation.

Corollary 1. *χ and θ are monotone.*

Simulation preserves logical properties:

Corollary 2. *For all initialized models $\mathcal{S}_1$ and $\mathcal{S}_2$ we have $\mathcal{S}_1 \leqslant \mathcal{S}_2$ and $\mathcal{S}_2 \models \phi$ imply $\mathcal{S}_1 \models \phi$.*

The pair (χ, θ) of maps forms a Galois connection between the preorders $(\mathsf{L}, \models)$ and $(\mathsf{S}, \leqslant)$.

Corollary 3. *For any finite initialized model $\mathcal{S}$ and all formulas ϕ, we have $\mathcal{S} \leqslant \theta(\phi)$ if and only if $\chi(\mathcal{S}) \models \phi$.*

[2] Here, isomorphism means a bijection of states and transitions, but labels have to be equal.

5.2 Maximal Flow Graph Construction

Maximal models constructed from structural properties by the above algorithm are in general not legal flow graphs. To restrict these to legal flow graphs, we conjoin the property with a so-called *characteristic formula* C_I constructed from the interface $I = (M^+, M^-, \mathit{Modify})$. C_I describes precisely the models that constitute flow graphs with interface I:

$$\begin{array}{l} C_I = \Phi_I[\Pi_I]\,,\ \Phi_I = \bigvee_{m\in M^+} X_m \\ \Pi_I = \{X_m = \bigwedge_{l\in L}[l]X_m \wedge a_m \wedge p_m \mid m \in M^+\} \\ a_m = m \wedge \bigwedge\{\neg m' \mid m' \in M^+, m' \neq m\} \\ p_m = \bigwedge\{v = v' \mid v \notin \mathit{Modify} \wedge v \in V\} \end{array}$$

With the help of C_I we obtain a variant of Theorem 5 for flow graphs.

Theorem 6. *Let $I = (M^+, M^-, \mathit{Modify})$ be an interface. For any initialized model $\mathcal{S} = (\mathcal{M}, E)$ over L and $A = M^+ \cup \{r\}$ we have:*

1. *$\mathcal{S} \models C_I$ if and only if $\mathcal{R}(\mathcal{S}) : I$*
2. *$\mathcal{S} \leqslant_s \theta(\phi \wedge C_I)$ if and only if $\mathcal{S} \models \phi$ and $\mathcal{R}(\mathcal{S}) : I$*

where $\mathcal{R}(\mathcal{S})$ denotes the reachable part of $\mathcal{S}$.

Proof. (1): "$\Rightarrow$" Suppose $(\mathcal{M}, E) \models C_I$. We use induction on the size of M^+.

Case $M^+ = \emptyset$. In this case $C_I \equiv \mathtt{ff}$, so the only model which satisfies this property is the empty flow graph $\emptyset_M$.

Case $M^+ = \{m\}$. Here $C_I = X_m[X_m = \bigwedge_{l\in L}[l]X_m \wedge m \wedge p_m]$. Any model satisfying this property is a single method graph and all states satisfy p_m. Thus $(\mathcal{M}, E)$ is an flow graph with interface $(\{m\}, M^-, \mathit{Modify})$.

Case $M^+ = M_1 \cup M_2$ for disjoint and non-empty M_1 and M_2. Since $(\mathcal{M}, E) \models C_I$, we know that every state in the model satisfies exactly one of the atomic predicates $m \in M^+$. We can define $(\mathcal{M}_1, E_1)$ and $(\mathcal{M}_2, E_2)$ as the restrictions of $(\mathcal{M}, E)$ *w.r.t.* $I_1 = (M_1, M^-, \mathit{Modify})$ and $I_2 = (M_2, M^-, \mathit{Modify})$. Notice that $(\mathcal{M}_1, E_1) \uplus (\mathcal{M}_2, E_2) = (\mathcal{M}, E)$. We can decompose $C_I = \Phi_{(I_1)} \wedge \Phi_{(I_2)}[\Pi_{(I_1)}, \Pi_{(I_2)}]$. By induction, $(\mathcal{M}_1, E_1) : (I_1)$ and $(\mathcal{M}_2, E_2) : (I_2)$, thus by the definition of flow graph $(\mathcal{M}, E) : I$.

The restriction to the reachable part of $\mathcal{S}$ is required, because the formula C_I does not constrain the unreachable part of $\mathcal{S}$.

"$\Leftarrow$" By Theorem 5, it is sufficient to show $(\mathcal{M}, E) \leqslant_s \theta(C_I)$. First, we calculate $\theta(C_I)$, which gives for each method name $m \in M^+$ the method graph $(S_m, L_m, \rightarrow_m, \{m, r\}, P, \lambda_{A_m}, \lambda_{P_m})$, where

$$\begin{array}{l} S_m \ = \{X_{m,r}, X_{m,\neg r}\} \\ \rightarrow_m = S_m \times L \times S_m \\ \lambda_{A_m} = \{(X_{m,r}, \{m, r\}), (X_{m,\neg r}, \{m\})\} \\ \lambda_{P_m} = \{(X_{m,r}, \{v = v' \mid v \notin \mathit{Modify} \wedge v \in \ \mathtt{Glob}\}), \\ \qquad\qquad (X_{m,\neg r}, \{v = v' \mid v \notin \mathit{Modify} \wedge v \in \ \mathtt{Glob}\})\} \end{array}$$

Using the relation $R = \{(s, t) \mid \lambda_A(s) = \lambda_A(t) \wedge \lambda_P(s) \sqsubseteq \lambda_P(t)\}$, it is easy to show that any model $(\mathcal{M}, E)$ with interface I is simulated by this flow graph.

(2): By (1), for flow graph $\mathcal{G} : I$, $\mathcal{G} \models \phi[\Pi]$ if and only if $\mathcal{G} \models \phi \wedge \Phi_I[\Pi, \Pi_I]$. The result then follows from Theorem 5. □

A maximal model that is constructed for the conjunction of a formula ϕ and characteristic formula for interface I, is a maximal flow graph for I and ϕ.

6 Compositional Verification

As mentioned in Section 5, for models and formulas as defined in Definition 1 and Definition 7, maximal models exist and are unique up to isomorphism. Therefore, for this choice of model and logic we can provide the following principle for compositional verification that is sound and complete for finite models:

> "To show $\mathcal{M}_1 \uplus \mathcal{M}_2 \models \psi$, it suffices to show $\mathcal{M}_1 \models \phi$, i.e., that component $\mathcal{M}_1$ satisfies a suitably chosen *local specification* ϕ, and $\theta(\phi) \uplus \mathcal{M}_2 \models \psi$, i.e., that $\mathcal{M}_2$, when composed with the maximal model $\theta(\phi)$, satisfies the *global property* ψ."

We exploit Theorem 6 to adapt this principle to flow graphs (as models) and structural logic and use the maximal flow graph construction from Section 5.2 to obtain the rule below.

$$\frac{\mathcal{G}_1 \models \phi \qquad \theta(\phi \wedge C_I) \uplus \mathcal{G}_2 \models \psi}{\mathcal{G}_1 \uplus \mathcal{G}_2 \models \psi} \tag{2}$$

The rule states that the composition of flow graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ satisfies the structural property ψ if flow graph $\mathcal{G}_1$ satisfies a local structural property ϕ, and the composition of flow graph $\mathcal{G}_2$ with the maximal flow graph for ϕ and interface I satisfies ψ.

Theorem 7. *Rule (2) is sound and complete.*

Proof. "⇒" Suppose $\mathcal{G}_1 \models \phi$ and $\theta(\phi \wedge C_I) \uplus \mathcal{G}_2 \models \psi$. By Theorem 6, and the first assumption we have $\mathcal{G}_1 \leqslant \theta(\phi \wedge C_I)$. It follows that $\mathcal{G}_1 \uplus \mathcal{G}_2 \leqslant \theta(\phi \wedge C_I) \uplus \mathcal{G}_2$ by Theorems 1 and 2. Hence, $\mathcal{G}_1 \uplus \mathcal{G}_2 \models \psi$ by Corollary 2 (instantiated to the behavioral level) and the second assumption.

"⇐" Suppose $\mathcal{G}_1 \uplus \mathcal{G}_2 \models \psi$ and set $\phi = \chi(\mathcal{G}_1)$. We have to show that $\mathcal{G}_1 \models \chi(\mathcal{G}_1)$ and $\theta(\chi(\mathcal{G}_1) \wedge C_I) \uplus \mathcal{G}_2 \models \psi$. The former follows from Theorem 3 (for $\mathcal{S}_1 = \mathcal{S}_2$, instantiated to structural level). To see the latter, we start by the observation that $\chi(\mathcal{G}_1) \wedge C_I \models \chi(\mathcal{G}_1)$. By the monotonicity of θ (Corollary 1), we get $\theta(\chi(\mathcal{G}_1) \wedge C_I) \leqslant \theta(\chi(\mathcal{G}_1))$. Lemma 1 states that $\theta(\chi(\mathcal{G}_1)) \cong \mathcal{G}_1$. Hence, using the definition of structural simulation, $\theta(\chi(\mathcal{G}_1) \wedge C_I) \leqslant_s \mathcal{G}_1$. It follows by Theorems 1 and 2 that $\theta_I(\chi(\mathcal{G}_1)) \uplus \mathcal{G}_2 \leqslant_b \mathcal{G}_1 \uplus \mathcal{G}_2$. Finally, Corollary 2 and the assumption imply that $\theta(\chi(\mathcal{G}_1) \wedge C_I) \uplus \mathcal{G}_2 \models \psi$. □

We restrict local specifications to structural properties, and by exploiting the fact that structural simulation implies behavioral simulation (Theorem 2), we obtain a complete compositional verification rule for global behavioral properties,

thus avoiding the possibility of false negatives. However, adapting the compositional verification principle to local behavioral specifications is more problematic, as behavioral properties in general do not give rise to unique maximal flow graphs. We can represent the set of flow graphs satisfying the local specification by a (pushdown) model that simulates the behavior of these flow graphs, but this necessarily leads to approximate (i.e., sound but incomplete) solutions, since such a model cannot be guaranteed to be a legal flow graph behavior.

7 Instantiation of the Framework with Full Data Abstraction

In this section, we present an instantiation of our generic framework when program data is abstracted away completely. We show that the resulting framework is isomorphic to our previous compositional verification framework. Thus, our generic framework is a proper generalization of the old one.

Throughout this section, we use the Java program shown in Figure 4 as a running example. This program was also used as a means of illustration in our previous work [32]. Readers can compare models and properties shown in the subsequent subsections with those of [32]. In this example we consider method `even` as a variable component. A global property and the structural local specification for method `even` are given in Figure 4. In the following subsections, we instantiate our generic framework to a concrete set of state assertions, actions and their semantics. The result will be a compositional verification technique for procedural programs that abstracts away all data.

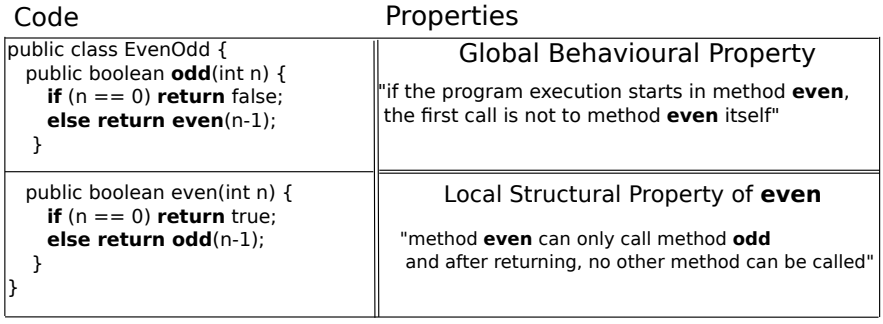


Fig. 4: A simple Java program

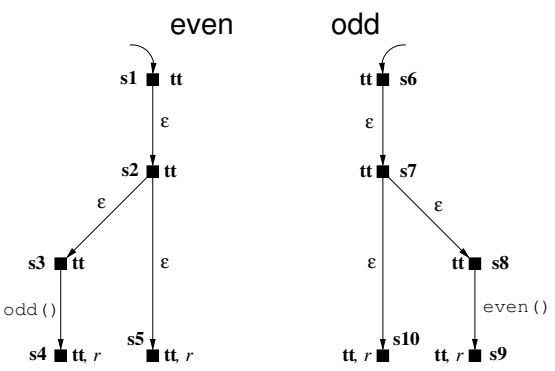
7.1 Program Models

Our flow graphs are parametrized on a set of state assertions and actions. The definition of flow graphs with full data abstraction is an instantiation of the generic definition of flow graphs with the assertions and actions defined below.

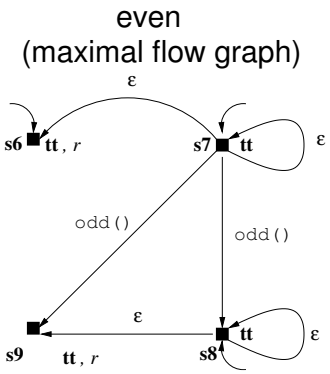
State Assertions. When the data is fully abstracted away, the set of assertions P should be empty. However, to be faithful to Definition 1, we use the (dummy) assertion `tt` and assign it to all states of data-less flow graphs. Then, the semantic entailment $\sqsubseteq$ on state assertions is defined by *equality*.

Actions. Since we are interested in control-flow properties, we do not identify any particular program instructions as actions. However, we include the neutral action ε for simplicity, and thus the set of actions is the singleton set $\{\varepsilon\}$.

Example 3. Figure 5a shows the flow graph of the program in Figure 4. Its interface is $M^+ = \{\texttt{even}, \texttt{odd}\}, M^- = \{\texttt{even}, \texttt{odd}\}, \mathit{Modify} = \{\}$. It consists of two method graphs, for methods `even` and `odd`. The interface of method `even` is $M^+ = \{\texttt{even}\}, M^- = \{\texttt{odd}\}, \mathit{Modify} = \{\}$.



(a) Flow graph of EvenOdd



(b) Maximal flow graph for method `even`

In order to instantiate the flow graph behavior we first need to define program states.

Program States. For the instantiation of our framework with full program abstraction, the sets of variables V and their values $\mathcal{D}$ are empty. However, to be faithful to our definition of program state we assume a dummy variable (v) and a single value (true) that form a single program state σ mapping v to true.

Flow Graph Behavior. The flow graph behavior with full data abstraction is simply defined as an instantiation of the generic definition of behavior (Definition 4) where the semantics of labels are *identity* function, i.e.,

$$[\![\varepsilon]\!]\sigma_1' = \sigma_1'$$

$$[\![call]\!]\sigma_1' = \sigma_1'$$

$$[\![ret]\!](\sigma_1', \sigma_2) = \sigma_2$$

Example 4. Consider the flow graph in Figure 5a. One example run through its (branching, infinite-state) behavior, for $\sigma = v \to true$, from an initial to a final configuration, is:

$$(\langle s_1, \sigma, \sigma\rangle, \epsilon) \xrightarrow{\varepsilon} (\langle s_2, \sigma, \sigma\rangle, \epsilon) \xrightarrow{\varepsilon} (\langle s_3, \sigma, \sigma\rangle, \epsilon) \xrightarrow{\texttt{even call odd()}} (\langle s_6, \sigma, \sigma\rangle, \langle s_4, \sigma, \sigma\rangle) \xrightarrow{\varepsilon}$$
$$(\langle s_7, \sigma, \sigma\rangle, \langle s_4, \sigma, \sigma\rangle) \xrightarrow{\varepsilon} (\langle s_{10}, \sigma, \sigma\rangle, \langle s_4, \sigma, \sigma\rangle) \xrightarrow{\texttt{odd ret() even}} (\langle s_4, \sigma, \sigma\rangle, \epsilon)$$

7.2 Logics

The structural and behavioral logics are also defined as instantiations of generic simulation logic (Definition 9) for the sets of assertions and actions defined above.

Example 5. Consider the local structural specification of method `even` mentioned informally in Figure 5a. This specification is formalized by the following structural formula $(\texttt{even} \Rightarrow X)[\Pi]$, where Π is:

$$X = [\texttt{even}()]\texttt{ff} \wedge [\varepsilon]X \wedge [\texttt{odd}()]Y$$
$$Y = [\texttt{even}()]\texttt{ff} \wedge [\texttt{odd}()]\texttt{ff} \wedge [\varepsilon]Y$$

The global behavioral property shown in the figure is formalized by the following behavioral formula $(\texttt{even} \Rightarrow X)[\Pi]$ where Π is:

$$X = [\texttt{even call even}()]\texttt{ff} \wedge [\varepsilon]X$$

7.3 Maximal Flow Graphs

Maximal flow graphs are constructed by applying the algorithm described in Section 5 for the set of structural labels (method names and actions), atomic propositions (method names and `r`), and the top element of the lattice of state assertions ordered by $\sqsubseteq$ (which is here `tt`).

Example 6. To construct maximal flow graph for method `even`, the following characteristic formula is constructed from its interface.

$(X0 \vee X1)[\Pi]$ where Π is:

$$X0 = ([\varepsilon](X0 \vee X1) \wedge [\texttt{even()}](X0 \vee X1) \wedge [\texttt{odd()}](X0 \vee X1) \wedge \texttt{r} \wedge \texttt{even} \wedge \texttt{tt})$$
$$X1 = ([\varepsilon](X0 \vee X1) \wedge [\texttt{even()}](X0 \vee X1) \wedge [\texttt{odd()}](X0 \vee X1) \wedge \neg\texttt{r} \wedge \texttt{even} \wedge \texttt{tt})$$

The above formula is conjoined with the local structural specification of method `even` presented above and the maximal flow graph shown in Figure 5b is constructed.

7.4 Compositional Verification

We use Rule 2 for the program models, logics and maximal flow graphs defined in sections 7.1, 7.2, and 7.3. As a result, we obtain a compositional verification technique for procedural programs where data is abstracted away.

Example 7. We use Rule 2 to verify the program in Figure 4, considering method `even` as a variable component. This divides the verification effort into the following two independent sub-tasks. (i) We check that the structural local specification of method `even` is satisfied by its implementation. (The implementation of `even` in the figure satisfies its local structural specification.) (ii) A maximal flow graph of method `even` is constructed and composed with the flow graph of method `odd`. Then the behavior induced from the resulting flow graph is model checked against the global properties of the program.

8 Instantiation of the Framework for Boolean Programs

In this section, we present an instantiation of our generic framework for Boolean programs [10]. These are procedural programs with Boolean variables as the only datatype. This language has been studied and used as abstract representation of real-life programming languages in several works [12,9,11,29,15]. Here, we first define formally the language and then instantiate the definitions of flow graph, behavior, logic, and the maximal flow graph construction algorithm to Boolean programs.

8.1 Syntax

The syntax of Boolean programs is shown in Table 1. The language has Boolean variables as the only datatype. It supports conditional statements, loops, and method calls. A method has a return value if an expression or a variable name is provided to the return statement, otherwise the method returns `false` by reaching a return statement or the end of its execution.

As shown in Table 1, the variables of Boolean programs are either global (if they are declared outside the scope of a method), local to a method (if they are declared inside the scope of a method), or parameters.

Throughout this section, we use the Boolean program shown in Figure 5 as a running example. The program resembles an electronic referendum voting system that gets a vote from a voter and registers it into an electronic ballot box, such as a database. The program has two global variables `a` and `v` both of them initialized to `false`. Variable `v` is used to store the vote and variable `a` is to store the result of the authentication performed in method `Authen`. The program starts in method `Main`. Method `GetVote` resembles a method to get a vote from a voter for example by using an input device. Method `Authen` resembles an external algorithm used for authenticating the voters. Here, however, these models randomly assign `true` or `false` to variables `v` and `a`. We assume that there are various algorithms for authentication and thus the program is delivered

Syntax			Description
prog	::=	*decl* proc**	A program is a list of global variable declarations followed by a list of method definitions
decl	::=	**decl** *id* **;**	Declaration of variables
	|	**decl** *id* = *val* **;**	
id	::=	[a-zA-Z_][a-zA-Z0-9_]*	An identifier is a C-style identifier
val	::=	**ture**	
	|	**false**	
proc	::=	*id* (*idlst*) { *decl* sseq* }	Function definition
idlst	::=	*id*	
	|	*id* **,** *idlst*	
sseq	::=	$stmt^+$	Sequence of statements
stmt	::=	*id* = *id* **;**	Assignment statement
	|	*id* = *val* **;**	Conditional statement
	|	**if** (*bexpr*) { *sseq* } **else** { *sseq* }	Iteration statement
	|	**while** (*bexpr*) { *sseq* }	Method call
	|	*id* (*idlst*) **;**	Return statement with return value
	|	**return** *id* **;**	Return statement
	|	**return ;**	
bexpr	::=	*bexpr binop bexpr*	
	|	**!** *bexpr*	Negation
	|	(*bexpr*)	
	|	*expr*	
binop	::=	**&&**	And
	|	||	Or
expr	::=	*id eqop id*	
	|	*id eqop val*	
	|	*	Non-deterministic choice
eqop	::=	==	Equality
	|	!=	Inequality

Table 1: Syntax of Boolean Programs

without any specific implementation for method `Authen`; it becomes complete by adding a concrete implementation of this method depending on the preferred authentication algorithm. Two implementations for this method are given in the left column of Figure 5 in which the one at the top is a legitimate implementation and the second one is an implementation by an intruder to manipulate the voting results.

Our task is to prove that a behavioral global property holds for the program, given the structural local specifications of its variable components. In the example above, method `Authen` is a variable component. A behavioral global property and the local specification of `Authen` are given in Figure 5.

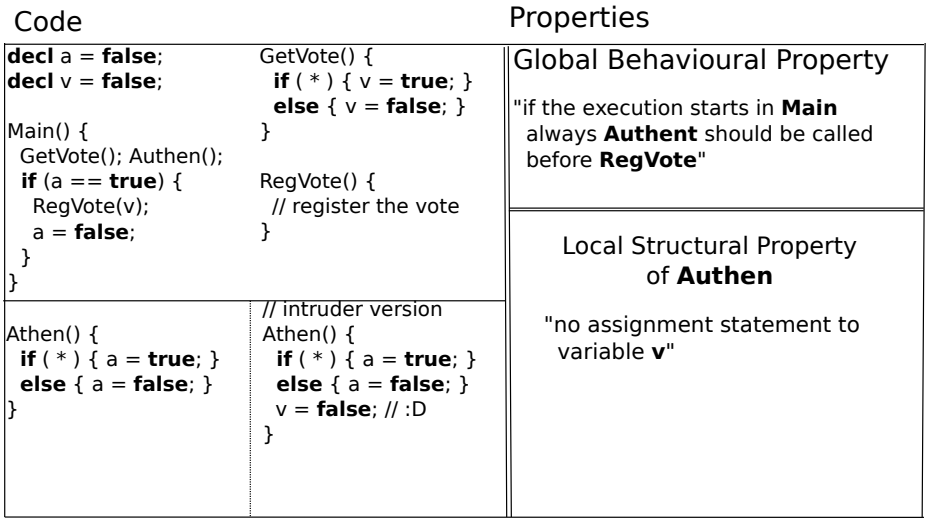


Fig. 5: Referendum voting program

8.2 Program Models

The definition of flow graphs with Boolean data is an instantiation of the definition of generic flow graphs with the assertions and actions defined below.

State Assertions. To formally define the set of state assertions of Boolean programs, we assume, w.l.o.g., that all methods have n local variables and w parameters. We represent the variables of a program by the set $V_{bool} = \texttt{Glob} \cup \texttt{Loc} \cup \texttt{Par}$, where $\texttt{Glob} = \{\texttt{g}_1, \ldots, \texttt{g}_\texttt{k}\}$ is the set of *global variables*, $\texttt{Loc} = \{\texttt{l}_1, \ldots, \texttt{l}_\texttt{n}\}$ is the set of *local variables*, and $\texttt{Par} = \{\texttt{p}_1, \ldots, \texttt{p}_\texttt{w}\}$ is the set of *parameters*. We let $v_1, \ldots, v_z$ represent the variables in V_{bool}. We analogously define the set $V'_{bool} = \texttt{Glob}' \cup \texttt{Loc}' \cup \texttt{Par}'$ as the set of *primed variables* where $\texttt{Glob}'$, $\texttt{Loc}'$, and $\texttt{Par}'$ are the sets of primed global variables, local variables, and parameters, respectively.

In addition to the variables above, we define a reserved global variable `ret` that will be used to store the return value from a method call. The variable `ret` does not have a primed version because it is not used in the program's code and we assume its value can only be changed through a return statement.

Atomic formulas *Cond* that form the assertions are equality and inequality checks over program and primed variables. They are of the form $v_1 = v_2$, $v_1 \neq v_2$, or $v_2 = val$, where $v_1 \in V_{bool}$, $v_2 \in V_{bool} \cup V'_{bool}$, and $val = \{\texttt{true}, \texttt{false}\}$. The set of assertions is $P_{bool} = \{\bigwedge C \mid C \subseteq Cond\}$. Then the preorder $\sqsubseteq$ on assertions is defined by *logical implication*.

Actions. Since we are interested in properties expressing the value of variables exactly before and after method calls, we do not identify any particular program

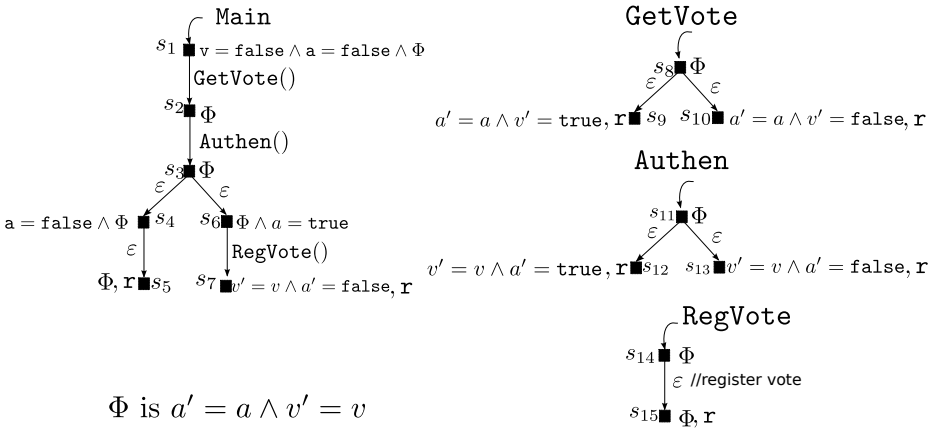


Fig. 6: Flow Graph Example for Boolean Program in Figure 5

instructions as actions. However, we include the neutral action ε for simplicity, and thus the set of actions of Boolean programs is the singleton set $\mathcal{A}_{bool} = \{\varepsilon\}$.

Example 8. Figure 6 shows a flow graph of the Boolean program shown in Figure 5. The guard of the if statement in the code of method `Main` is captured by the state assertions assigned to states s_4 and s_6, and assignment `a:=false` is captured by the assertion of state s_7. Variable `ret` is not shown in this example to simplify the presentation. Ignoring this variable does not affect the example, because methods do not return values.
The interface of method `Authen` is $M^+ = \{\texttt{Authen}\}, M^- = \{\}, \mathit{Modify} = \{a\}$.

Program States. Program states of Boolean programs are mappings from the set of variables V_{bool} to their values in $\{\texttt{false}, \texttt{true}\}$.

Flow Graph Behavior. The flow graph behavior for Boolean programs is an instantiation of Definition 4 for the set of actions $\mathcal{A}_{bool}$ and their semantics. We formalize the state updates by multiple mappings of the form $\sigma[v_i \mapsto \sigma(g_i)]_{1 \le i \le k}$ (or $\sigma[v_i \mapsto c]_{1 \le i \le k}$), which for $i \in \{1, \cdots, k\}$ updates the equivalence class of variable v_i in σ to the one of g_i in σ (or concrete value c).

$$[\![\varepsilon]\!]\sigma'_1 = \sigma'_1$$

$$[\![call]\!]\sigma'_1 = \sigma'_1[p_i \mapsto \sigma'_1(a_i)]_{1 \le i \le w} \, [l_i \mapsto \texttt{false}]_{1 \le i \le n}$$

$$[\![ret]\!](\sigma'_1, \sigma_2) = \sigma_2[g_i \mapsto \sigma'_1(g_i)]_{1 \le i \le k} \, [\texttt{ret} \mapsto \sigma'_1(v)]$$

Example 9. An example sequential run through the branching behavior of the flow graph of Figure 6 is shown below. In the run, the program states are represented by boxes. The value of the variables in the left box are `false`, and the ones in the right box are `true`. For example $\boxed{\texttt{a}|\texttt{v}}$ shows that the value of variable `a` is `false` and the value of variable `v` is `true`.

Notice that the assertion of state s_1 (i.e., $\texttt{a} = \texttt{false} \wedge \texttt{v} = \texttt{false}$) restricts the initial program state to $\boxed{\texttt{a, v}\,|\,}$.

$$(\langle s_1, \boxed{\texttt{a, v}\,|\,}, \boxed{\texttt{a, v}\,|\,}\rangle, \epsilon)$$

$$\xrightarrow{\texttt{Main call GetVote()}} (\langle s_8, \boxed{\texttt{a, v}\,|\,}, \boxed{\texttt{a, v}\,|\,}\rangle, \langle s_2, \boxed{\texttt{a, v}\,|\,}\rangle)$$

$$\xrightarrow{\varepsilon} (\langle s_9, \boxed{\texttt{a, v}\,|\,}, \boxed{\texttt{a}|\texttt{v}}\rangle, \langle s_2, \boxed{\texttt{a, v}\,|\,}\rangle)$$

$$\xrightarrow{\texttt{GetVote ret Main}} (\langle s_2, \boxed{\texttt{a}|\texttt{v}}, \boxed{\texttt{a}|\texttt{v}}\rangle, \epsilon)$$

$$\xrightarrow{\texttt{Main call Authen()}} (\langle s_{11}, \boxed{\texttt{a}|\texttt{v}}, \boxed{\texttt{a}|\texttt{v}}\rangle, \langle s_3, \boxed{\texttt{a}|\texttt{v}}\rangle)$$

$$\xrightarrow{\varepsilon} (\langle s_{12}, \boxed{\texttt{a}|\texttt{v}}, \boxed{\,|\texttt{a, v}}\rangle, \langle s_3, \boxed{\texttt{a}|\texttt{v}}\rangle)$$

$$\xrightarrow{\texttt{Athen ret Main}} (\langle s_3, \boxed{\,|\texttt{a, v}}, \boxed{\,|\texttt{a, v}}\rangle, \epsilon)$$

$$\xrightarrow{\varepsilon} (\langle s_6, \boxed{\,|\texttt{a, v}}, \boxed{\,|\texttt{a, v}}\rangle, \epsilon)$$

$$\xrightarrow{\texttt{Main call RegVote()}} (\langle s_{14}, \boxed{\,|\texttt{a, v}}, \boxed{\,|\texttt{a, v}}\rangle, \langle s_7, \boxed{\,|\texttt{a, v}}\rangle)$$

$$\xrightarrow{\varepsilon \text{ //register the vote}} (\langle s_{15}, \boxed{\,|\texttt{a, v}}, \boxed{\,|\texttt{a, v}}\rangle, \langle s_7, \boxed{\,|\texttt{a, v}}\rangle)$$

$$\xrightarrow{\texttt{RegVote ret Main}} (\langle s_7, \boxed{\,|\texttt{a, v}}, \boxed{\texttt{a}|\texttt{v}}\rangle, \epsilon)$$

8.3 Logics

The structural and behavioral logics are also defined as instantiations of generic simulation logic (Definition 9) for the sets of assertions and actions defined above.

Example 10. The local structural property of method `Authen` shown in Figure 5 is specified by the structural formula $(X)[\Pi]$, where Π is:

$$X = \bigwedge_{l \in L_s} [l]X \ \wedge \ \texttt{v}' = \texttt{v}$$

and $L_s = \{\varepsilon, \texttt{Authen}()\}$.

The global property in Figure 5 can be formalized in behavioral simulation logic $(\texttt{Main} \Rightarrow X)[\Pi]$, where Π is:

$$X = [\texttt{Main call RegVote}()]\texttt{ff} \ \wedge \ \bigwedge_{l \in L} [l]X$$

and $L = L_b \backslash \{\texttt{Main call RegVote}(), \texttt{Main call Authen}()\}$.

8.4 Maximal Flow Graphs

Maximal flow graphs are constructed by applying the algorithm described in Section 5 for the set of structural labels (method names and actions), atomic propositions (method names and `r`), and the top element of the lattice of state assertions ordered by $\sqsubseteq$.

Example 11. As a first step of constructing a maximal flow graph for method `Authen`, the following characteristic formula is constructed form its interface.

$(X0 \vee X1)[\Pi]$ where Π is

$$X0 = ([\varepsilon](X0 \vee X1) \wedge [\texttt{Authen()}](X0 \vee X1) \wedge \texttt{r} \wedge \texttt{Authen} \wedge v' = v)$$
$$X1 = ([\varepsilon](X0 \vee X1) \wedge [\texttt{Authen()}](X0 \vee X1) \wedge \neg\texttt{r} \wedge \texttt{Authen} \wedge v' = v)$$

The above formula is conjoined with the local structural specification of method `Authen` presented in Section 8.3 and the maximal flow graph shown in Figure 7 is constructed.

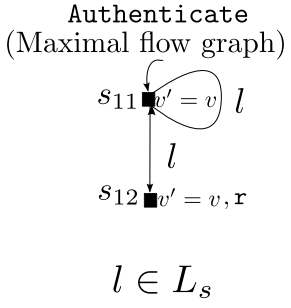


Fig. 7: Maximal Boolean flow graph of method `Authen`
$L_s = \{\varepsilon, \texttt{Authen}()\}$

Note: The characteristic formula constructed from the interface of method `Authen` only allows modifications to variable `a`, forbidding any change to variable `v`. The local property of this method essentially expresses the same constraint on the implementation of the method. For the maximal flow graph construction, where the constraints specified by interfaces and local properties are conjoined, it is sufficient to express such constraints either in interfaces or local properties.

8.5 Compositional Verification

We use Rule 2 for the program models, logics and maximal flow graphs defined in sections 8.2, 8.3, and 8.4. As a result, we obtain a compositional verification technique for these programs.

Example 12. Consider the program in Figure 5 and its properties. We use Rule 2 to verify this program, assuming method `Authen` as the only variable component. This divides the verification into the following two independent sub-tasks. (i) We check that the local specification of method `Authen` is satisfied by its implementation (whenever it is available or changed). (The implementation of `Authen` at the top satisfies its local structural specification, while the one at the bottom does not.) (ii) A maximal flow graph is constructed from the interface and local specification of method `Authen` and composed with the flow graph of the remaining methods of the program. Then the behavior induced from the resulting flow graph is model checked against the global property of the program.

9 Instantiation of the Framework for the PoP Language

In this section, we present an instantiation of our generic verification framework for Pointer Programs (PoP), that are programs with pointers as the only datatype. We instantiate the definitions of flow graph, flow graph behavior, logic, and maximal flow graph construction algorithm to the PoP language.

9.1 syntax

The syntax of the PoP language is an adaptation of the syntax of Boolean programs shown in Table 1 with two additional statements:

new *id* **;**	creating a fresh pointer and assigning it to a variable
del *id* **;**	deleting the pointer a variable pointing to

Also, in the PoP language, the concrete values for variables do not exist, i.e., variables cannot be assigned to values, and conditional statements are only equality and inequality checks on variables and non-deterministic choice. Thus, declaration **decl** *id* = *val* **;**, statement *id* = *val*, and Boolean expression *id eqop val* do not exist in the syntax of PoP language.

PoP method calls are call-by-reference. A method has a return value if an expression or a variable name is provided to the return statement, otherwise the method returns `null` by reaching a return statement or the end of its execution.

We illustrate our approach on a web application that has local variables and parameters. The program in Figure 8 is an implementation of a container and two implementations of a servlet, in which the one in the right column extends the left one by adding a logging facility through calling method `LogSys`. In the code, the variables are pointers to requests. The global variable `P` points to the last-received request, while `c` to the current one. At the beginning of the execution, the request `c` is received by `Container` and passed to `Servlet` that non-deterministically calls back `Container` with the current or a fresh request which variable `n` points to. By this, we mimic the call-backs to `Container` in a real web-application when servlets call each other via the container. `Container` drops (i.e., deletes) the requests that are bounced back to it (when `P = c`) to avoid cycles in the computation. The code of method `LogSys` is not shown in

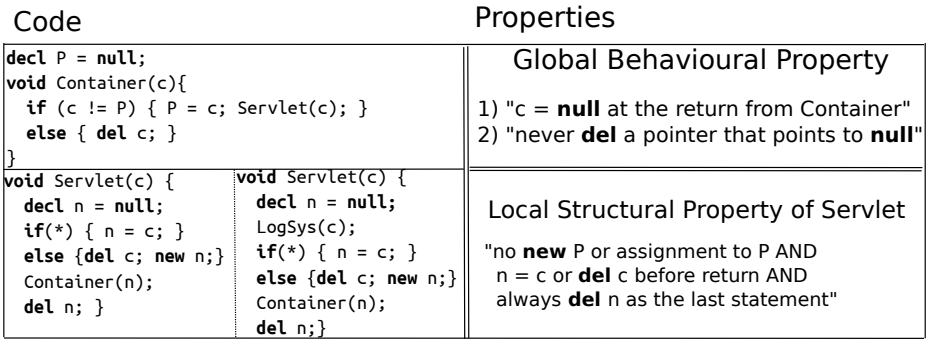


Fig. 8: Web Server Application

the figure but we assume that it does not modify the global variables. In this example, again we consider each method as a component and we consider method `Servlet` as the variable part of the program.

The structural local specifications of methods `Servlet` and `LogSys`, and a behavioral global property are given in the figure. As mentioned, in our framework, structural local specifications are more meaningful than behavioral ones. The reason is that they express properties of the syntax of programs rather than their execution, thus can independently be checked against the component's code rather than the execution of the whole program. For example, a behavioral local specification of method `Servlet` is "the parameter `c` points to `null` at any return point of method `Servlet`"[3], which cannot be checked for method `Servlet` in isolation from `Container` and `LogSys`, because the value of `c` is not clear after the return from the call to `Container`.

As we shall see, to verify the behavioral global property, the structural local specifications of methods `Servlet` and `LogSys` are checked independently (sub-task (i)), and that the composition of these local specifications with the implementation of `Container` entails the global property (sub-task (ii)).

9.2 Program Models

The definition of flow graphs for PoP programs is defined as an instantiation of our generic flow graph notion with the state assertions and actions defined below.

State Assertions. The state assertions of PoP programs are defined similarly to those of Boolean programs. Again, to formally define the set of state assertions of PoP programs, we assume, w.l.o.g., that all methods have n local variables and w parameters. Program variables are represented by the set $V_{pop} = \mathtt{Glob} \cup \mathtt{Loc} \cup \mathtt{Par}$, where $\mathtt{Glob} = \{\mathtt{g}_1, \ldots, \mathtt{g}_k\}$ is the set of *global variables*, $\mathtt{Loc} = \{\mathtt{l}_1, \ldots, \mathtt{l}_n\}$ is the set of *local variables*, and $\mathtt{Par} = \{\mathtt{p}_1, \ldots, \mathtt{p}_w\}$ is the set of *parameters*. We let

[3] Which gives rise to infinitely many implementations.

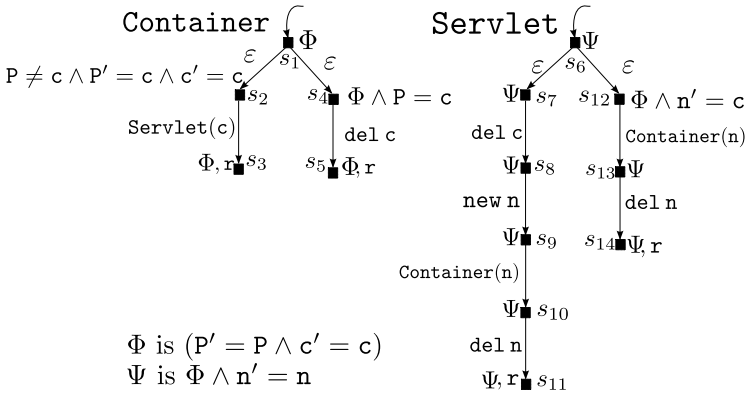


Fig. 9: Flow Graph Example

$v_1, \ldots, v_z$ represent the variables in V_{pop}. Again, we analogously define the set $V'_{pop} = \texttt{Glob}' \cup \texttt{Loc}' \cup \texttt{Par}'$ as the set of *primed variables* where $\texttt{Glob}'$, $\texttt{Loc}'$, and $\texttt{Par}'$ are the sets of primed global variables, local variables, and parameters, respectively.

In addition to the variables above, we define a reserved global variable `ret` that will be used to store the return value from a method call. The variable `ret` does not have a primed version because it is not used in the program's code and we assume that its value can only be changed by a return statement.

Atomic formulas *Cond* that form the logical assertions are equality and inequality checks over program and primed variables of the form $v_1 = v_2$ or $v_1 \neq v_2$, where $v_1 \in V_{pop}$ and $v_2 \in V_{pop} \cup V'_{pop}$. The set of assertions is $P_{pop} = \{\bigwedge C \mid C \subseteq Cond\}$. Then the preorder $\sqsubseteq$ on assertions is defined by logical entailment.

Actions. The actions of PoP programs are pointer creation and deletion, and a neutral action denoted by ε, thus $\mathcal{A}_{pop} = \{\texttt{del}, \texttt{new}, \varepsilon\}$.The arity of `del` and `new` is one and of ε is zero.

Example 13. Figure 9 shows a flow graph of the PoP program shown in Figure 8. The guard of the if statement and assignment `P:=c` in the code of method `Container` are captured by the state assertions assigned to states s_2 and s_4. Again, variable `ret` is not shown in this example to simplify the presentation. Ignoring this variable does not affect the example, because methods do not return values.
The interface of method `Servlet` is $M^+ = \{\texttt{Servlet}\}, M^- = \{\texttt{Container}, \texttt{LogSys}\}, \mathit{Modify} = \{\}$, where *Modify* is empty, because in the code of method `Servlet` there is no syntactic assignment to the global variable `P`.

Program State. PoP programs can create infinite state spaces for two reasons: unbounded call stacks, and infinite memory allocation. To be able to model check

the behavioral models of such programs, we address the former by modeling the unbounded call stacks as pushdown automata/systems and exploit existing model checking algorithm such as [24]. For the latter however, we abstractly represent the infinite memory allocations by finitely many program states. To achieve this, we exploit the fact that at any point of execution (behavior) of PoP programs, only finitely many memory locations are referenced by program variables. This allows us to represent the allocated memories as a partitioning of variables into equivalence classes. The idea is that two variables are in one equivalence class if they are pointing to the same memory. Then the number of equivalence classes is at most equivalent to the number of variables of the program (when all variables are distinct). Such an abstraction is adequate to capture the behavior of PoP programs because only equality and inequality checks of variables are allowed by the language.

To implement this abstraction, we use an extension of the technique proposed by Rot et al. in [30]. They use a set of so-called "freeze" variables $\mathcal{F}_g = \{g_1^f, \ldots, g_k^f\}$ that consists of one additional local variable for each global variable of the program. These variables are used to store the values of global variables at each method invocation and as a means of recomputation of their values after returns from method calls (see the definition of function $\mathfrak{G}$ in the flow graph behavior). The global variable `ret` does not have a freeze variable. We introduce an additional set of freeze variables for parameters $\mathcal{F}_p = \{p_1^f, \ldots, p_w^f\}$ to provide support for call-by-reference parameter passing. These variables store the values of parameters at the method calls and are used to compute the values of local variables at the returns (see the definition of function $\mathfrak{L}$ in the flow graph behavior).

In addition to variables $\mathcal{F}_g$ and $\mathcal{F}_p$, program states include a set of so called "indicator" variables $\mathbb{D} = \{d_1, \ldots, d_{k+w}\}$ that consists of one Boolean variable for each global variable or parameter. These are used to provide support for delete instructions. Let us explain by an example why these variables are needed. Consider the following code.

```
1 decl x;
2 Caller() {
3  decl l;
4  new x;
5  l = x;
6  Callee();
7 }
```

```
8 Callee() {
9  decl l2;
10 l2 = x;
11 del l2;
12 }
```

In order to have an accurate partitioning of variables, we need to be able to find out the value of local variable `l` upon returning from a call to method `Callee` (line 7). Assume that `l` and `x` point to memory m before the call. Since `l` is local to `Caller`, its value cannot be directly modified in `Callee`. Still, in `Callee`, m can be deleted through deleting the global variable `x`. We set variables in $\mathbb{D}$ to `true` if a deleted variable in the current context points to a memory that is referenced in the previous context. This is checked at every delete operation and the result is stored as a Boolean value in variables in $\mathbb{D}$ (see the definition

of $[\![del]\!]$ in the flow graph behavior). For example, to construct the behavioral model for the code above, while line 11 is executing, the variable in $\mathbb{D}$ that corresponds to global variable `x` is set to `true`. Upon a return from a call, the values of these variables are used to find out if the memory a local variable is pointing to has been deleted during the execution of the called method, through a global variable or parameter (see the definition of function $\mathfrak{L}$ in the flow graph behavior).

Thus, a program state of PoP programs consists of two parts:

1. A partitioning over the set of program and freeze variables $V^f_{pop} = V_{pop} \cup \mathcal{F}_g \cup \mathcal{F}_p$. We define such partitionings by assigning to each variable a natural number that represents its equivalence class.
2. A mapping of variables in $\mathbb{D}$ to `true` or `false`.

As a result, program states are defined formally as $\Sigma : V^f_{pop} \to \{n \mid 0 \leq n \leq |V^f_{pop}| + 1\} \times \mathbb{D} \to \{\texttt{true}, \texttt{false}\}$. By this formulation, two distinct variables v_1 and v_2 belong to the same equivalence class iff $\sigma(v_1) = \sigma(v_2)$. We use 0 for the equivalence class of `null` (also denoted by $\perp$).

Flow Graph Behavior. The flow graph behavior for PoP programs is an instantiation of Definition 4 for the set of actions $\mathcal{A}_{pop}$ and their semantics. To define formally the semantics of actions, we formalize state updates by multiple mappings of the form $\sigma[v_i \mapsto \sigma(g_i)]_{1 \leq i \leq k}$ (or $\sigma[v_i \mapsto c]_{1 \leq i \leq k}$), which updates for $i \in \{1, \cdots, k\}$ the equivalence class of variable v_i in σ to the one of g_i (or the concrete class c). We sometimes use conditional updates of the form $\sigma[b \,?\, update_1 : update_2]$ that is $\sigma[update_1]$ if b is evaluated to *true* and $\sigma[update_2]$ otherwise.

Intuitively, the semantics of ε is the identity function, `del(`v`)` moves v and all of its aliases to the equivalence class of `null` (i.e., 0) and sets the values of variables in $\mathbb{D}$, `new(`v`)` maps v to a fresh equivalence class (through using function $\mathfrak{N}$), *call* initializes a program state for the called method, and *ret* is defined through the functions $\mathfrak{L}$ and $\mathfrak{G}$.

Function $\mathfrak{N}(v, \sigma)$ returns the least natural number between 1 and $|V^f_{pop}| + 1$ which is not used in program state σ, if all numbers are used it returns $\sigma(v)$. Other functions ($\mathfrak{R}$, $\mathfrak{L}$, $\mathfrak{N}$, and $\mathfrak{G}$) are formally defined below. They all return a natural number. Function $\mathfrak{R}$ computes the value of variable `ret` upon a return from a call. Function $\mathfrak{L}$ copies the values of the local variables from the state before the call to the state after its return. The value of a local variable is 0 if it is an alias for a global variable that is deleted in the called method (see Example 14). Function $\mathfrak{G}$ recursively computes the equivalence classes of global variables on a return from a method call based on their equivalence classes before the call and at the return of the call, as in [30]. In short, it moves a global variable to either the equivalence class of `null`, or of another variable, or to a fresh equivalence class, if it is deleted, assigned to another variable in the called method, or assigned to a new pointer, respectively.

$$
\begin{aligned}
[\![\varepsilon]\!]\sigma_1' &= \sigma_1' \\
[\![\texttt{new}(v)]\!]\sigma_1' &= \sigma_1'[v \mapsto \mathfrak{N}(v, \sigma_1')] \\
[\![\texttt{del}(v)]\!]\sigma_1' &= \sigma_1'[\sigma_1'(v) = \sigma_1'(l_i)\ ?\ l_i \mapsto 0 : li \mapsto \sigma_1'(l_i)]_{1\le i\le n} \\
&\qquad [\sigma_1'(v) = \sigma_1'(g_i)\ ?\ g_i \mapsto 0 : g_i \mapsto \sigma_1'(g_i)]_{1\le i\le k} \\
&\qquad [\sigma_1'(v) = \sigma_1'(p_i)\ ?\ p_i \mapsto 0 : p_i \mapsto \sigma_1'(p_i)]_{1\le i\le w} \\
&\qquad [\sigma_1'(v) = \sigma_1'(g_i^f)\ ?\ d_i \mapsto \texttt{tt} : d_i \mapsto \sigma_1'(d_i)]_{1\le i\le k} \\
&\qquad [\sigma_1'(v) = \sigma_1'(p_i^f)\ ?\ d_{i+k} \mapsto \texttt{tt} : d_{i+k} \mapsto \sigma_1'(d_{i+k})]_{1\le i\le w} \\
[\![call]\!]\sigma_1' &= \sigma_1'[g_i^f \mapsto g_i]_{1\le i\le k} \\
&\qquad [p_i \mapsto \sigma_1'(a_i)]_{1\le i\le w} \\
&\qquad [p_i^f \mapsto \sigma_1'(a_i)]_{1\le i\le w} \\
&\qquad [l_i \mapsto 0]_{1\le i\le n} \\
&\qquad [d_i \mapsto \texttt{ff}]_{1\le i\le k+w} \\
[\![ret]\!](\sigma_1', \sigma_2) &= \sigma_2[l_i \mapsto \mathfrak{L}(l_i, \sigma_1', \sigma_2)]_{1\le i\le n} \\
&\qquad [p_i \mapsto \mathfrak{L}(p_i, \sigma_1', \sigma_2)]_{1\le i\le w} \\
&\qquad [g_i \mapsto \mathfrak{G}(i, \sigma_1', \sigma_2)]_{1\le i\le k} \\
&\qquad [\texttt{ret} \mapsto \mathfrak{R}(v, \sigma_1', \sigma_2)]
\end{aligned}
$$

$$
\mathfrak{L}(v, \sigma_1, \sigma_2) = \begin{cases} 0 & \text{if} \quad \begin{pmatrix} \exists\ j\ .\ 1 \le j \le k\ \wedge\ \sigma_2(v) = \sigma_2(g_j)\ \wedge\ \sigma_1(d_j) = \texttt{tt} \vee \\ \exists\ j\ .\ k+1 \le j \le k+w\ \wedge\ \sigma_2(v) = \sigma_2(p_j)\ \wedge\ \sigma_1(d_j) = \texttt{tt} \end{pmatrix} \\ \sigma_2(v) & \text{otherwise} \end{cases}
$$

$$
\mathfrak{G}(i, \sigma_1, \sigma_2) = \begin{cases} 0 & \text{if} & \sigma_1(g_i) = 0 \\ \mathfrak{G}(j, \sigma_1, \sigma_2) & \text{if} & j \text{ is the least index s.t. } \sigma_1(g_i) = \sigma_1(g_j) \\ \sigma_1(g_j^f) & \text{if} & j \text{ is the least index s.t. } \sigma_1(g_i) = \sigma_1(g_j^f) \\ \sigma_1(p_j^f) & \text{if} & j \text{ is the least index s.t. } \sigma_1(g_i) = \sigma_1(p_j^f) \\ \mathfrak{N}(g_i, \sigma_2) & \text{otherwise} & \end{cases}
$$

$$
\mathfrak{R}(v, \sigma_1, \sigma_2) = \begin{cases} \mathfrak{G}(i, \sigma_1', \sigma) & \text{if} & i \text{ is the least index s.t. } \sigma_1'(v) = \sigma_1'(g_i) \\ \sigma_1'(g_i^f) & \text{if} & i \text{ is the least index s.t. } \sigma_1'(v) = \sigma_1'(g_i^f) \\ \sigma_1'(p_i^f) & \text{if} & i \text{ is the least index s.t. } \sigma_1'(v) = \sigma_1'(p_i^f) \\ 0 & \text{if} & v = 0 \\ \mathfrak{N}(v, \sigma_2) & \text{otherwise} & \end{cases}
$$

Example 14. An example sequential run through the branching behavior of the flow graph in Figure 9 is shown below. In the run, the boxes represent the equivalence classes of variables, where the left-most box always represents the class of `null`, e.g., $\boxed{\texttt{P}, \texttt{P}^f \mid \texttt{c}}$ shows that variables `P` and $\texttt{P}^f$ are in the equivalence class of `null` and `c` is in a different one. In this example, we do not show the values of $\mathbb{D}$ variables for simplicity. Their values do not effect the execution.

We start from initial program state $\boxed{\texttt{P},\ \texttt{P}^f, \texttt{c}^f \mid \texttt{c}}$.

$$(\langle s_1, \boxed{\texttt{P}, \texttt{P}^f, \texttt{c}^f \mid \texttt{c}}, \boxed{\texttt{P}, \texttt{P}^f, \texttt{c}^f \mid \texttt{c}} \rangle, \epsilon)$$

$$\xrightarrow{\varepsilon} (\langle s_2, \boxed{\texttt{P}, \texttt{P}^f, \texttt{c}^f \mid \texttt{c}}, \boxed{\texttt{P}^f, \texttt{c}^f \mid \texttt{P}, \texttt{c}} \rangle, \epsilon)$$

$$\xrightarrow{\texttt{Container call Servlet(c)}} (\langle s_6, \boxed{\texttt{n} \mid \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f}, \boxed{\texttt{n} \mid \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f} \rangle, \langle s_3, \boxed{\texttt{P}^f, \texttt{c}^f \mid \texttt{P}, \texttt{c}} \rangle)$$

$$\xrightarrow{\varepsilon} (\langle s_{12}, \boxed{\texttt{n} \mid \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f}, \boxed{\mid \texttt{n}, \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f} \rangle, \langle s_3, \boxed{\texttt{P}^f, \texttt{c}^f \mid \texttt{P}, \texttt{c}} \rangle)$$

$$\xrightarrow{\texttt{Servlet call Container(n)}} (\langle s_1, \boxed{\mid \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f}, \boxed{\mid \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f} \rangle, \langle s_{13}, \boxed{\mid \texttt{n}, \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f} \rangle \cdot \langle s_3, \boxed{\texttt{P}^f, \texttt{c}^f \mid \texttt{P}, \texttt{c}} \rangle)$$

$$\xrightarrow{\varepsilon} (\langle s_4, \boxed{\mid \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f}, \boxed{\mid \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f} \rangle, \langle s_{13}, \boxed{\mid \texttt{n}, \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f} \rangle \cdot \langle s_3, \boxed{\texttt{P}^f, \texttt{c}^f \mid \texttt{P}, \texttt{c}} \rangle)$$

$$\xrightarrow{\texttt{del(c)}} (\langle s_5, \boxed{\texttt{P}, \texttt{c} \mid \texttt{P}^f, \texttt{c}^f}, \boxed{\texttt{P}, \texttt{c} \mid \texttt{P}^f, \texttt{c}^f} \rangle, \langle s_{13}, \boxed{\mid \texttt{n}, \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f} \rangle \cdot \langle s_3, \boxed{\texttt{P}^f, \texttt{c}^f \mid \texttt{P}, \texttt{c}} \rangle)$$

$$\xrightarrow{\texttt{Container ret Servlet}} (\langle s_{13}, \boxed{\texttt{n}, \texttt{P}, \texttt{c} \mid \texttt{P}^f, \texttt{c}^f}, \boxed{\texttt{n}, \texttt{P}, \texttt{c} \mid \texttt{P}^f, \texttt{c}^f} \rangle, \langle s_3, \boxed{\texttt{P}^f, \texttt{c}^f \mid \texttt{P}, \texttt{c}} \rangle)$$

$$\xrightarrow{\texttt{del(n)}} (\langle s_{14}, \boxed{\texttt{n}, \texttt{P}, \texttt{c} \mid \texttt{P}^f, \texttt{c}^f}, \boxed{\texttt{n}, \texttt{P}, \texttt{c} \mid \texttt{P}^f, \texttt{c}^f} \rangle, \langle s_3, \boxed{\texttt{P}^f, \texttt{c}^f \mid \texttt{P}, \texttt{c}} \rangle)$$

$$\xrightarrow{\texttt{Servlet ret Container}} (\langle s_3, \boxed{\texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f}, \boxed{\texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f} \rangle, \epsilon)$$

Consider transition $\xrightarrow{\texttt{Container ret Servlet}}$ and how it updates the program states. At the return state $\langle s_5, (\boxed{\texttt{P}, \texttt{c} \mid \texttt{P}^f, \texttt{c}^f}, \boxed{\texttt{P}, \texttt{c} \mid \texttt{P}^f, \texttt{c}^f}) \rangle$ the global variable `P` is in the equivalence class of `null`. Thus, aliases of `P` (i.e., variables in the equivalence class of `P`) in the state before the call to `Container` (i.e., the state at the top of the stack $\langle s_{12}, \boxed{\mid \texttt{n}, \texttt{P}, \texttt{P}^f, \texttt{c}, \texttt{c}^f} \rangle$) are moved to the equivalence class of `null` in the state after the return from `Container` ($\langle s_{12}, \boxed{\texttt{n}, \texttt{P}, \texttt{c} \mid \texttt{P}^f, \texttt{c}^f} \rangle$).

Also consider the second global property in Figure 1. The transition $\xrightarrow{\texttt{del n}}$ falsifies this property because `n` belongs to the equivalence class of `null` in the state exactly before deleting.

9.3 Logics

The structural and behavioral logics are also defined as instantiations of generic simulation logic (Definition 9) for the sets of assertions and actions defined above. In Example 1, we showed a structural and a behavioral property for the PoP instantiation of the generic simulation logic. Here, we specify properties expressed informally in Figure 8.

Example 15. The local structural property of method `Servlet()` in Figure 8 is specified by the structural formula $(X_0 \wedge X_1 \wedge X_2)[\Pi]$, where Π is

$$\begin{aligned}
X_0 &= [\texttt{new(P)}]\texttt{ff} \wedge \textstyle\bigwedge_{l \in L_{\texttt{Serv}} \setminus \texttt{new(P)}} [l] X_0 \\
X_1 &= (\neg \texttt{r} \wedge \textstyle\bigwedge_{l \in L_{\texttt{Serv}} \setminus \{\texttt{del(c)}\}} [l] X_1) \vee \texttt{n'=c} \\
X_2 &= [\texttt{del(n)}] X_3 \wedge \textstyle\bigwedge_{l \in L_{\texttt{Serv}} \setminus \{\texttt{del(n)}\}} [l] X_2 \\
X_3 &= \textstyle\bigwedge_{l \in L_{\texttt{Serv}}} [l]\texttt{ff} \wedge \texttt{n'=n}
\end{aligned}$$

where `Container` is the set of labels representing calls to the method `Container`. In the formula, X_0 formalizes "no `new P` statement", the equation X_1 formalizes "`n := c` or `del c` before return", and the equations X_2 and X_3 together formalize "no statement after `del n`". Notice that the formula does not formalize the

constraint "no assignment statement to P", because in the interface of method Servlet (given in the Flow Graph paragraph), $\mathtt{P} \notin \mathit{Modify}$.

The first global behavioral property in Figure 8 is specified by the behavioral formula $X[\Pi]$, where Π is

$$X = \bigwedge_{l \in L_b} [l]X \wedge (\mathtt{r} \wedge \mathtt{Container} \rightarrow (\mathtt{c} = \mathtt{null}))$$

The syntax of the formulas may look complicated, but syntactic sugar can be introduced by means of macros or specification templates to simplify the presentation (e.g., see [19]).

9.4 Maximal Flow Graph

The maximal flow graphs are constructed by applying the algorithm described in Section 5 for the set of structural labels (method names and actions), atomic propositions (method names and r), and the top element of the lattice ordered by $\sqsubseteq$ (which is here logical implication).

Example 16. We construct the maximal flow graph of method Servlet from its interface and local structural property. To do this, the following characteristic formula is constructed from the interface.

$(X0 \vee X1)[\Pi]$ where Π is

$$\begin{aligned} X0 = {} & ([\varepsilon](X0 \vee X1) \wedge [\mathtt{Container}](X0 \vee X1) \wedge \\ & [\mathtt{Servlet}](X0 \vee X1) \wedge \mathtt{r} \wedge \mathtt{Servlet} \wedge P' = P) \\ X1 = {} & ([\varepsilon](X0 \vee X1) \wedge [\mathtt{Container}](X0 \vee X1) \wedge \\ & [\mathtt{Servlet}](X0 \vee X1) \wedge \neg\mathtt{r} \wedge \mathtt{Servlet} \wedge P' = P) \end{aligned}$$

The above formula is conjoined with the local structural property of Servlet and maximal flow graph shown in Figure 10 is constructed.

9.5 Compositional Verification

We use Rule 2 for the program models, logics and maximal flow graphs defined for PoP programs. As a result, we obtain a compositional verification technique for PoP programs.

Example 17. Consider the program in Figure 8 and its properties. We use the PoP instantiation of Rule 2 to verify this program, assuming method Servlet as a variable component. This divides the verification into the following two independent sub-tasks. (i) We check that the local specification of method Servlet is satisfied by its implementation (whenever it is available or changed). The implementation of Servlet in the figure satisfies its local structural specification. (ii) A maximal flow graph is constructed from the interface and local structural property of method Servlet and composed with the flow graph of method Container. Then the behavior induced from the resulting flow graphis model checked against the global properties of the program.

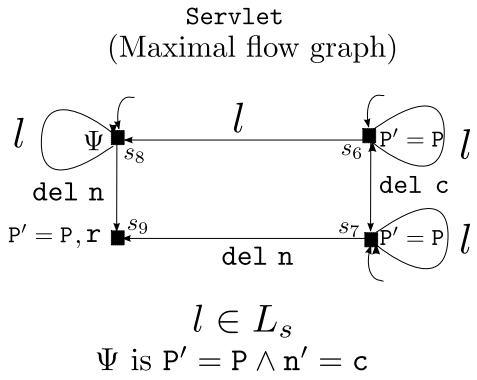


Fig. 10: Maximal flow graph constructed from local structural property and interface of method `Servlet`

9.6 Tool Support and Evaluation.

We have extended our compositional verification toolset [22] for the verification of PoP programs in the presence of variability. Apart from the necessary data structures, the toolset includes a maximal flow graph constructor, a tool to induce behaviors from flow graphs, and external model checkers CWB [14] and Moped [24]. We used this toolset to verify a Java J2EE application consisting of 1087 lines of code, of which 297 lines are variable.

We focused on properties of *database connections*, such as "at the end of the execution, all database connections should have been closed". We therefore abstracted away all program data except variables of this type, constructed and destructed by invoking methods `getConnection` and `close`, respectively. To extract flow graphs with this abstraction, we first extracted a data-less flow graph from the Java code with our flow graph extractor tool ConFlEx [6,17]. Then we manually inserted all 4 database connection variables of the program into the extracted flow graphs and replaced any call to `getConnection` and `close` methods with `new` and `del` actions, respectively. This was necessary because currently we do not have a tool to extract PoP flow graphs from code. We also specified each method of the program with a structural local specification, expressing its safe sequences of invocation of methods `getConnection` and `close` (here renamed to `new` and `del`). We then (i) model checked the flow graphs of variable components against their corresponding local specifications with CWB (took 0.5 sec.), and (ii) constructed maximal flow graphs from the local specifications of the variable components (took 4.1 sec.), composed them with the flow graphs of the other components and model checked the result against a property of database connection with Moped (took 2.1 sec.). Recall that to re-verify the program after a change in the variable components only sub-task (i) needs to be repeated.

10 Related Work

In the context of compositional verification of temporal properties, the maximal model technique was first proposed by Grumberg and Long for ACTL, the universal fragment of CTL [18], and later generalized by Kupferman and Vardi for ACTL* [26]. These works do not address the verification of infinite state systems. In our previous work, we used maximal models constructed from safety μ-calculus formulas to verify infinite state context-free behaviors, where the program data is disregarded [19]. In this work we extend our previous work to a generic framework that captures program data.

For a different class of properties, Hoare logic provides a popular framework for compositional verification of programs, (see e.g. [28]) that is technically capable of verifying programs with variability. Also, of particular interest to our technique is the work by Alur and Chaudhuri [3], which proposes a unification of Hoare logic and Manna-Pnueli-style temporal reasoning by defining a set of proof rules for the verification of some particular classes of (non-regular) temporal properties. Our technique is partially inspired by this work.

Related to our approach of relativizating global properties on local specifications, Andersen introduces partial model checking in which global properties of concurrent systems are reduced to local properties of their components (processes) [7]. The work only considers finite-state systems; however, the approach suggests the possibility of extending our technique to generate local properties for variable components of programs when the global properties are fixed.

Several successful tools and techniques exist for (non-compositional) verification of behavioral properties of procedural programs. However, as mentioned, compositionality is essential for the verification of variable programs. Still, related to our two-step verification procedure, tools such as SLAM [12] and ESP [16] divide the verification into (local) intra- and (global) inter-procedural analysis to achieve scalability. It is interesting to explore if the ideas presented here can be used to adapt these tools for the verification of systems with variability.

Closely related to our flow graph model are *recursive state machines* [2], defined by Alur and others as a formalism to model procedural programs with recursive calls. The authors propose efficient LTL and CTL* model checking algorithms. However, they do not address compositional verification.

As for specification languages, the temporal logic of nested calls and returns [5] and its generalization to nested words [4,1] are of particular interest to this work. These logics are capable of abstracting internal computations by moving from a call to its corresponding return point in one step. However, they do not make a clear separation of structure and behavior, and may therefore require more involved maximal model constructions.

11 Conclusion

This paper presents a generic framework for compositional verification of temporal safety properties of sequential procedural programs in the presence of vari-

ability. The framework is a generalization of a previously developed framework which disregards program data. Our technique relies on local specifications of the variable components, in that the correctness of global properties of the program is relativized on the composition of the maximal flow graphs constructed from these local specifications and the flow graphs of the stable components.

The framework is parametric on a set of selected "visible" program instructions that are explicitly represented as transition labels, while the effect of all other instructions is captured abstractly by means of Hoare-style state assertions. This distinction allows to keep the level of detail of specifications within practical limits. It also allows a (symbolic) formulation of the maximal model construction for program models with data that does not add to the complexity of the construction for models without data. To evaluate our technique in practice, we provide tool support for the verification of evolving PoP programs.

In the current setting, our (symbolic) flow graphs induce behaviors with concrete data from finite domains. We conjecture that program data can be represented symbolically in the behaviors as well, using the state assertions of the structural program model (Definition 1). We plan to investigate the expressiveness of symbolic behaviors. We are currently working on a parametric flow graph extractor to extract flow graphs of Java programs for the given sets of actions and assertions. We also plan to provide tool support for the verification of programs with other datatypes, such as integers and Booleans.

References

1. R. Alur, M. Arenas, P. Barcelo, K. Etessami, N. Immerman, and L. Libkin. First-order and temporal logics for nested words. In *Logic in Computer Science (LICS '07)*, pages 151–160, Washington, DC, USA, 2007. IEEE Computer Society.
2. R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27:786–818, 2005.
3. R. Alur and S. Chaudhuri. Temporal reasoning for procedural programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI '10)*, volume 5944 of *Lecture Notes in Computer Science*, pages 45–60. Springer, 2010.
4. R. Alur, S. Chaudhuri, and P. Madhusudan. Languages of nested trees. In *Computer Aided Verification (CAV '06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 329–342. Springer, 2006.
5. R. Alur, K. Etessami, and P. Madhusudan. A temporal logic for nested calls and returns. In *Tools and Algorithms for the Analysis and Construction of Software (TACAS '04)*, volume 2998 of *Lecture Notes in Computer Science*, pages 467–481. Springer, 2004.
6. A. Amighi, P. de Carvalho Gomes, D. Gurov, and M. Huisman. Sound control-flow graph extraction for java programs with exceptions. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *Software Engineering and Formal Methods (SEFM)*, volume 7504 of *Lecture Notes in Computer Science*, pages 33–47. Springer Berlin / Heidelberg, 2012.
7. H. Andersen. Partial model checking (extended abstract). In *Logic in Computer Science (LICS '95)*, pages 398–407. IEEE Computer Society Press, 1995.

8. A. Arnold and D. Niwiński. *Rudiments of μ-calculus*, volume 146 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Publishing, 2001.
9. T. Ball, A. Podelski, and S. Rajamani. Boolean and cartesian abstraction for model checking c programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2001.
10. T. Ball and S. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, 2000.
11. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 113–130, London, UK, UK, 2000. Springer-Verlag.
12. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Principles of programming languages (POPL '02)*, pages 1–3, 2002.
13. H. Bekič. Definable operators in general algebras, and the theory of automata and flowcharts. Technical report, IBM Laboratory, 1967.
14. R. Cleaveland, J. Parrow, and B. Steffen. A semantics based verification tool for finite state systems. In *International Symposium on Protocol Specification, Testing and Verification*, pages 287–302. North-Holland Publishing Co., 1990.
15. B. Cook, D. Kroening, and N. Sharygina. *Symbolic model checking for asynchronous boolean programs*. Springer, 2005.
16. M. Das, S. Lerner, and M. Seigle. ESP: Path–sensitive program verification in polynomial time. In *Programming Language Design and Implementation (PLDI '02)*, pages 57–68. ACM, 2002.
17. P. de Carvalho Gomes, A. Picoco, and D. Gurov. Sound control flow graph extraction from incomplete java bytecode programs. In *Fundamental Approaches to Software Engineering (FASE)*, volume 8411 of *Lecture Notes in Computer Science*, pages 215–229, Berlin, 2014. Springer.
18. O. Grumberg and D. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):843–871, 1994.
19. D. Gurov, M. Huisman, and C. Sprenger. Compositional verification of sequential programs with procedures. *Information and Computation*, 206(7):840–868, 2008.
20. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32:137–161, 1985.
21. M. Huisman, I. Aktug, and D. Gurov. Program models for compositional verification. In *International Conference on Formal Engineering Methods (ICFEM '08)*, volume 5256 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 2008.
22. M. Huisman and D. Gurov. CVPP: A tool set for compositonal verification of control-flow safety properties. In *Formal Verification of Object–Oriented Software (FoVeOOS '10)*, volume 6528 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2010.
23. Servlet Development. Release 2 (9.0.3) `http://docs.oracle.com/cd/A97688_16/generic.903/a97680/develop.htm#1007089`.
24. S. Kiefer, S. Schwoon, and D. Suwimonteerabuth. Moped - a model-checker for pushdown systems. `http://www.informatik.uni-stuttgart.de/fmi/szs/tools/moped/`.
25. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

26. O. Kupferman and M. Vardi. An automata-theoretic approach to modular model checking. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):87–128, 2000.
27. K. Larsen. Modal specifications. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 232–246. Springer, 1989.
28. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer, 2002.
29. A. Podelski and T. Wies. Boolean heaps. In C. Hankin and I. Siveroni, editors, *Static Analysis*, volume 3672 of *Lecture Notes in Computer Science*, pages 268–283. Springer Berlin Heidelberg, 2005.
30. J. Rot, F. de Boer, and M. Bonsangue. A pushdown system representation for unbounded object creation. In *Informal pre-proceedings of Formal Verification of Object–Oriented Software (FoVeOOS '10)*, 2010.
31. J. Rot, F. S. de Boer, and M. M. Bonsangue. Unbounded allocation in bounded heaps. In *Fundamentals of Software Engineering (FSEN)*, volume 8161 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2013.
32. S. Soleimanifard, D. Gurov, and M. Huisman. Procedure–modular verification of control flow safety properties. In *Workshop on Formal Techniques for Java Programs (FTfJP '10)*, 2010.
33. S. Soleimanifard, D. Gurov, and M. Huisman. Procedure-modular specification and verification of temporal safety properties. *Software & Systems Modeling*, pages 1–18, 2013.
34. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–310, 1955.