Procedure-Modular Verification of Temporal Safety Properties

SIAVASH SOLEIMANIFARD

Licentiate Thesis
Stockholm, Sweden, 2012
Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie licentiatexamen i datalogi onsdagen den 23 maj 2012 klockan 13.00 i sal D3 Lindstedtsvägen 5 Kungl Tekniska högskolan, Stockholm.

© Siavash Soleimanifard, April 23, 2012

Tryck: E-print
Abstract

This thesis presents a fully automated technique for procedure-modular verification of control flow temporal safety properties. Procedure-modular verification is a natural instantiation of modular verification where modularity is achieved at the level of procedures. Here it is used for the verification of software systems 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). The technique is built on top of a previously developed modular verification framework based on maximal model construction. In the framework, program data is abstracted away completely to achieve algorithmic verification. This restricts the class of properties that can be verified. The technique is supported by a fully automated tool called ProMoVer which is described and evaluated on a number of real-life case studies. ProMoVer is quipped with a number of features, such as automatic specification extraction, to facilitate easy usage. Moreover, it provides a proof storage and reuse mechanism for efficiency.

An application area which can significantly benefit from modular verification is software product line (SPL) design. 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 a hierarchical variability model 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 hierarchical model. Therefore, scalable and efficient verification for SPL is only possible by exploiting modular verification techniques. In this thesis, we propose a hierarchical variability model for modeling product families. Then the modular verification technique and ProMoVer are adapted for the SPLs described with this hierarchical model.

A natural extension of the modular verification technique is to include program data in a conservative fashion, by encoding data from a finite domain through control. By this, a wider class of properties can be supported. As a first step towards including program data, Boolean values are added to the program model, specification languages, maximal model construction and modular verification principles.
# Contents

## Contents

<table>
<thead>
<tr>
<th>CONTENTS</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements</td>
<td>vii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Preliminaries</td>
<td></td>
</tr>
<tr>
<td>2.1 Models</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Pushdown Automata and Systems</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Temporal Logics</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Model Checking</td>
<td>13</td>
</tr>
<tr>
<td>2.5 Background on Compositional Verification of Temporal Properties</td>
<td>13</td>
</tr>
<tr>
<td>3 A Verification Framework</td>
<td></td>
</tr>
<tr>
<td>3.1 Model and Logic</td>
<td>16</td>
</tr>
<tr>
<td>3.2 Program Model</td>
<td>18</td>
</tr>
<tr>
<td>3.3 Maximal Flow Graph</td>
<td>21</td>
</tr>
<tr>
<td>3.4 Compositional Verification</td>
<td>25</td>
</tr>
<tr>
<td>3.5 Private Method Abstraction</td>
<td>25</td>
</tr>
<tr>
<td>4 A Verification Tool Set</td>
<td></td>
</tr>
<tr>
<td>4.1 Overview of the Tool Set</td>
<td>27</td>
</tr>
<tr>
<td>4.2 Typical Verification Scenarios</td>
<td>30</td>
</tr>
<tr>
<td>5 ProMoVer</td>
<td></td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>33</td>
</tr>
<tr>
<td>5.2 A User’s View</td>
<td>34</td>
</tr>
<tr>
<td>5.3 Flow Graph Behaviour</td>
<td>36</td>
</tr>
<tr>
<td>5.4 Specification Languages</td>
<td>37</td>
</tr>
<tr>
<td>5.5 Modular Verification</td>
<td>38</td>
</tr>
<tr>
<td>5.6 ProMoVER Overview</td>
<td>39</td>
</tr>
<tr>
<td>5.7 Experiments</td>
<td>42</td>
</tr>
<tr>
<td>5.8 Conclusion</td>
<td>46</td>
</tr>
</tbody>
</table>
CONTENTS

6 Verification of Software Families 47
   6.1 Introduction ................................................. 47
   6.2 Related Work ............................................ 48
   6.3 A Hierarchical Variability Model ............................ 49
   6.4 Compositional Verification of SHVM .................. 53
   6.5 Experiments ............................................. 57
   6.6 Conclusion ............................................... 59

7 Boolean Flow Graphs 61
   7.1 Program Model ........................................... 61
   7.2 Maximal Boolean Flow Graphs .............................. 65
   7.3 Compositional Verification ....................... 67
   7.4 Examples ............................................... 67
   7.5 Conclusion ............................................. 74

8 Conclusion 75
   8.1 Discussion ............................................. 75
   8.2 Future Work ........................................... 76

Bibliography 77
Acknowledgements

It is my pleasure to write some lines and express my gratitude towards my colleagues and other people who have supported me during past 2.5 years.

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 is insightful.

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

I am thankful to Duoglas Wikström for reading the drafts of this thesis and for his valuable comments. Your comments truly helped me improve this thesis. Thanks to Musard Balliu, Pedro de Carvalho Gomes, Gurvan Le Guernic, and Narges Khakpour for reading and commenting on some parts of this thesis. I believe your comments enhanced the presentation of the work in this thesis.

I would like to express my appreciation to Pedro de Carvalho Gomes, Emma Enström, Benjamin Greschbach, Jana Götte, Sangxia Huang, Gunnar Kreitz, Lukáš Poláček, and Oliver Schwarz for the interesting discussions and plans 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 quickly 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.
Chapter 1

Introduction

Today’s society is becoming increasingly dependent on software systems. Failures of such systems can result in the loss of time, money or even in some cases lives. A list of over one-hundred of such failures that became catastrophic can be found on the website [26]. For these reasons, it is becoming necessary to check the correctness of software systems, discover their failures and correct them before the systems are put in operation.

Verification is the task of checking the correctness of software systems. One approach to verification is to manually inspect the code of the software. But this approach is error-prone, time consuming and costly for complex systems.

Testing and formal verification are two alternative methods for verifying the correctness of software systems. Both methods assume access to a so-called specification of the system, which is a description of its desired behaviour. These methods compare the actual behaviour of the system to its specification.

In testing, a so-called test case is generated which consists of an input to the software and its expected behaviour, in accordance to the specification. The input is fed to the software. If the behaviour is as expected, the system is said to have passed the test case, otherwise it has failed. By feeding a set of such test cases to a software, it is tested against its specification. Testing is extensively studied and used in software development. However, it cannot guarantee in general that the behaviour of the system matches its specification. The reason is that, when the set of possible inputs becomes very large or even infinite, it is impossible to confirm that the system behaves correctly in all possible circumstances, and some errors may not be discovered.

The most reliable approach for checking program correctness is formal verification. In this approach, a system is specified by a formal specification and the correctness of the system is proved by showing that the formal specification matches the program, using mathematical techniques. The formal specification is a description of the system which is usually expressed in a mathematical formalism. There are various mathematical formalisms that have been developed to express a variety
CHAPTER 1. INTRODUCTION

of properties of programs, such as, predicate logic and temporal logics.

The field of formal verification is vast, including various methods and automated tools. Most formal verification methods fall into two main categories: proof-based methods, and state-exploration methods. In the proof-based methods, an implementation and its specification are usually expressed as higher-order or first-order logical formulas. Their relationship is regarded as a theorem to be proved in a logical system using axioms and inference rules. Since the full power of mathematics can be used, in these methods programs can be explained with almost any level of detail. But the drawback is that specifications are usually complicated and a high level of expertise is required to specify them.

In the state-exploration methods, the model of programs is restricted to state machines and these are checked against the specification by exhaustively exploring their state-space. The specification is often expressed as a temporal logic formula stating quantified means of time. The main advantage of these methods is that they require less manual work and the checking is mechanical. The main drawback is that the state-space of a program can be very large or even infinite. This problem is known as the state-space explosion problem. One approach to deal with this problem is abstraction. In abstraction, the state-space of the state machine is reduced by hiding some details of the program. When using abstraction, one must show that the abstracted model over-approximates the actual one.

Another, but complementary, approach to attack the state-space explosion problem is compositional verification. This approach divides the verification of the whole system into two tasks: verification of its components, and verification of their composition. The main idea is to verify properties of each individual component, and then relativize the property of the system on the properties of its components, thus avoiding to compute the state-space of the whole system.

Compositional verification has been used in various contexts, such as distributed systems, e.g., [43, 24] and concurrent and real-time systems, e.g., [39, 44]. Also several techniques have been proposed to perform it (e.g., [8, 38, 25] for proof-based verification, and [57, 32, 18, 1, 2, 19] for state-exploration verification). Compositional verification of systems, where system modules are taken as its components, is called modular verification.

The focus of the present work is on the verification of software systems in the presence of variability. The term variability is used for software systems that are dynamically or statically configured by replacing, adapting, or adding new code. Variability can be thought of as the following four typical scenarios: (i) code evolution where programs are configured by replacing or adapting code, or (ii) mobile code where applications are configured by downloading new code to plug-in and provide more functionality, or (iii) open systems where programs are delivered incomplete and are completed later by mobile codes, or (iv) software product line where multiple implementations of modules (or components) exist and these are composed to produce different products. Modularity is the key to provide support for the above scenarios and in general software systems in presence of variability.

In modular verification, each module of a system is specified and verified inde-
pendently (locally). Then the correctness of the whole system, which is specified through a global property, is verified by relativizing the global property on the specifications of the modules (local specifications), rather than on their implementations. Therefore, in modular verification the global correctness is not directly related to the implementations of the modules, but on their specifications. This decoupling of the global properties from the implementation of the modules allows to provide support for variability, e.g., if the implementation of a module is changed, only a local re-verification of that particular module against its specification is needed and not of the whole system.

In this work, our aim is to verify temporal properties. We focus on safety properties of programs. Intuitively, these express that a particular bad event does not happen in the system. One example of such a property is “two processes are never in their critical section at the same time”.

One common approach to modular verification of temporal properties is the Pnueli’s assume-guarantee reasoning [57]. Grumberg and Long proposed a technique allowing to partially automate the assume-guarantee paradigm for finite-state systems [32]. In this study, a so-called maximal model is constructed from each logical specification and used to relativize the global property of a system on the specifications of its components. A maximal model for a specification is the most general model in which the specification holds. It represents the class of all implementations that satisfy the specification.

Later, Gurov, Huisman, and Sprenger proposed the use of maximal models for verification of infinite-state systems, namely programs with procedure calls and recursion. This technique was developed for compositional verification of open systems as a framework called Compositional Verification of Programs with Procedures (CVPP) [34]. In CVPP, to verify open systems, maximal models are constructed from specifications of the unavailable modules and are used to relativize the correctness of the global property. To achieve algorithmic verification, CVPP abstracts away all program data and the models are basically the over-approximated control flow of the method invocation of the programs. The framework focuses on verifying properties of the context-free behaviour of the method invocations. This rather extensive abstraction restricts the class of properties that can be verified by the framework. However, as we show in the next chapters, many interesting properties can still be addressed.

The advantage of the assume-guarantee style of reasoning is that it provides support for variability. However, from the practical point of view, the drawback is that modules specification is done manually and requires considerable time and knowledge. Moreover, maximal model construction is computationally expensive.

Contributions The work in this thesis is developed on top of the CVPP framework by automating and enhancing its usage, evaluating its utility by means of case studies, and explaining its possible extensions. Here, modular verification is achieved at the procedure-level of granularity; the correctness of the temporal global
property of a program is relativized on the correctness of the specifications of its procedures. We believe that writing specifications at the procedure-level is intuitive for developers. However, the restriction to modularity at the procedure-level is not fundamental, and can be relaxed.

The principle contributions of this thesis are the followings.

- Full automation of the usage of the CVPP framework by developing a push-button tool, called ProMoVer. The tool brought the CVPP framework to a stage where people can easily use it. It is easy to use in the sense that it takes as input a Java program annotated with the global property and the local specifications. The tool is accessible through a web-interface [63] for public usage.

- Enhancing the usability by providing a specification extractor and a proof storage and reuse mechanism. These two provide practical solutions for the general limitations of the assume-guarantee reasoning style. The specification extractor extracts candidate specifications from the implementation of modules, thus significantly reducing the effort required for specifying them. Moreover, proof storage and reuse mechanism has considerably improved the efficiency by minimizing the use of maximal model construction and model extraction. Furthermore, the usability is enhanced by providing support for different specification languages.

- Evaluating and identifying application areas that would particularly benefit from the CVPP paradigm of compositional verification. As a particular promising application area, we show how the CVPP framework and ProMoVer can be adapted and used to verify correctness of software product families. For this, a novel hierarchical variability model is introduced, the CVPP modular verification principle is adapted and its soundness is proved. In addition, ProMoVer is adapted by extending its annotation language and verification principle.

- Extending the class of properties that CVPP can handle by including Boolean data. The program model, specification languages and maximal model construction are adapted to include concrete Boolean values. The relevant definitions and theorems of the CVPP framework are adapted and if needed proved. With this extension, it is possible to verify considerably wider class of properties. The usage of the extension is evaluated on a couple of small but meaningful examples.

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


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


The following paper is also published but not used in this thesis.


**Thesis Organization** This thesis is organized as follows. The next chapter explains the basic notions of models and logics. Chapter 3 describes the CVPP compositional verification framework, and Chapter 4 gives an overview of the tool set developed for the CVPP framework. Chapter 5 introduces ProMoVer as a fully automated tool for procedure-modular verification with CVPP, and Chapter 6 explains software families as an application area for CVPP and its fully automated tool support. Chapter 7 describes an extension on CVPP for programs with Boolean data. Chapter 8 draws conclusions and discusses directions of future works.
Chapter 2

Preliminaries

In this chapter, we briefly review the main notions and definitions that are needed to understand the next chapters of this thesis. Here, we provide definitions of structures, transition systems and pushdown automata as models. Then we define two logics for expressing properties, namely Linear-Time Temporal Logic (LTL) and modal $\mu$-calculus. And at the end, review some model checking algorithms and present some background information on compositional verification of temporal properties.

2.1 Models

There are various models for capturing structure and behaviour of programs. In this thesis, we use Kripke structures, transition systems and pushdown automata.

2.1.1 Kripke Structures

Kripke structures are finite-state transition graphs introduced by Saul Kripke [47]. They can be used to capture structure and behaviour 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 $L : S \rightarrow \mathcal{P}(AP)$ is a labelling function.

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

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

where,

$^{1}$\(\mathcal{P}(AP)\) denotes the power set of $AP$. 


• $S \overset{\text{def}}{=} \{s_0, s_1\}$
• $S_0 \overset{\text{def}}{=} \{s_0\}$
• $AP \overset{\text{def}}{=} \{p, r, q\}$
• $\rightarrow \overset{\text{def}}{=} \{(s_0, s_1), (s_1, s_0), (s_1, s_1)\}$
• $L \overset{\text{def}}{=} \{(s_0, \{p, r\}), (s_1, \{q\})\}$.

Figure 2.1 shows a graphical representation of the Kripke structure $M$.

A path of structure $M$ is an infinite sequence of states $\pi = s_0s_1s_2s_3\ldots$ 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 $\pi$, and we denote by $\pi^i$ the $i$-th suffix $s_is_{i+1}s_{i+2}s_{i+3}\ldots$ of $\pi$.

**Example 2.2.** Consider the Kripke structure $M$ from Example 2.1. A path of $M$ is: $\pi = s_0s_1s_2\ldots$ where $\pi(1) = s_1$ and $\pi^1 = s_1s_1\ldots$.

### 2.1.2 Transition Systems

In the literature, various types of transition systems have been proposed. Here, we define labeled transition systems which are transition graphs with labels (actions) on edges (transitions). Labeled transition systems are suitable for modeling the behaviour 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 $T = (S, \text{Act}, \rightarrow)$ where $S$ is a set of states, Act is a set of actions, and $\rightarrow \subseteq S \times \text{Act} \times S$ is a transition relation.

**Example 2.3.** An example of a labeled transition system is defined as follows.

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

where,

• $S \overset{\text{def}}{=} \{s_0, s_1\}$
2.2. Pushdown Automata and Systems

Pushdown automata and pushdown systems are natural ways of modelling the behaviour of programs with procedure calls and recursion (see e.g., [14, 3] for analysis techniques and [27, 15] for applications). These models are variations of automata that make use of a stack to store data. In the next definitions, we formally define pushdown automata and their configurations.

**Definition 2.3 (Pushdown Automata).** A Pushdown Automaton (PDA) is a tuple $P = (Q, \Sigma, \Gamma, \Delta, Q', \bot)$ where $Q$ is a set of control states, $\Sigma$ a finite input alphabet, $\Gamma$ finite stack alphabet, $Q'$ is the initial configuration, and $\Delta \subseteq (Q \times \Gamma) \times \Sigma \times (Q \times \Gamma^*)$ a set of labeled productions (or rewrite rules) of the shape $(q_1, A) \xrightarrow{a} (q_2, \gamma)$.

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

The set of production induces a labeled transition relation of configurations as the least relation which contains the initial configuration and is closed under the prefix rewrite rule: $(q_1, A \cdot \gamma) \xrightarrow{a} (q_2, \gamma \cdot \gamma')$ whenever $(q_1, A) \xrightarrow{a} (q_2, \gamma) \in \Delta$.

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

- $Q \overset{\text{def}}{=} \{q_a, q_b\}$
- $\Sigma \overset{\text{def}}{=} \{a, b\}$
- $\Gamma \overset{\text{def}}{=} \{S, A\}$
Chapter 2. Preliminaries

\[ \Delta \overset{\text{def}}{=} \begin{cases} 
\langle q_a, S \rangle \overset{a}{\to} \langle q_a, A \rangle , \\
\langle q_a, A \rangle \overset{a}{\to} \langle q_a, AA \rangle , \\
\langle q_a, A \rangle \overset{b}{\to} \langle q_b, \epsilon \rangle , \\
\langle q_b, A \rangle \overset{b}{\to} \langle q_b, \epsilon \rangle 
\end{cases} \]

- \( Q' \overset{\text{def}}{=} q_a \)
- \( \bot \overset{\text{def}}{=} S \)

Figure 2.3 shows a graphical representation of the pushdown automaton \( P \). In the figure, the labeled productions are depicted by transitions such that a labeled production \( \langle q_1, A \rangle \overset{a}{\to} \langle q_2, \gamma \rangle \) is depicted by a transition from state \( q_1 \) to state \( q_2 \) with the label \( a; A/\gamma \).

\[ \text{Figure 2.3: Pushdown automaton } P \]

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

### 2.3 Temporal Logics

Temporal logic formulas are used to express behaviour of systems by quantified means of time. Time is discrete and extends infinitely into the future. The formulas express when (w.r.t., time) a statement is true. Various temporal logics have been introduced and used for verification. Here, we define Linear-Time Temporal Logic (LTL) introduced by Pnueli [56] and Modal \( \mu \)-calculus introduced by Kozen [46].

#### 2.3.1 Linear-Time Temporal Logic

Linear-Time Temporal Logic (LTL) [56] is the most commonly used temporal logic. It is defined as follows.

**Definition 2.6 (Syntax of LTL).** The formulas of LTL are inductively defined by:
2.3. TEMPORAL LOGICS

\[ \Phi ::= p \mid \neg \Phi \mid \Phi \land \Psi \mid \Phi \lor \Psi \mid X \Phi \mid G \Phi \mid F \Phi \mid \Phi \cup \Psi \]

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

In the above syntax, the symbols \( \land, \lor, \neg \) are the usual Boolean connectives. The symbols \( X, G, F, \cup \), and \( W \) are temporal operators, \( X \Phi \) means \( \Phi \) holds in the next state, \( G \Phi \) means \( \Phi \) holds in all future states including the current state, \( F \Phi \) means \( \Phi \) is true in some future state, and \( \Phi \cup \Psi \) means \( \Phi \) remains true until a state is reached where \( \Psi \) is true.

Next, we give the precise semantics of LTL.

**Definition 2.7 (Semantics of Syntax of Linear-Time Temporal Logic).** Let \( M = (S, S_0, AP, \rightarrow, L) \) be a model, and let \( \pi \) be a path of \( M \).

\[
\begin{align*}
\pi \models^M p & \iff p \in L(\pi(0)) \\
\pi \models^M \neg \Phi & \iff \text{not } \pi \models^M \Phi \\
\pi \models^M \Phi \land \Psi & \iff \pi \models^M \Phi \text{ and } \pi \models^M \Psi \\
\pi \models^M \Phi \lor \Psi & \iff \pi \models^M \Phi \text{ or } \pi \models^M \Psi \\
\pi \models^M X \Phi & \iff \pi^1 \models^M \Phi \\
\pi \models^M G \Phi & \iff \forall i \geq 0. \pi^i \models^M \Phi \\
\pi \models^M F \Phi & \iff \exists i \geq 0. \pi^i \models^M \Phi \\
\pi \models^M \Phi \cup \Psi & \iff \exists i \geq 0. (\pi^i \models^M \Psi \land \forall j < i. \pi^j \models^M \Phi)
\end{align*}
\]

Some authors have defined and used some other LTL operators. In this thesis, we will use operator \( W \) which is the weak version of \( U \) operator, such that \( \Phi W \Psi \) means \( \Phi \) remains true until a state is reached where \( \Psi \) is true or \( \Phi \) remains true in all states of the path. The formal semantics of operator \( W \) can be given through operators \( U \) and \( G \) as follows.

\[
\Phi W \Psi = (\Phi U \Psi) \lor G \Phi
\]

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

**Example 2.5.** The property “\( r \) and \( q \) are never true at the same time” can be expressed by the following LTL formula.

\[
G \models (r \land q)
\]

This property holds for the Kripke structure \( M \) from the Example 2.1.
2.3.2 Modal $\mu$-Calculus

Modal $\mu$-calculus [46] is an extension of Hennessy-Milner logic [37] 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 $\mu$-Calculus).** The formulas of the modal $\mu$-calculus are defined as follows [66].

$$\Phi ::= \texttt{tt} \mid \texttt{ff} \mid Z \mid \Phi \lor \Phi \mid \Phi \land \Phi \mid \langle \alpha \rangle \Phi \mid [\alpha] \Phi \mid \mu Z. \Phi \mid \nu Z. \Phi$$

where $\alpha$ 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 $\mu$-Calculus).** For a labelled transition system $T = (S, Act, \rightarrow)$, let $V : \text{Var} \rightarrow \mathcal{P}(S)$ be a valuation. The semantics of the modal $\mu$-calculus for all $s \in S$ is defined as follows.

$$\| \texttt{tt} \|_V^T \overset{def}{=} S$$
$$\| \texttt{ff} \|_V^T \overset{def}{=} \emptyset$$
$$\| Z \|_V^T \overset{def}{=} V(Z)$$
$$\| \Phi \lor \Psi \|_V^T \overset{def}{=} \| \Phi \|_V^T \cup \| \Psi \|_V^T$$
$$\| \Phi \land \Psi \|_V^T \overset{def}{=} \| \Phi \|_V^T \cap \| \Psi \|_V^T$$
$$\| \langle \alpha \rangle \Phi \|_V^T \overset{def}{=} \| \langle \alpha \rangle \|_V^T (\| \Phi \|_V)$$
$$\| [\alpha] \Phi \|_V^T \overset{def}{=} \bigcup \{ S' \subseteq S \mid S' \supseteq \| \Phi \|_V[S'/Z] \}$$
$$\| \mu Z. \Phi \|_V^T \overset{def}{=} \bigcap \{ S' \subseteq S \mid S' \supseteq \| \Phi \|_V[S'/Z] \}$$
$$\| \nu Z. \Phi \|_V^T \overset{def}{=} \bigcup \{ S' \subseteq S \mid S' \subseteq \| \Phi \|_V[S'/Z] \}$$

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

$$\| \langle \alpha \rangle \|_V^T (S') \overset{def}{=} \{ s \in S \mid \exists s' \in S. (s \xrightarrow{\alpha} s' \land s' \in S') \}$$
$$\| [\alpha] \|_V^T (S') \overset{def}{=} \{ s \in S \mid \forall s' \in S. (s \not\xrightarrow{\alpha} s' \Rightarrow s' \in S') \}$$

and

$$(V[S'/Z})(Y) = \begin{cases} S' & \text{if } Y = Z \\ V(Y) & \text{otherwise} \end{cases}$$

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

- $\| (\sigma Z. \Phi)^\alpha \|_V^T$...
2.4. MODEL CHECKING

\[ (\mu Z. \Phi)^0 \] \( T \) def \( = S_T \)

\[ (\mu Z. \Phi)^{\alpha + 1} \] \( T \) def \( = \| \Phi \|_{V_{((\mu Z. \Phi)^{\alpha})/Z}}^{T} \]

\[ (\nu Z. \Phi)^{\lambda} \] def \( = \bigcup \{ \| (\nu Z. \Phi)^{\alpha} \|_{V} | \alpha \leq \lambda \} \)

\[ (\nu Z. \Phi)^{\lambda} \] def \( = \bigcap \{ \| (\nu Z. \Phi)^{\alpha} \|_{V} | \alpha \leq \lambda \} \)

**Example 2.6.** The property “exactly after occurrence of action \( a \), another action \( a \) cannot happen” for a system with actions \( a \) and \( b \), can be formalized by the following \( \mu \)-calculus formula.

\[ \nu Z. [b] Z \land [a] ([a] ff \land [b] Z) \]

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

2.4 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. Model checking algorithms are generally divided into two categories, finite-state and infinite-state. For model checking of finite-state models, several algorithms have been proposed, e.g., explicit model checking by Lichtenstein and Pnueli [49], symbolic model checking by Grumberg and Long [16] and by McMillan [52], and bounded model checking by Biere et al. [12]. Several algorithms have also been proposed for infinite-state model checking, e.g., model checking of pushdown systems against LTL formulas [62], (see [14] for a survey).

In the present work, we do not develop model checking algorithms. However we employ model checking algorithms for model checking of finite-state and infinite-state models against \( \mu \)-calculus and LTL formulas.

2.5 Background on Compositional Verification of Temporal Properties

Compositional verification was introduced to avoid the state explosion problem by dividing the verification of the whole system into verification of its components and their composition. The main idea is to verify properties of each individual component, and then relativize a property of the system on the properties of its components and thus avoiding to compute the state space of the whole system.

Compositional verification has been studied in the context of compositional model checking as in [17, 18], reactive models as in [1, 2] and assume-guarantee style as in [57, 32] (see [55] for a survey).

Here, we focus on assume-guarantee approach. This approach was first proposed in the context of temporal logic by Pnueli [57]. In Pnueli’s system, one works with
triples of the form $\langle \Phi \rangle M \langle \Psi \rangle$ which is read as “assuming the environment of $M$ satisfies $\Phi$, component $M$, in this environment, guarantees to satisfy $\Psi$”. Then a typical chain of reasoning would be as follows.

$$
\frac{\langle \Phi \rangle M \langle \Psi \rangle}{\langle \text{true} \rangle M' \langle \Phi \rangle}
\frac{\langle \text{true} \rangle M' \langle \Phi \rangle}{\langle \text{true} \rangle M' | M \langle \Psi \rangle}
$$

The above principle states that if $M'$ satisfies $\Phi$, and if the environment of $M$ satisfies $\Phi$, then $M$ satisfies $\Psi$, then the composition of $M'$ and $M$ will satisfy $\Psi$.

The obvious advantage of verification by the above approach is that we never have to examine the composite state-space of $M' | M$. Nevertheless, this principle brings us the possibility of doing hierarchical verification.

The down side is that the user must determine an appropriate $\Phi$ which requires knowledge of the system behaviour. However, automation of the other parts of the above compositional verification principle has been studied by Grumberg and Long in [32] for finite-state systems. They proposed a simulation preorder (denoted $\preceq$) on the finite-state models that preserves satisfaction of temporal logic formulas of a subset of Computation Time Logic (CTL) called $\forall$CTL. This simulation preorder has two important following properties.

- It is a (simulation) preorder on structures that preserves satisfaction of the formulas of the logic, i.e., for models $M$ and $Q$ and formula $\Phi$,

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

- The preorder is preserved by composition, i.e., for models $M$, $M'$, $Q$ and $Q'$,

  $$(M \preceq M') \land (Q \preceq Q') \Rightarrow (M | Q) \preceq (M' | Q')$$

Moreover, the following condition is necessary. For all $\forall$CTL formulas $\Phi$ there exists a process $\text{Max}(\Phi)$ such that

$$M \preceq \text{Max}(\Phi) \iff M \models \Phi$$

Here, the key point is that any $\forall$CTL formula $\Phi$ has a maximal model $\text{Max}(\Phi)$ which can be obtained by a tableau construction. By this tableau construction, formula $\Phi$ is associated with a structure which is called maximal model of formula $\Phi$. By this, Grumberg and Long proposed the following sound compositional verification principle.

$$
\frac{M' \models \Phi}{\frac{\text{Max}(\Phi) | M \models \Psi}}
$$

The rule reduces checking $M | M' \models \Psi$ to the following steps: 1) decomposition of the global property $\Psi$ to the local property $\Phi$ on component $M'$, 2) the check that $M'$ satisfies $\Phi$, 3) construction of a maximal model $\text{Max}(\Phi)$ for $\Phi$, and 4) the check that $\text{Max}(\Phi) | M$ satisfies $\Psi$. 

Chapter 3

A Verification Framework

In this chapter of the thesis, we describe the theoretical underpinnings of the compositional verification of programs with procedures (CVPP) framework, developed by Gurov, Huisman and Sprenger and presented in [34]. As mentioned in the introduction, CVPP is a compositional verification technique to verify temporal safety properties of infinite-state systems. Intuitively, the CVPP framework for compositional verification relativizes the global correctness of a property of a system on the local correctness of its components. The local correctness is defined by means of the specifications of components (local specifications) provided by the users. These abstractly express the safety property that should hold by the corresponding component.

In the CVPP framework, flow graphs are serving as the program model. These are finite-state models which can be extracted from the implementation of components or can be constructed from their local specifications. To relativize the global correctness on the local specifications of components, usually a saturated flow graph is constructed from the local specification which is called maximal flow graph. 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 verification.

For the flow graph extraction from an implementation of a program, the technique relies partially on external tools and approaches, such as SawJa [41]. Whereas, for maximal flow graphs, a theoretical framework and a tool set have been developed. The theoretical framework will be discussed in this chapter and the tool set will be briefly described in Chapter 4.

In the CVPP framework, the program analysis is algorithmic because all the program data is abstracted away and programs will be analyzed only for the safety temporal properties of the over-approximated control flow of the program. Although this sounds like a severe restriction, still many useful properties can be expressed at this level of abstraction. These include platform-specific security properties and application-specific properties such as:
within atomic transactions, there is no call to non-atomic methods such as non-atomic array copy;

• a given method that changes certain sensitive data is only called from within another dedicated authentication method, i.e., unauthorized access is not possible;

• before program state is being dumped into memory, a serialization method is called to arrange the state;

• in a voting system, candidate selection has to be finished, before the vote can be confirmed;

• in a door access control system, the password has to be checked before the door is unlocked, and the password can only be changed if the door is unlocked.

In the CVPP framework, properties of components are analyzed with respect to the finite-state structure of the component’s code (i.e., flow graph), called component structure, and the infinite-state behaviour induced from the structure, called component behaviour. Properties of the component structure and behaviour are called structural and behavioural properties, respectively. Local and global properties can be either structural or behavioural. If a property is structural, it can be checked against a flow graph by standard finite-state model checking. If a property is behavioural then the flow graph should be translated into a behavioural format representing as an infinite-state pushdown automaton model, it is checked against the property by standard infinite-state model checking for pushdown automata.

In the next sections, we will formally define 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.

3.1 Model and Logic

We begin by formally defining the general notion of model and logics for specifying properties.

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 $\lambda : S \rightarrow \mathcal{P}(A)$ 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 $S = (\mathcal{M}, E)$ is defined by $\mathcal{R}(S) = (\mathcal{M}', E)$, where $\mathcal{M}'$ is obtained from $\mathcal{M}$ by deleting all states and transitions not reachable from any entry state in $E$.

The definition of simulation is standard.
3.1. MODEL AND LOGIC

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 $s \preceq t$, 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 $(\mathcal{M}_1, E_1) \preceq (\mathcal{M}_2, E_2)$ 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 $\mathcal{S}_1 \preceq \mathcal{S}_2$ and $\mathcal{S}_2 \preceq \mathcal{S}_1$. Disjoint union is extended to initialized models $(\mathcal{M}_1, E_1) \uplus (\mathcal{M}_2, E_2) = (\mathcal{M}_1 \uplus \mathcal{M}_2, E_1 \uplus E_2)$ and the following theorem shows that simulation is preserved by disjoint union.

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

For the proof we refer the readers to [34].

As property specification language we use the safety fragment of the modal $\mu$-calculus [66] with the box and $\nu$ operators. The syntax and semantics of the modal $\mu$-calculus is explained in Section 2.3.2. Here the specification language is parameterized on a set of atomic propositions $A$ and labels $L$.

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

$$\phi ::= p \mid \neg p \mid X \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid [a] \phi \mid \nu X. \phi$$

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

Alternative to simulation logic defined above, one can use modal equation systems where instead of the $\nu$ operator of $\mu$-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:

$$\phi ::= p \mid \neg p \mid X \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid [a] \phi$$
CHAPTER 3. A VERIFICATION FRAMEWORK

where \( p \in A \), \( a \in L \) and \( X \) ranges over 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 \( \Phi_i \) is a formula of basic simulation logic over \( L \) and \( A \). The set of variables occurring in \( \Sigma \) is partitioned into the set of bound variables, defined by \( bv(\Sigma) = \{X_i | i \in I\} \), and the set of free variables \( fv(\Sigma) \).

Using the definitions of basic simulation logic and modal equation systems, the formulas of simulation logic is defined by \( \Phi[\Sigma] \) over \( L \) and \( A \), where \( \Phi \) is a formula of basic simulation logic and \( \Sigma \) is a modal equation system. These formulas are equally expressive as the modal \( \mu \)-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 [11, 7]. The translation in the other direction is straightforward and it 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 behaviour induced from it.

3.2.1 Program Structure

The 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 procedure of the program. Let \( \text{Meth} \) be a countably infinite set of method names. A method graph is an instance of the general notion of initialized model.

**Definition 3.7 (Method graph).** A method graph for method \( m \in \text{Meth} \) over a set \( M \subseteq \text{Meth} \) of method names is an initialized model \( (M_m,E_m) \) where \( M_m = (V_m,L_m,\rightarrow_m,A_m,\lambda_m) \) 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.

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 we generate 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 \( I^+, I^- \subseteq \text{Meth} \) are finite sets of names of provided and required methods, respectively. The composition of two interfaces \( I_1 = (I^+,I^-) \) and \( I_2 = (I^{+,1},I^{-,1}) \)
3.2. PROGRAM MODEL

\((I^+, I^-)\) is defined by \(I_1 \cup I_2 = (I_1^+ \cup I_2^+, I_1^- \cup I_2^-)\). 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.

\[\text{Example 3.1.}\] Figure 3.1 shows a Java program and its flow graph. Its interface is \(\{\text{even}, \text{odd}\}, \{\text{even}, \text{odd}\}\), thus the flow graph is closed. It consists of two method graphs, for method \text{even} and method \text{odd}, respectively. Entry nodes are depicted as usual by incoming edges without source.

\[\text{Example 3.2.}\] Consider the closed flow graph in Example 3.1. The property “if the program execution starts in method \text{even}, the first call is to method \text{odd} and after returning no other method can be called” is formalized by the following structural formula. \(\text{even} \Rightarrow \nu X. [\text{even}]f \land [z].X \land [\text{odd}]) \phi\), where \(\phi\) is: \(\nu Y.[\text{even}]f \land [\text{odd}])f \land [z].Y\).

3.2.2 Program Behaviour

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

**Behaviour of Closed Flow Graphs**

We define the behaviour of a flow graph as a labeled transition system (LTS). We use transition label \(\tau\) for internal transfer of control, \(m_1 \text{ call } m_2\) for the invocation of method \(m_2\) by method \(m_1\) when method \(m_2\) is provided by the program, \(m_2 \text{ ret } m_1\) for the corresponding return from the call.
Definition 3.9 (Behaviour). Let $G = (M, E) : (I^+, I^-)$ be a flow graph such that $M = (V, L, \rightarrow, A, \lambda)$. The behaviour of $G$ is defined as initialized model $\text{b}(G) = (M_b, E_b)$, where $M_b = (S_b, L_b, \rightarrow_b, A_b, \lambda_b)$, such that $S_b = V \times V^*$, i.e., states (or configurations) are pairs of control points $v$ and stacks $\sigma$, $L_b = \{m_1 k m_2 | k \in \{\text{call, ret}\} \cup \{\tau\}, A_b = A, \lambda_b((v, \sigma)) = \lambda(v), \text{and} \rightarrow_b \subseteq S_b \times L_b \times S_b$. The set of initial configurations is defined by $E_b = E \times \{\varepsilon\}$, where $\varepsilon$ 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) behaviour, from an initial to a final configuration, is:

$$
(v_0, \varepsilon) \xrightarrow{\text{transfer}} (v_1, \varepsilon) \xrightarrow{\text{call}} (v_2, v'_1 \cdot \sigma) \xrightarrow{\text{ret}} (v_6, v_3) \xrightarrow{\text{ret}} (v_3, \varepsilon)
$$

An alternative way to express flow graph behaviour is to use pushdown automata (PDA) or pushdown systems (PDS). This can be exploited by using pushdown automata/systems model checking for verifying behavioural properties.

Example 3.4. Consider the closed flow graph in Example 3.3. The property structurally formalized in Example 3.2 is formalized by the following behavioural formula.

$$
\text{even} \Rightarrow \nu X. [\text{even call even}] \mathbf{ff} \land [\tau] X \land [\text{even call odd}][\text{odd ret even}] \phi, \text{where } \phi =: \nu Y. [\text{even call even}] \mathbf{ff} \land [\text{even call odd}] \mathbf{ff} \land [\tau] Y.
$$

The instantiation of the definition of simulation for behaviour $\preceq_b$ is:

$$
G_1 \preceq_b G_2 \Leftrightarrow \text{b}(G_1) \preceq \text{b}(G_2).
$$

Also it is shown that the structural simulation implies the behavioural simulation:

$$
G_1 \preceq_s G_2 \Rightarrow G_1 \preceq_b G_2
$$

For the proof we refer the reader to [34].

Behaviour 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 the Definition 3.9 for
open flow graphs where some required methods are external by adding the following labels to the set \( L_b \).

- \( \text{call!} \) is used when an internal method calls an external one,
- \( \text{call?} \) is used when an external method calls an internal one,
- \( \text{ret!} \) is used when an internal method returns to an external one,
- \( \text{ret?} \) is used when an external method returns to an internal one.

Having the above labels the set of rules should also be extended by the following rules.

\[
\begin{align*}
\text{call!} & \quad (v_1, \sigma) \xrightarrow{m_1 \text{call!} m_2} (m_2, v'_1 \cdot \sigma) & \text{if } m_1 \in I^+, m_2 \notin I^+, \\
\text{call?} & \quad (m_2, \sigma) \xrightarrow{m_3 \text{call?} m_1} (v, m_3 \cdot m_2 \cdot \sigma) & \text{if } m_1 \in I^+, m_2, m_3 \notin I^+, \\
\text{ret!} & \quad (v, m_3 \cdot m_2 \cdot \sigma) \xrightarrow{m_1 \text{ret!} m_3} (m_2, \sigma) & \text{if } m_1 \in I^+, m_2, m_3 \notin I^+, \\
\text{ret?} & \quad (m_2, v_1 \cdot \sigma) \xrightarrow{m_2 \text{ret?} m_3} (v_1, \sigma) & \text{if } m_1 \in I^+, m_2 \notin I^+, v_1 \models m_1
\end{align*}
\]

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

\[
(v_0, \bot) \xrightarrow{\tau} (v_1, \varepsilon) \xrightarrow{\text{even call even}} (v_2, \varepsilon) \xrightarrow{\text{odd call! odd}} (v_3, \text{odd \cdot odd \cdot v_3}) \xrightarrow{\tau} (v_0, \text{odd \cdot odd \cdot v_3}) \xrightarrow{\text{even call! odd}} (v_1, \text{odd \cdot odd \cdot v_3}) \xrightarrow{\text{odd ret? even}} (v_3, \varepsilon).
\]

**Example 3.6.** Consider the open flow graph in Example 3.5. The property “the first call is to external method \texttt{odd} and no other method can be called after returning from the call” is formalized by the following open behavioural formula.

\[
\nu X. [\text{even call even}] \mathbf{ff} \land [\tau] X \land [\text{even call! odd}] [\text{odd ret? even}] \phi,
\]

where \( \phi \) is:

\[
\nu Y. [\text{even call even}] \mathbf{ff} \land [\text{even call! odd}] \mathbf{ff} \land [\tau] Y.
\]

The next thing we need for the compositional verification to work is the maximal flow graphs. In the next section, we will introduce them and briefly explain their construction process.

### 3.3 Maximal Flow Graph

Intuitively, a maximal model for a property \( \phi \) is a model that satisfies \( \phi \) and simulates all models satisfying \( \phi \). The use of maximal models was first proposed by
Grumberg and Long in [32]. As explained in Section 2.5, they proposed a maximal model construction for a subset of Computational Tree Logic (CTL), called $\forall$CTL. In this section, however, we describe a maximal model construction for models defined in Definition 3.1 and logical formulas defined in Definition 3.4 which was proposed by Gurov, Huisman, and Sprenger in [34]. Since the program structure and behaviour are instances of models, the maximal model construction can be used for both structures and behaviours. In [34], the existence and uniqueness of maximal models constructed from structural properties are proved. However, in general, these maximal models are not legal flow graphs. For structural properties, this problem can be solved for a fixed interface by a so-called characteristic formula which is a formula constructed from the interface and constrains the constructed maximal models to legal flow graphs. Basically the characteristic formula for interface $I$ precisely defines all flow graphs with interface $I$. Concretely, if $\sigma_{I_m}$ is the characteristic formula of component $m$ and $\sigma_m$ is the property of $m$, we construct the maximal model of $\sigma_{I_m} \land \sigma_m$ which is a legal flow graph with interface $I_m$ and simulates all flow graphs satisfying the property $\sigma_m$ with interface $I_m$. We will define characteristic formula and the process of construction maximal flow graphs from structural formula later in this section.

For behavioural properties however, there is no such way to characterize all models that are loyal to the definition of the flow graph behaviour. The intuitive reason is that in the behavioural logic context-free properties cannot be expressed. Moreover, the maximal flow graph behaviour is in general infinite-state and therefore, the maximal model construction for a behavioural property has to return a finite representation of this behaviour. Gurov et al. chose flow graph structure for this representation. By this, the problem reduces to finding a flow graph structure that satisfies the given behavioural property and behaviourally simulates all other flow graphs satisfying the property. However, in general such a maximal flow graph is not unique. Therefore, they provide a translation of behavioural properties into a set of structural properties that can be used for maximal flow graph construction.

In this section, we first explain the construction of maximal models, then we define the characteristic formula to be used for constructing maximal flow graphs for structures. The translation of behavioural properties into a set of structural ones is explained in [33].

The materials of this section are adapted from [34] where the details and all proofs can be found.

### 3.3.1 Maximal Model Construction

To start with the construction of maximal models we need two auxiliary functions $\theta$ and $\chi$ which form a Galois connection between finite specifications and formulas in simulation logic. The function $\chi$ translates each finite specification into formula, and the function $\theta$ translates formulas into (finite) specifications. The function $\theta$ is defined on formulas on a so-called simulation normal form (SNF). Here we only define the auxiliary functions $\chi$ and $\theta$, and SNF. In [34], it is proved that every
3.3. MAXIMAL FLOW GRAPH

Simulation logic formula can be transformed to a SNF formula and an algorithm for the transformation is given.

**Definition 3.10** ($\chi$). By $\chi$ each finite specification $(M, E)$ is translated into its characteristic formula $\chi(M, E) = \phi_E[\Sigma_M]$, where $\Sigma_M$ is defined by an equation below.

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

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

![Figure 3.2: Specification example](image)

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

\[
\Sigma = \begin{bmatrix}
X_{s_1} &=& [a][\text{ff} \land [\varepsilon]X_{s_2} \land p \land q] \\
X_{s_2} &=& [a][X_{s_1} \lor X_{s_3}] \land [\varepsilon][\text{ff} \land p \land \neg q] \\
X_{s_3} &=& [a][X_{s_1} \lor [\varepsilon]X_{s_2} \land \neg p \land \neg q]
\end{bmatrix}.
\]

**Theorem 3.11.** Let $S_1, S_2$ be two models and suppose $S_2$ is finite. Then $S_1 \leq S_2$ if and only if $S_1 \models \chi(S_2)$.

**Definition 3.12 (Simulation normal form).** A formula $\phi[\Sigma]$ is of simulation logic over $L$ and $A$ is in simulation normal form (SNF) if $\phi$ has the form $\bigvee_{\chi} \chi$ for some finite set $\chi \subseteq \text{bv}(\Sigma)$ and all equations of $\Sigma$ have the following state normal form

\[ X = \bigwedge_{a \in L} [a] \bigvee_{Y_{X,a} \subseteq \text{bv}(\Sigma)} \bigwedge_{p \in B_X} p \land \bigwedge_{q \notin B_X} \neg q \]

where each $Y_{X,a} \subseteq \text{bv}(\Sigma)$ is a finite set of variables and $B_X \subseteq A$ is a set of atomic propositions.
Definition 3.13 ($\theta$). By $\theta$ each formula in SNF is translated to (finite) specification. From formula $\phi(\bigvee X)[\Sigma]$ over $L$ and $A$ in SNF we derive the specification

$$\theta(\phi(\bigvee X)[\Sigma]) = ((S, L, \to, A, \lambda), E)$$

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

Theorem 3.14 (Maximal model theorem). For $\phi$ in SNF, we have $S \leq \theta(\phi)$ if and only if $S \models \phi$.

Proof. Follows from Theorem 3.11 and that $X$ and $\theta$ are each others inverse up to equivalence. For complete proof see [34].

3.3.2 Maximal Flow Graph Construction

As explained above, to construct maximal flow graphs from structural property $\phi$ 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 this characteristic formula as a simulation logic structural formula.

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

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

where, $P_m = m \land \bigwedge_{m' \in I^+ \setminus \{m\}} \lnot m'$

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

$$\sigma_{I_m} = \bigvee_{m \in I^+} X_m$$

$$\Sigma_I = \{X_m = [I^-, \epsilon]X_m \land P_m \mid m \in I^+\}$$

$$p_m = m \land \bigwedge \{\lnot m' \mid m' \in I^+, m' \neq m\}$$

Theorem 3.15. Let $I$ be an interface. For any initialized model $S = (\mathcal{M}, E)$ over labels $L = I^- \cup \{\epsilon\}$ and atomic propositions $A = I^+ \cup \{r\}$ we have

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

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

If the maximal model is constructed for the conjunction of a formula $\phi$ and characteristic formula for interface $I$, it is a maximal flow graph for $I$ and $\phi$. 
3.4 Compositional Verification

Based on the maximal model theorem, the following sound and complete compositional verification principle for models can be obtained.

To show that $M_1 \uplus M_2 \models \phi$ it suffices to show that $M_1 \models \psi$ where $\psi$ is the local property of $M_1$ and $\text{Max}(\psi, I) \uplus M_2 \models \phi$ where $\text{Max}(\psi, I)$ is the maximal flow graph constructed from property $\psi$ and interface $I$.

Formally, the above compositional verification for structural property $\phi$ of composition of flow graphs $G_1$ and $G_2$ and structural property $\psi$ and interface $I$ of flow graph $G_1$ is defined by the following principle.

\[
\frac{G_1 \models_s \psi \quad \text{Max}(\psi, I) \uplus G_2 \models_s \phi}{G_1 \uplus G_2 \models_s \phi}
\] (3.1)

In case of behavioural property $\phi$ and $\psi$ the principle would be as follows.

\[
\begin{align*}
G_1 \models_b \psi & \quad \biguplus_{\chi \in \Pi(\psi)} \text{Max}(\chi, I) \uplus G_2 \models_b \phi \\
\frac{}{G_1 \uplus G_2 \models_b \phi}
\end{align*}
\] (3.2)

Where $\Pi(\psi)$ is the set of structural formula translated from $\psi$.

As proved in [34], the 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 [34], where local structural properties are combined with global behavioural ones.

The proof rules presented above are flexible to be used for reasoning about combination of concrete components (i.e., given through their implementation) and abstract components (i.e., given through their specification), both at the structural and the behavioural levels [42]. In Chapter 4 we show how these proof rules can be instantiated for different verification scenarios such as verification of open systems or modular verification. Also in Chapter 5 we show an instantiation of the proof rules that is used for procedure-modular verification.

3.5 Private Method Abstraction

Often private methods are viewed as a way of implementing public ones, i.e., as an implementation detail that specifications should abstract from. To support reasoning on the level of public methods, Gurov et al. propose a transformation based on inlining the private methods into the public ones. The transformation is sound (i.e., over-approximates the behaviour) in general, and complete for last-call recursive programs and $\tau$-insensitive properties.

The algorithm works as follows. Given a set of public methods $M$ of a flow graph $G$, the algorithm transforms $G$ such that every call to a private method $m$
in methods in $M$ will be inlined by $m$’s method graph (see [34] for details). The recursive calls to private methods are not inlined, but create loops in the resulting graph and external methods are treated as public methods. By this, only the method graphs of public methods of a flow graph are used for the verification and this make a significant optimization for the maximal flow graph construction.
Chapter 4

A Verification Tool Set

The CVPP compositional verification technique based on maximal flow graph construction explained in Chapter 3 is implemented in the form of a tool set that can be used to achieve verification results in various scenarios. In this chapter, we will give an overview of the tool set. Then we will explain the scenarios that the tool set can be used for. The content of this chapter is adapted from [42].

4.1 Overview of the Tool Set

Figure 4.1 presents an overview of the CVPP tool set. In the figure, the rounded boxes represent data formats, squared boxes denote tool components, and the dashed boxes is used to show the external tools or data formats.

4.1.1 Data Formats

The CVPP tool set includes the following data formats.

- **Model**: the representation of the program in the CVPP framework. The model of a program is its flow graph either extracted from the implementation of
the program or constructed through a maximal flow graph construction. The textual representation of the model 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, the `node` keyword defines a node of the flow graph, followed by a list of atomic propositions that hold in the node, while the `edge` keyword defines a transition by starting node, target node, and the transition label, respectively. The atomic propositions `entry` and `ret` specify entry and return nodes, respectively, `meth` is to distinguish method names from `entry` and `ret` atomic propositions, and label `eps` is the textual representation of $\varepsilon$.

- **Formula**: the property representation. Formulas can specify structural and behavioural simulation logic, modal equation systems and linear temporal logic (LTL) formats. 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]\neg f f \land (\[eps\]X1) \land \[odd]\nu X2. (\[even]\neg f f \land (\[odd]\neg f f \land (\[eps\]X2))),
  ```

  Behavioural formula in Example 3.4:

  ```
  meth(even) =>
  ```
4.1. OVERVIEW OF THE TOOL SET

nu X1. (((\text{even call even}) \land (\tau X1)) \land ((\text{even call odd}) \land \text{odd ret even})) \land ((\text{even call even}) \land ((\text{even call odd}) \land (\tau X2)))

- **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 tool set 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

- **Analyzer**: the tool for extracting flow graphs from program code. Currently, Analyzer extracts flow graphs of Java programs from Java bytecode. It is based on a static analysis library for Java programs called Sawja [41]. It parses Java bytecode produced from the source code of the program and disregards all stack operations and program data in the bytecode, thus extracting only the flow graph of the program\(^1\) [5].

- **Graph**: the collection of algorithms for transformations of the program model representation. By this tool, the program models are composed together, PDA and PDS behaviour and CCS format of the program model is generated and method graphs of private methods are inlined into method graphs of public ones. The later case is called **Graph Inliner** and explained in Section 3.5.

- **Formula**: the tool for translation, transformation and simplification of the property representation. By this tool the behavioural formulas are translated to sets of structural formulas, LTL formulas are converted to simulation logic formulas, simulation logic formulas are translated to CWB \(\mu\)-calculus formulas, and simulation logic formulas are simplified. The behavioural to structural translation is the main functionality of the tool were a behavioural formula is translated into a set of structural ones to be used in the maximal model construction (see Chapter 3).

- **Maximal Model**: to construct maximal flow graphs. This tool is constructing a maximal flow graph from a structural formula and interface of a

\(^1\)At the moment, CVPP only supports Java programs because the Analyzer 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.
component by the approach explained in Section 3.3. As input property language this tool accepts modal equation systems. Hence, for using this tool, the formula has to be transformed to a modal equation system by Formula.

- Model Checkers: the external model checkers for the local and global model checking tasks. Moped [45] is used for the global model checking task to perform model checking of LTL formula on a PDS representation of the behaviour of the program. On the other hand, Edinburgh Concurrency Workbench (CWB) [21] is used for the local model checking of μ-calculus formulas on a CCS format of the flow graphs.

4.2 Typical Verification Scenarios

The components of the CVPP tool set can be used for different verification scenarios. In this section we explain three main verification scenarios that the CVPP tool set supports. In the next chapter, we describe ProMoVer which is a tool for executing one of the most useful verification scenarios (namely procedure-modular verification) with the CVPP tool set enhanced with a number of features to facilitate easy usage.

4.2.1 Verification of Open Systems

The CVPP framework and consequently its tool set are designed for modular verification of open systems. Open systems are systems 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 known or not stable at the verification time.

In this situation, the verification of a global property of the open system should be performed by relativizing the global property on the specifications of the abstract components. Thus the verification rule 3.2 can be applied. To achieve this, the following tasks for global and local checks should be accomplished.

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

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

The concrete tasks by using the CVPP tool set 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 that is described below.
4.2. TYPICAL VERIFICATION SCENARIOS

Global Check:
(a) extract the flow graph from the code of the concrete components by Analyzer tool, and use Graph Inliner to generate the publicly visible flow graph,
(b) for each of the abstract components, construct maximal flow graph of the specification of the component 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 below.

4.2.2 Modular Verification
In modular verification, the goal is to verify each module (component) of the system, independently and then to relativize the correctness of the global property on basis the local correctness. In the cvpp setting, modular verification is an instantiation of the verification of open systems explained above, i.e., 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 components. This eliminates task (a) and simplifies task (c) of the Global Check of the open system scenario, resulting in the following steps:

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

Global Check:
(a) for each component, construct maximal flow graph of the specification of the component 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 below.

4.2.3 Non-Compositional Verification
Both verification of open systems and modular verification give rise to some non-compositional verification tasks. In fact, cvpp can be used for non-compositional verification as well. This scenario is useful for verifying behavioural properties because even in the absence of data, the verification of behavioural properties of procedural programs is infinite-state due to unbounded recursion. While, the verification of structural properties is finite-state and therefore in cvpp, by applying the behavioural to structural property translator, we can translate a behavioural property to a set of structural properties and then perform the finite-state verification. This is done as follows:
1. extract the flow graph from the code of the concrete components by Analyzer tool, and use Graph Inliner to generate the publicly visible flow graph,

2. convert the flow graph to a CCS term by using the Graph tool,

3. if the property is structural, match the flow graph interface with the specification and model check the CCS term against the local property by CWB,

4. if the property is behavioural, cast the composed flow graph as a pushdown system using Graph, and model check it against the global property using Moped.

The modular verification method explained in this section can be performed at different levels of granularity. One level is the procedure level. In the next chapter, we will describe ProMoVer, which is a tool that uses the cvpp tool set to automatically perform procedure-modular verification.
Chapter 5

ProMoVer

In this chapter, we introduce ProMoVer which is a fully automated tool for procedure-modular verification. The content of this chapter is adapted from [65].

5.1 Introduction

As explained in the last chapter, the cvpp tool set and framework are suitable to be used in various scenarios of software verification. One useful scenario is modular verification which brings us the support for variability. Modular verification can be achieved at different levels of granularity. We believe that procedure-level is a convenient level because it is the level of abstraction that testers, developers and engineers think.

Procedure-modular verification is a popular approach which has been used in various tools based on the Hoare logic framework, e.g., [29, 53, 9] (see [35] for a survey). 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 chapter 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.

As explained in the previous chapter, the cvpp tool set consists of a number of stand-alone tools that can work together in order to verify the correctness of a property of a Java program. These stand-alone tools cannot talk to each other directly and they need to be invoked by a skillful user in a specific order to provide correct verification results. Even in some cases, the returned results from a tool have to be manually adjusted in order to be used by other tools in the tool set or become understandable for the users. In order to achieve fully automated verification we have developed a wrapper for procedure-modular verification called
ProMoVer [64, 65, 63]. It is equipped with a number of optimization tools for efficiency and easy usage, such as, automatic specification extraction, graph and proof storage and reuse, and a library of global properties.

In this chapter, first, we take a tour through ProMoVer, its usage, design and capabilities, then some experimental results will be presented.

5.2 A User’s View

Our main goal for developing ProMoVer is to achieve fully automatic procedure-modular verification of control flow safety properties. Inspired by the JML annotation language, we designed ProMoVer to accept annotated Java programs where annotations are local specifications and global property in a JML-like syntax. The local specifications consist of a local behavioural property that prohibits the illegal order of method invocations within the corresponding method and the required interface\(^1\). The global interface is extracted from the program and need not to be mentioned in the annotations.

Given an annotated Java program, the correctness of the global program property is relativized on the local properties of the individual methods. Thus, the overall verification task naturally divides into two independent subtasks:

(i) check that the local property of each method matches its implementation, and

(ii) check that the composition of local properties entails the global property.

Notice that the second subtask only relies on the local properties and does not require the implementations of the individual methods.

Control flow safety properties can be expressed in automata-based or process-algebraic notations, as well as in temporal logics such as LTL [66] and the safety fragment of the modal $\mu$-calculus [46]. ProMoVer currently supports the latter two notations, as illustrated by the example below.

**Example 5.1.** Consider again the Java program in Figure 3.1, copied here in Figure 5.1 for the sake of convenience. Figure 5.2 shows this program annotated with a global control flow safety property, and a specification for each method, where each specification consists of a local property and an interface specifying the required methods. To help users, the local method specifications can be extracted by ProMoVer. This feature will be explained in detail in Section 5.6.1.

\(^1\)The provided interface is needless to mention since only the corresponding method is provided.
5.2. A USER'S VIEW

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

Figure 5.1: Flow graph of EvenOdd

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

    /** @local_interface: required even */
    @local_prop:
    * nu X1. (((odd call odd)ff) /
      ([odd]X1) /
      (odd caret even) nu X2.
    * (((odd call odd)ff) /
      (odd caret even)ff) /
    */
    public boolean odd(int n) {
        if (n == 0) return false;
        else return even(n - 1);
    }
}
```

Figure 5.2: A simple annotated Java program
Here we give an intuitive description of the properties specified in the example; a formal definition of the temporal logic LTL is given below in Definition 5.2. 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 symmetric.

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

Example 5.2. 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”, ProMoVer detects this and rechecks the global property for the already computed composition of local properties. The local properties do not have to be reverified. The verification of the global property fails. As a counter example, ProMoVer returns the following program execution that is allowed by the local properties, but violates the global one:

\[(\text{even}, \varepsilon) \xrightarrow{\text{even call odd}} (\text{odd, even}) \xrightarrow{\text{odd ret even}} (\text{even}, \varepsilon)\]

adapted for user understandability by replacing program points with the names of the methods they belong to (cf. Definition 5.1).

5.3 Flow Graph Behaviour

In ProMoVer, for the sake of convenience, we assume that any call to an external method is followed by an immediate return. This means that the external intermediate behaviour between the external method invocations and their returns is ignored. This treatment of method calls is inspired by the temporal logic CARET [4], and is convenient for specifying the local behaviour of flow graphs. In the following definition of the flow graph behaviour, 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 behaviour. When writing global specifications, however, one has to be aware that in this way possible callbacks from external methods are not captured.

By this assumption, the behaviour of an open flow graph is defined as a labeled transition system (LTS) with the following transition labels. Label \(\tau\) for internal transfer of control, \(m_1 \text{ call } m_2\) for the invocation of method \(m_2\) by method \(m_1\) when method \(m_2\) is provided by the program, \(m_2 \text{ ret } m_1\) for the corresponding return from the call, and label \(m_1 \text{ caret } m_2\) for the (atomic) invocation of and return from an external method \(m_2\) by method \(m_1\).
Definition 5.1 (Behaviour). Let $G = (M, E) : (I^+, I^-)$ be an open flow graph such that $M = (V, L, \rightarrow, A, \lambda)$. The behaviour of $G$ is defined as initialized model $b(G) = (M_b, E_b)$, where $M_b = (S_b, L_b, \rightarrow_b, A_b, \lambda_b)$, such that $S_b = V \times V^*$, i.e., states (or configurations) are pairs of control points $v$ and stacks $\sigma$, $L_b = \{m \mid m \in I^+ \cup \{\text{call, ret} \}, m_1, m_2 \in I^+ \} \cup \{m \mid m_1 \in I^+ \land m_2 \notin I^+ \} \cup \{\tau \}, A_b = \lambda_b((v, \sigma)) = \lambda(v)$, and $\rightarrow_b \subseteq S_b \times L_b \times S_b$ is defined by the rules:

\[
\begin{align*}
\text{[transfer]} & \quad (v, \sigma) \xrightarrow{\tau} (v', \sigma) \\
& \quad \text{if } m \in I^+, v \xrightarrow{m} v', v \models \neg r \\
\text{[call]} & \quad (v_1, \sigma) \xrightarrow{m_1 \text{ call } m_2} (v_2, v_1' \cdot \sigma) \\
& \quad \text{if } m_1, m_2 \in I^+, v_1 \xrightarrow{m_2} m_1, v_1 \models \neg r, v_2 \models m_2, v_2 \in E \\
\text{[ret]} & \quad (v_2, v_1 \cdot \sigma) \xrightarrow{m_2 \text{ ret } m_1} (v_1, \sigma) \\
& \quad \text{if } m_1, m_2 \in I^+, v_2 \models m_2 \land r, v_1 \models m_1 \\
\text{[caret]} & \quad (v_1, \sigma) \xrightarrow{m_1 \text{ caret } m_2} (v_1', \sigma) \\
& \quad \text{if } m_1 \in I^+, m_2 \in I^+, v_1 \xrightarrow{m_2} m_1, v_1' \models m_1, v_1 \models \neg r
\end{align*}
\]

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

Example 5.3. Recall the flow graph from Figure 5.1. One example run through its (branching, infinite-state) behaviour, from an initial to a final configuration, is:

\[
\begin{align*}
(v_0, \epsilon) \xrightarrow{\tau} (v_1, \epsilon) \xrightarrow{\tau} (v_2, \epsilon) \xrightarrow{\text{even call odd}} (v_3, v_3) \xrightarrow{\text{odd ret even}} (v_3, \epsilon)
\end{align*}
\]

Now, consider just the method graph of method even as an open flow graph, having interface $\{\text{even}, \{\text{odd}\}\}$. The local contribution of method even to the above global behaviour is the following run:

\[
\begin{align*}
(v_0, \epsilon) \xrightarrow{\tau} (v_1, \epsilon) \xrightarrow{\tau} (v_2, \epsilon) \xrightarrow{\text{even caret odd}} (v_3, \epsilon)
\end{align*}
\]

As mentioned in Chapter 3, flow graph behaviour can be alternatively expressed by pushdown automata (PDA) or pushdown systems (PDS). In this work, we use PDS to express behaviours and use the tool Moped to model check PDS against temporal formulas [45].

5.4 Specification Languages

ProMoVer supports two specification languages: CARet simulation logic, and the safety fragment of LTL. Internally, however, as described in Section 3, the whole machinery is based on simulation logic and maximal flow graph construction uses structural simulation logic. The safety fragment of LTL is somewhat less expressive than simulation logic and can be uniformly encoded in it [23]. In ProMoVer we
have developed a translator that translates the safety fragment of LTL to simulation logic and use these for the underlying machinery.

In this section, we only define the safety fragment of LTL. For the definition, of simulation logic reader is referred to Section 3.1, and for the complete syntax and semantics of LTL to Section 2.3.1.

**Definition 5.2 (Safety LTL).** Let \( p \in A_b \cup \{\text{entry}\} \) and \( m \in M \). The formulae of Safety LTL are inductively defined by:

\[
\phi ::= p \mid \neg p \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid X \phi \mid G \phi \mid \phi_1 W \phi_2
\]

**Example 5.4.** Consider the global property of class `EvenOdd` in Figure 5.2 (where `&&` is ASCII notation for `\land`) and its intuitive meaning in Example 5.1. Flow graph extraction and construction ensures 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 5.1.

### 5.5 Modular Verification

ProMoVer uses an adaptation of beh-comp compositional verification principle introduced in Section 3.4 for the modular verification explained in Section 4.2.2. For a system that is partitioned into \( k \) sets of components, the main principle of modular verification based on maximal flow graphs can be presented, as a proof rule with \( k + 1 \) premises:

\[
\begin{align*}
\mathcal{G}_1 \models \psi_1 & \cdots \mathcal{G}_k \models \psi_k & \bigcup_{i=1,\ldots,k} \text{Max}(\psi_i, I_i) \models_b \phi \\
\bigcup_{i=1,\ldots,k} \mathcal{G}_i & \models_b \phi
\end{align*}
\]

where \( \text{Max}(\psi, I) \) is the maximal flow graph for property \( \psi \) and interface \( I \). The principle states that the composition of the sets of components with the respective interfaces \( \mathcal{G}_1 : I_1, \ldots, \mathcal{G}_k : I_k \) satisfies a behavioural global property \( \phi \) if for some local properties \( \psi_i \) satisfied by the corresponding sets of methods \( \mathcal{G}_i \), the composition of the maximal flow graphs for \( \psi_i \) and \( I_i \) satisfies property \( \phi \).

As mentioned above, 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 5.2 (where \( M \) is the set of program methods, with \( k = |M| \), and \( \psi_i \) and \( C_i \) are the specification and the implementation of method \( m_i \), respectively):
5.6. PROMOVer OVERVIEW

(i) **Checking** $C_i \models \psi_i$ for $i = 1, \ldots, k$: For each method $m_i \in M$, (a) extract the method flow graph $G_i$ from $C_i$, and (b) model check $G_i$ against $\psi_i$. For the latter, we exploit the fact that flow graphs are *Kripke structures*, and apply standard finite-state model checking.

(ii) **Checking** $\bigcup_{i=1}^{k} \text{Max}(\psi_i, I_i) \models \phi$: (a) Construct maximal flow graphs $\text{Max}(\psi_i, I_i)$ for all method specifications $\psi_i$ and interfaces $I_i$, then (b) compose the graphs, resulting in flow graph $G_{\text{Max}}$, and finally (c) model check $G_{\text{Max}}$ against global property $\phi$. For the latter, represent the behaviour of $G_{\text{Max}}$ as a PDS and use a standard PDS model checker.

**Example 5.5.** Consider again the annotated Java program from Example 5.1. PROMOVer first extracts the method flow graphs of methods `even` and `odd`, denoted $G_{\text{even}}$ and $G_{\text{odd}}$, respectively. Next, PROMOVer checks $G_{\text{even}} \models \psi_{\text{even}}$ and $G_{\text{odd}} \models \psi_{\text{odd}}$ by standard finite state model checking. Independently, it constructs the maximal flow graphs of methods `even` and `odd`, denote $\text{Max}(\psi_{\text{even}}, I_{\text{even}})$ and $\text{Max}(\psi_{\text{odd}}, I_{\text{odd}})$, respectively, and composes the flow graphs to obtain $G_{\text{Max}} = \text{Max}(\psi_{\text{even}}, I_{\text{even}}) \cup \text{Max}(\psi_{\text{odd}}, I_{\text{odd}})$. Finally, PROMOVer translates $G_{\text{Max}}$ to a PDS and model checks the latter against the global property.

As briefly mentioned in Chapter 3, the CVPP framework provides an inlining based technique for abstracting away private methods, considering these as for implementing the public ones (see [34] for details). PROMOVer implements this, requiring only the public methods to be specified with local properties.

5.6 ProMoVer Overview

Figure 5.3 shows overview of ProMoVer and its internal tools. As it is seen in the figure, PROMOVer essentially consists of a pre-processor which parses the annotated Java program as input and retrieves the local specifications and global properties. Then the proper tools from the CVPP tool set are invoked given the proper arguments and the specifications and property. The box labeled with (i) wraps the tools that are used for the local check (task (i)) and the box labeled with (ii) wraps the tools that are used for the global check (task (ii)).

In task (i), first Analyzer is invoked to extract the method graph of the program. Then Graph is used to translate the flow graph of each method into a CCS term. All the CCS terms are then model checked against the respective local specifications using CWB.

In task (ii), first a maximal flow graph of each method is constructed using Maximal Model. Then Graph is used to compose the maximal flow graphs and convert them into a PDS. Finally, Moped is used to model check the PDS against the global property.

The post-processor collects all model checking results and converts them into a user-understandable format. It only returns positive answer if all model checking
tasks successfully finished. If one of the local model checking tasks fails, the name of
the respective method is returned. If the global model checking fails, the provided
counter example by Moped, is transformed into a program execution and returned.

ProMoVer

Figure 5.3: Overview of ProMoVer and underlying tools

If ProMoVer is invoked with inliner option, private methods are not specified.
In this case, Graph Inliner will be used in task (i) to inline private method graphs
into public method graphs after the method graphs are extracted by Analyzer
and before Graph translates the method graphs to CCS terms. Also in task (ii),
maximal model of private methods are not constructed. Using inliner option, in
practical applications, significantly reduces the number of specifications.

5.6.1 Automatic Specification Extraction

The most time consuming, and perhaps relatively difficult, part of the modular
verification is to find out the appropriate local specifications. To assist users for
writing specifications, ProMoVer provides support to extract a specification from
a given method implementation. The resulting specification is the \( \tau \) insensitive
over-approximation order of method invocations in the given method expressed in
modal equation system (as defined in Chapter 3). This result can be directly used
for verification, however, might be unnecessarily specific for this implementation.
Therefore, to relax the specification for the possible code evolution, user can inspect
the extracted specification and remove superfluous method orders.

To illustrate the automatic specification extraction result, let’s consider the even-
odd program in Figure 5.1. The specification extraction for method even results
the following modal equation system (defined in Definition 3.6), where $\texttt{eps}$ is ASCII notation for $\varepsilon$, and $\texttt{ff}$ denotes $false$.

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

This specifies that method $\texttt{odd}$ may be called at most once: initially $X0$ holds, and method $\texttt{odd}$ may be called or an internal step (labelled $\texttt{eps}$) may be made. After calling $\texttt{odd}$, $X1$ should hold and only internal steps are allowed.

As a more involved example, consider method $\texttt{m}$ and its extracted specification:

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

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

The formula captures that first only $\texttt{m1}$ can be called, then only $\texttt{m2}$, and then either $\texttt{m3}$ or $\texttt{m4}$, and no further calls can be made. However, the order of invocation of $\texttt{m1}$ and $\texttt{m2}$ is not important for this program, thus the user can refine the extracted specification and remove this order from the specification. So 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):

```java
X0 = [m4]ff \land [m1](X10) \land [m3]ff \land [m2](X11) \land [m]ff \land [eps]X0;
X10 = [m4]ff \land [m1]ff \land [m3]ff \land [m2](X2) \land [m]ff \land [eps]X10;
X11 = [m4]ff \land [m1](X2) \land [m3]ff \land [m2]ff \land [m]ff \land [eps]X11;
```

5.6.2 Proof Storage and Reuse

ProMoVeR stores all extracted method graphs and constructed maximal flow graphs when a program is verified. These will be used in the next verifications.
if the corresponding implementation or specification has not changed. However, if the implementation of method \( m \) is changed, a new method graph is extracted and checked against its specification. If the specification of method \( m \) is changed, its existing method graph is checked against the new specification, a new maximal flow graph is constructed from \( m \)'s specification, composed with all other existing maximal flow graph of the methods of the program and the result of the composition is model checked against the global property.

The proof storage and reuse mechanism facilitates the reuse of components and libraries. It also provides a full support for verification of open systems. Moreover, in case of code evolution the stored maximal flow graphs and method graphs can be reused.

As an application scenario, API libraries can be delivered with local specifications and maximal flow graphs of their methods. Later, whenever these are used, the verification will reuse the maximal flow graphs of the API methods, compose them with the maximal flow graphs of the newly developed methods and perform the model checking.

Given the fact that maximal flow graph construction is exponential in the size of the specification and extracting method graph is expensive, proof storage and reuse provides a significant improvement in the scalability.

### 5.6.3 Property Specification Library

ProMoVer's web interface provides a collection of global properties expressed in safety LTL. These properties are platform or application specific properties, prohibiting insecure order of API function calls for a specific platform or application.

### 5.7 Experiments

In this section, we present our experiments with ProMoVer. In the experiments we verify a standard control flow safety property of a number of Java Card applications.

Java Card technology provides a secure environment to support applications on smart cards, developed by Sun Microsystems. It is one of the leading interoperable platforms for smart cards. Many smart card applications are security-critical implementing for example e-commerce applications.

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. Safe use of it demands that certain methods are not called 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.
5.7. EXPERIMENTS

Below, we first take a closer look at the transaction mechanism of Java Card, and then describe the JavaPurse application and the properties we are interested in verifying.

The Java Card Transaction Mechanism. Smart cards have two types of writable memory, persistent memory (EEPROM or Flash) and transient memory (RAM). The Java Card memory model adheres to this. 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.

As an example, assume that an array of data containing 16 elements is updated from transient to persistent memory and this operation is performed non-atomically. If after updating 8 elements of the array card tear occurs, the persistent memory will be left in an inconsistent state and the data of the array in persistent memory will be corrupted.

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 [40] – all declared in class JCSystem of the Java Card API, which is a class for controlling applet execution, resource management, atomic transaction management, object deletion, and inter-applet object sharing in the Java Card environment [40].

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. Typical Java Card programming standards, such as the Global Platform specification, state that these methods may not be used within a transaction (for safe array updating within a transaction, the class provides the atomic method arrayCopy). We use PROMoVeR to verify in a procedure-modular way that applications comply with this Safe 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 net-
CHAPTER 5. PROMOVER

<table>
<thead>
<tr>
<th>Application</th>
<th>#LoC</th>
<th>#Methods (Public)</th>
<th>#Calls (Relevant)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccountAccessor</td>
<td>190</td>
<td>9 (7)</td>
<td>38 (4)</td>
</tr>
<tr>
<td>TransitApplet</td>
<td>918</td>
<td>18 (5)</td>
<td>106 (5)</td>
</tr>
<tr>
<td>JavaPurse</td>
<td>884</td>
<td>19 (9)</td>
<td>190 (25)</td>
</tr>
</tbody>
</table>

Table 5.1: Applications details

work service. It contains methods to look up and to 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 information related to a loyalty program, and to validate and update the values of transactions, balance and PIN code.

Table 5.1 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.

JavaPurse is a smart card electronic purse application providing secure money transfers. The application contains a balance record denoting the user’s current and maximum credits. It contains methods `processInitializeTransaction` and `processCompleteTransaction` that initialize, perform and complete a secure transaction using the Java Card transaction mechanism. Further, it also contains methods to update information related to a loyalty program, and to validate and update the values of transactions, balance and PIN code. These data updates use the API methods `arrayFillNonAtomic`, `arrayCopyNonAtomic` and `arrayCopy` mentioned above. This functionality of the JavaPurse application is implemented by means of 19 methods with approximately 1K lines of code in total.

The JavaPurse application contains 222 method calls, 15 of which are method calls to `arrayCopyNonAtomic` and 6 to `arrayFillNonAtomic`. Transactions are used in two places. One of the transactions contains 2 API method invocations, the other contains 3 API method invocations. Method `abortTransaction` is not used in JavaPurse, and is not considered here.

**Specification of Safe 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:

\[ \text{G}(\text{beginTransaction} \rightarrow (\text{¬arrayCopyNonAtomic} \land \text{¬arrayFillNonAtomic}) \text{W commitTransaction}) \]

### Extracting Local Method Specifications.

The specification extractor is used to obtain local specifications for every public method. Basically, these describe the order of method invocations. We inspected those for immaterial orderings, and translated the adjusted representations into safety LTL. The intention is that local method specifications capture the allowed sequences of method calls made from within the specified method, but in an abstract way, allowing for possible evolution of the method implementations.

### Verification Results.

After annotating the applications, they are passed to ProMoVer. The tool extracts the flow graph of the applications, and partitions them into the individual method graphs to verify adherence to the local properties. Further, for each local property a maximal flow graph is constructed, and their composition is verified w.r.t. the global property above. The statistics for these verifications are given summarized in Table 5.2. The table shows: the time spent by the pre-processor (PPT) and the graph extractor (GE), 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 times are in seconds, and were obtained on a SUN SPARC machine.

We also experimentally evaluated the advantages of exploiting the proof storage and reuse mechanism. After the first verification, when method and maximal flow graphs are stored, for each application, we once changed the source code and once the local specification of a public method, and used ProMoVer to reverify the application. The result of proof reuse are shown in Table 5.3. The numbers show that proof reuse can reduce significantly the verification time for larger applications.
5.8 Conclusion

In this chapter, we introduce ProMoVer which is a fully automated tool for procedure-modular verification. It encapsulates a previously developed tool set for compositional verification for a procedure-modular verification scenario. ProMoVer accepts a Java program annotated with method specifications and the global property as input and verifies the entailment of global property by performing two independent verification tasks. Task (i) is to verify that every method specification matches its implementation and task (ii) is to relativize the global property on the composition of the method specifications. The modularity of the verification allows an independent evolution of the implementations of the individual methods. We illustrate two important points: 1. temporal safety properties provide a meaningful abstraction for individual methods; and 2. procedure-modular verification of temporal safety properties can be performed automatically. 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 the presence of static and dynamic variability, as is the case, e.g., for mobile code or software product lines.

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.

One of the applications of ProMoVer is software product line, that is, “a family of products designed to take advantage of their common aspects and predicted variabilities” [68]. In Chapter 6, we adapt ProMoVer for compositional verification of software product lines.

Many important safety properties require program data to be taken into account. As a first step towards handling data, in Chapter 7, we enhance the CVPP framework with Boolean data.
Chapter 6

Verification of Software Families

In this chapter, we present an adaptation of the CVPP compositional verification technique and ProMoVer for a particular application area, namely software product families. The content of this chapter is adapted from [61].

6.1 Introduction

Software industry increasingly demands for delivering systems that simultaneously exist in many different variants in order to adapt to their application context. Software product line engineering [58] aims at developing a family of products with predicted commonalities and variabilities by managed reuse in order to decrease time to market and improve quality. It is an industrial design approach for reuse and quality improvement in software engineering and it is proved to be commercially successful [51]. In the product line design, usually two sets of artifacts are developed, commonality (here also called core) and variability. Commonality artifacts are the essential parts of all products of the product family. While, variability artifacts are used to produce variety of different products.

Usually the verification techniques for software product lines aim at verifying a desired property for all products in the product family. Nevertheless, the verification of software product families brings new challenges when compared to verification of ordinary software systems: the number of products that can be produced from a software product line is exponential in the number of variability artifacts. Therefore, verifying each and every product is extremely inefficient. The verification of such large number of systems can only be accomplished compositionally. Unfortunately, the current formal modeling and (specially) verification techniques for product lines are not scalable and applicable to industrial-size applications.

In this chapter, we present a formal model that is flexible enough to capture the relevant aspects of commonalities and variabilities of software product lines, at the same time suits the compositional verification based on the CVPP framework. Then, we show an annotation language that can represent the formal model of product
lines, we will integrate the annotation language into ProMoVER and evaluate the tool.

6.2 Related Work

As mentioned before, there are only a few verification techniques proposed for product lines. In this section, we review the most influential of the existing verification techniques for such systems (see [22] for a survey). Then, we review the existing modeling techniques for them.

6.2.1 Verification Techniques for Product Families

Model checking. Some existing model checking techniques are adapted to verify the optional behaviour defined by variant annotations. Among these the most cited approaches are: the work by A. Fantechi et al. where modal transition systems are extracted by the variability operators requires and possible [28]; A. Classen et al. propose a labeled transition system that is constructed from a product family and features appear as the labels in a way that state reachability on a set of features can be computed [20]; A. Gruler et al. use an extension of CCS process calculus by variant operators to model a family of processes [31]; and in the work by K. Lauenroth, transitions on I/O-automata are related to variants [48].

Compositional verification. Blundell et al. (in [13]) and Liu et al. (in [50]) have proposed different techniques for compositional verification of product families. These are the only existing compositional verification techniques suggested for such systems. In these techniques, each feature is represented as a state machine, while the other features can be attached to it via a particular type of states, called interface states or variation points. A temporal property is verified if the composition of desired features satisfy the property, i.e., the compositional results are based on the features and their composition.

As stated, ordinary model checking and verification techniques are not scalable for product line applications. Also, the few compositional approaches that has been proposed so far, are very restrictive. However, the CVPP compositional verification technique and ProMoVER can handle variability and can be perfectly used for such applications. Hence, we adapt ProMoVER for compositionally verifying properties of product families.

6.2.2 Modeling Techniques for Product Families

The existing approaches for modeling variability of software product lines can be classified into three categories according to the types of models: (i) annotative approaches, as in [69, 30], where a model is for representing all products; (ii) compositional approaches, as in [10, 67, 54, 6], which represents an association between
product fragments, features and the fragments that are involved in forming a feature configuration; (iii) transformation approaches, as in [36], which variability is represented through a set of rules that specify the fragments of a base model that have to be replaced for each particular product model.

Since none of the existing modeling techniques suits the compositional verification by PROMoVER, we propose a hierarchical variability model, where the model represents all products and the variabilities are represented by explicit variations and variation points over the hierarchy. This model, when defined formally, should be appropriate to be used by the cvpp framework since it is hierarchical and each level of the hierarchy can be a subject of an independent compositional verification.

6.3 A Hierarchical Variability Model

In this section, we define the simple hierarchical variability model (SHVM) which is a formal modeling of commonalities and variabilities. Each SHVM is used for producing a family of products. Here, a product is a set of methods. The product need not be closed; it can call external methods, e.g., API methods. Each method \( m \) from a set of methods \( \text{Meth} \) consists of a method name, the type of the parameters and return value, and the its body. Methods are either public or private. As usual public methods are visible to the outside of the product and private methods are only visible from the product. Interface is defined for each product by provided and required methods.

The Hierarchical Variability Model is used to represent a product line \( PL \) that is a set of method sets \( PL \subseteq 2^{\text{Meth}} \). In \( PL \) the commonalities are represented as a core set of methods and the variabilities are represented by variation points. Each variation point consists of a set of variants. A variant contains a set of core methods and a set of variation points, i.e., it can be either a set or core methods or a hierarchical variability model itself. A product from a hierarchical variability model is produced by selection variants at the variation points on each level of the hierarchy.

In this chapter, we use simple hierarchical variability model (SHVM) which is a simplified version of the hierarchical variability model explained above. The simplification is such that from each variation point it is required to select exactly one variant to obtain a product. Also in SHVMs we disallow the selection constrains over variants. By these simplifications we provide a clean model that can be directly used for the compositional verification purpose.

Definition 6.1 (Simple Hierarchical Variability Model). A simple hierarchical variability model (SHVM) \( S \) is inductively defined as:

(i) a ground model consisting of a core set of methods \( M_C = (M_{\text{pub}}, M_{\text{priv}}) \), partitioned into public and private methods \( M_{\text{pub}}, M_{\text{priv}} \subseteq \text{Meth} \), or

(ii) a pair \( (M_C, \{VP_1, \ldots, VP_N\}) \), where \( M_C \) is defined as above and where \( \{VP_1, \ldots, VP_N\} \) is a non-empty set of variation points. A variation
point \( VP_i = \{S_{i,j} \mid 1 \leq j \leq k_i \} \) is a non-empty set of SHVMs. The members of a variation point are called variants.

The variant interface of a pair \((M_C, \{VP_1, \ldots, VP_N\})\) is defined as a pair of public required and public provided methods. The set of public provided methods is the union of all sets of public provided methods in the core methods and the variation points. The set of public required methods is the union of all sets of public methods required by the core methods and by the variation points without the methods provided by the core methods or another variation point.

We impose two following well-formedness constrains on SHVMs. The first constrain is that all of the variation points have to provide and require the same set of public methods. This pair of provided and public methods is called variation point interface. By this constraint, all the variants will provide the same public visibility and the variability is provided by using different implementations for public methods and different private methods. The second constraint is that the set of provided methods in the interfaces of variation points of an SHVM should be disjoint with each other and the core method set. By this constrain we guarantee that the products will not have several methods with the same names.

**Example 6.1.** Lets consider a product line of cash desks that is a simplified version of the trading system product line case study proposed in [59]. Each product produced from the case desk product line, processes the purchases by retrieving the prices of all items to be purchased and at the end calculating the total price. When the total price is calculated, the purchase is made by printing a receipt and updating the stock accordingly. The purchase process is the commonality part of all cash desk products while the variability part comes from the methods for entering goods and payment.

One instantiation of the cash desk is where the cash desk product line that provides entering products using keyboard, scanner, or a both keyboard and scanner which can be chosen by the cashier. We call this instantiation CashDesk and we use it as a running example throughout this chapter. The CashDesk example can be defined by the following SHVM.

\[
\text{CashDesk} = (\{(\text{sale}),\{\text{updateStock, writeReceipt}\}\),
(\text{@EnterProducts, @Payment}\})
\]

where \(\text{@EnterProducts} = \{\text{Keyboard, Scanner, KeyboardOrScanner}\}\)
\(\text{@Payment} = \{\text{Cash, Card, CashOrCard}\}\)

\(\text{and Keyboard} = (\{(\text{enterProd}),\{\text{useKeyboard}\}\})\)
\(\text{Scanner} = (\{(\text{enterProd}),\{\text{useScanner}\}\})\)
\(\text{KeyboardOrScanner} = (\{(\text{enterProd}),\{\text{useScanner, useKeyboard}\}\})\)
\(\text{Cash} = (\{(\text{payment}),\{\text{cashPay}\}\})\)
\(\text{Card} = (\{(\text{payment}),\{\text{enterCard, cardPay}\}\})\)
\(\text{CashOrCard} = (\{(\text{payment}),\{\text{cashPay, enterCard, cardPay}\}\})\)
In the CashDesk example the commonalities are represented by sale code public method. The private methods updateStock and writeReceipt are the helper methods to perform internal details of the sale process. The two variation points @EnterProducts and @Payment represent the variabilities of the cash desks. The variation point @EnterProducts has the associated variants Keyboard, Scanner and Keyboard-Or-Scanner for entering product by keyboard, by scanner or providing both options. Both provide the public method enterProd that is internally realized by the different private methods useKeyboard, useScanner or their combination. Similarly, the variation point @EnterProducts has the associated variants Cash, Card and CashOrCard that provide the public method payment which is internally realized by different private methods in the respective variants.

Figure 6.1: The CashDesk SHVM

An SHVM can be shown as a tri-partite directed graph having an SHVM-node as root, where SHVM-nodes have one core methods leaf child (split in public and private methods) and optional VP-node children that have two or more SHVM-node children. For the cashdesk example, a graphical presentation is shown in Figure 6.1. In the figure, SHVM-nodes are depicted by blue rounded boxes, the nodes for core methods by red ovals, and VP-nodes by yellow diamonds. The dotted rounded boxes depict what we call modules of the SHVM, defining the boundaries between SHVMs at different levels of hierarchy. The size of an SHVM is defined as the number of modules in its graph.

An SHVM induces a set of products $P$ through all possible ways of resolving the variabilities of the SHVM. Variability resolution means to recursively select exactly one variant for each variation point. The set of products induced by a ground model containing only core methods is the singleton set comprising the set of core methods (and, thus, representing one product). The set of products induced by a
variation point is the union of the product sets induced by its variants. Finally, the
set of products induced by an SHVM with a non-empty set of variation points is
the set of all products consisting of the core methods and of exactly one product
from the set induced by each variation point.

**Definition 6.2 (Variability Resolution).** Let $S$ be an SHVM as defined above. The
set $\text{products}(S) \subseteq 2^{M_{\text{eth}}}$ induced by $S$ is inductively defined as follows:

\[
\begin{align*}
\text{products}(M_C) & = \{M_C\} \\
\text{products}(VP) & = \bigcup_{S \in VP} \text{products}(S) \\
\text{products}(M_C, \{VP_1, \ldots, VP_N\}) & = \{M_C \cup \bigcup_{1 \leq i \leq N} M_i \mid M_i \in \text{products}(VP_i)\}
\end{align*}
\]

**Example 6.2.** The SHVM defined in Example 6.1 induces the products:

\[
\text{products}(\text{CashDesk}) = \{P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9\}
\]

where:

\[
\begin{align*}
P_1 & = \{\text{sale, updateStock, writeReceipt, enterProd}_{\text{Keyboard}}, \text{useKeyboard, payment}_{\text{Cash}}, \text{cashPay}\} \\
P_2 & = \{\text{sale, updateStock, writeReceipt, enterProd}_{\text{Scanner}}, \text{useScanner, payment}_{\text{Cash}}, \text{cashPay}\} \\
P_3 & = \{\text{sale, updateStock, writeReceipt, enterProd}_{\text{KeyboardOrScanner}}, \text{useKeyboard, useScanner, payment}_{\text{Cash}}, \text{cashPay}\} \\
P_4 & = \{\text{sale, updateStock, writeReceipt, enterProd}_{\text{Keyboard}}, \text{useKeyboard, payment}_{\text{Card}}, \text{enterCard, cardPay}\} \\
P_5 & = \{\text{sale, updateStock, writeReceipt, enterProd}_{\text{Scanner}}, \text{useScanner, payment}_{\text{Card}}, \text{enterCard, cardPay}\} \\
P_6 & = \{\text{sale, updateStock, writeReceipt, enterProd}_{\text{KeyboardOrScanner}}, \text{useKeyboard, useScanner, payment}_{\text{Card}}, \text{enterCard, cardPay}\} \\
P_7 & = \{\text{sale, updateStock, writeReceipt, enterProd}_{\text{Keyboard}}, \text{useKeyboard, payment}_{\text{CashOrCard}}, \text{cashPay, enterCard, cardPay}\} \\
P_8 & = \{\text{sale, updateStock, writeReceipt, enterProd}_{\text{Scanner}}, \text{useScanner, payment}_{\text{CashOrCard}}, \text{CashPay, enterCard, cardPay}\} \\
P_9 & = \{\text{sale, updateStock, writeReceipt, enterProd}_{\text{KeyboardOrScanner}}, \text{useKeyboard, useScanner, payment}_{\text{CashOrCard}}, \text{cashPay, enterCard, cardPay}\}
\end{align*}
\]

To disambiguate methods with the same name, but coming from different variants,
we add as subscript the name of the parent SHVM-node, for instance,
$\text{enterProd}_{\text{Keyboard}}$ refers to the method $\text{enterProd}$ of the variant $\text{Keyboard}$.

For a given SHVM, let $\text{AND}$ and $\text{OR}$ denote the maximal branching factors at
SHVM and variation point nodes, respectively, and let $\text{ND}$ be its nesting depth.
The number of products induced by the SHVM is bound by
\[ OR^{\text{ND} \cdot (\text{AND}^{\text{ND+1}} - 1) \cdot (\text{AND} - 1)} \]
and is thus exponential in the size of the SHVM, which is bound by
\[ \frac{(OR \cdot AND)^{(\text{ND+1})} - 1}{OR \cdot AND - 1}. \]
These bounds are obtained in a routine fashion by solving the corresponding recurrence relations. Notice that in SHVMs with a small nesting depth as in the example above, the exponential blow-up in the number of products is not observed: With branching factors of 3 and a nesting depth of 1, we have at most 9 products, but 7 modules. However, adding just another level of hierarchy, e.g., variability in the accepted type of cards, immediately results in an explosion (see Section 6.5).

6.4 Compositional Verification of SHVM

In this section we adapt the cvpp compositional verification technique for verification of SHVMs. The main motivation for this adaptation is that the compositional verification is linear in the size of the SHVM while if one applies non-compositional verification, each product has to be verified independently and then the number of individual verifications for an SHVM is exponential in the size of the SHVM.

For the cvpp compositional verification technique, we need to specify the components of the system to be able to use compositional verification. Here, we specify every module \((MC, \{VP_1, \ldots, VP_N\})\) of the SHVM by a behavioural property \(\phi\) and its public interface \(I\). We also specify each variation point \(VP_i\) by its behavioural property \(\psi_{VP_i}\) and its public interface \(I_{VP_i}\). Every public method \(m\) should be specified by its behavioural property \(\psi_m\) and its public interface \(I_m = (I^+_m, I^-_m)\). The SHVM nodes in the variation points \(VP_i\) inherit their specification from \(VP_i\). The top most SHVM node will be specified by global property.

Having the specifications we can verify each SHVM module separately by:

(i) locally checking the properties of underlying core methods and variation points, and

(ii) globally relativizing the property of the SHVM node on the composition of the maximal flow graphs constructed from the underlying variation points and core methods.

Notice that the local check of each variation point gives raise to compositional check of the including SHVM nodes. More formally we can propose the following verification procedure for the compositional verification of SHVMs.
CHAPTER 6. VERIFICATION OF SOFTWARE FAMILIES

Verification Procedure  For every module \((M_C, \{VP_1, \ldots, VP_N\})\) of the SHVM, perform the following two independent tasks:

1. For every public method \(m \in M_{pub}\), extract the method graph \(G'_m\) from the implementation of \(m\), then inline the already extracted graphs of the private methods, and finally model check the resulting method graph \(G'_m\) against the specification \(\psi_m\) of \(m\) to establish \(G'_m \models \psi_m\). For the latter, we apply standard finite-state model checking.

2. For all public methods \(m \in M_{pub}\) with specification \((I_m, \psi_m)\), construct the maximal method graphs \(\mathsf{Max}(\psi_m, I_m)\), and for all variation points \(VP\), with specification \((I_{VP}, \psi_{VP})\), construct the maximal flow graphs \(\mathsf{Max}(\psi_{VP}, I_{VP})\). Then, compose the constructed graphs, resulting in flow graph \(G_{\mathsf{Max}}\), and model check the latter against the SHVM property \(\phi\), i.e.,

\[
\bigcup_{m \in M_{pub}} \mathsf{Max}(\psi_m, I_m) \uplus \bigcup_{1 \leq i \leq N} \mathsf{Max}(\psi_{VP_i}, I_{VP_i}) \models \phi \quad (6.1)
\]

For properties given in sLTL, we represent the behaviour of \(G_{\mathsf{Max}}\) as a PDS and use standard PDS model checking.

With the above verification procedure the total number of verification tasks needed to establish the global product line property is, thus, equal to the number of modules, because we have to complete one verification task per module. In contrast, the number of products is exponential in the number of modules.

By the following theorem we prove that the given verification procedure is sound for any SHVM.

Theorem 6.3. Let \(S\) be an SHVM with global property \(\phi\). If the verification procedure succeeds for \(S\), then \(p \models \phi\) for all its products \(p \in \text{products}(S)\).

Proof. The proof is by induction on the structure of \(S\). For the base case, let \(S\) be a ground model, i.e., a core set of methods \(M_C = (M_{pub}, M_{priv})\) with no variation points. Assume the verification procedure succeeds for \(S\). It has then established:

1. \(G'_m \models \psi_m\) for all public methods \(m \in M_{pub}\), and
2. \(\bigcup_{m \in M_{pub}} \mathsf{Max}(\psi_m, I_m) \models \phi\)

From these, and by soundness of rule (5.1) refined for private method abstraction, it follows \(M_C \models \phi\). Since \(\text{products}(S) = \{M_C\}\) in this case, we have \(p \models \phi\) for all \(p \in \text{products}(S)\).

For the induction step, let \(S\) be a non-ground model \((M_C, \{VP_1, \ldots, VP_N\})\) with variation points \(VP = \{S_{i,j} \mid 1 \leq j \leq k_i\}\), where \(k_i\) is the number of variants of \(VP_i\). Further, let \((\psi_{VP_i}, I_{VP_i})\) be the specification of \(VP_i\). Assume the result for all \(S_{i,j}\) (induction hypothesis). Next, assume that the verification procedure succeeds for \(S\). The following has then been established for the top-level module:
6.4. COMPOSITIONAL VERIFICATION OF SHVM

(i) $G'_m \models \psi_m$ for all public methods $m \in M_{pub}$, and

(ii) $\left( \bigcup_{m \in M_{pub}} \text{Max}(\psi_m, I_m) \cup \bigcup_{1 \leq i \leq N} \text{Max}(\psi_{VP_i}, I_{VP_i}) \right) \models \phi$

By the assumption, the verification procedure has also succeeded for all $S_{i,j}$. 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:

$$\forall i : 1 \leq i \leq N. \forall j : 1 \leq j \leq k_i. \forall p \in \text{products}(S_{i,j}). p \models \psi_{VP_i},$$

By Definition 6.2 we have $\text{products}(VP_i) = \bigcup_{1 \leq j \leq k_i} \text{products}(S_{i,j})$, and hence:

(iii) $\forall i : 1 \leq i \leq N. \forall p \in \text{products}(VP_i). p \models \psi_{VP_i}$

Also by Definition 6.2, we know that every product $p$ of $S$ is the union of a core $M_C$ and exactly one sub-product from every variation point. Due to (i), the public methods of $M_C$, after inlining the private ones, meet their respective specifications. Similarly, by (iii), all sub-products meet their respective specifications. Finally, by (ii) and from soundness of rule (5.1) refined for private method abstraction follows that $p \models \phi$. This concludes the proof.

Example 6.3. To illustrate our compositional verification approach, we use the cashdesk product line described in Example 6.1. The global behavioural 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 methods 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) behaviour of the flow graphs of the products induced by the product line, this property can be formalized in sLTL as follows:

$$\varphi_{CD} = sale \rightarrow (\neg payment \ W (enterProd \land r \land X sale))$$

where the sub-formula $enterProd \land r \land X sale$ captures a return from enterProd to sale.

First, we have to specify all public core methods and variation points of the cashdesk SHVM. The specification of the sale method and the @EnterProd and @Payment variation points are as follows:
CHAPTER 6. VERIFICATION OF SOFTWARE FAMILIES

- The interface of method sale is \( I_{sale} = \{sale\}, \{\text{enterProd}, \text{payment}\} \). In order to entail the global property, the local behavioural property that method sale (or, more precisely, its method graph as an open flow graph) has to satisfy is that it has to have invoked method enterProd and returned from the call before it can invoke method payment, after the return from which no more methods are invoked. Formally, this can be expressed by the SMTL formula:

\[
\varphi_{sale} = sale \cdot enterProd \cdot sale \cdot payment \cdot (G sale)
\]

where the derived temporal operator \( \phi \cdot \psi \) abbreviates \( \phi \land (\phi \cdot \psi) \) and is by convention right-associative.

- The interface of variation point @EnterProducts is \( I_{EP} = \{\text{enterProd}\}, \{\text{payment}\} \). The property required for the variation point is that the enterProd method never calls the payment method, neither directly nor via a call to one of its non-public methods. Formally, this property can be expressed by the formula:\n
\[
\varphi_{EP} = G \neg \text{payment}
\]

- The interface of variation point @Payment is \( I_{P} = \{\text{payment}\}, \{\text{enterProd}\} \). Similarly to the variation point above, the property required for this variation point is that the payment method never calls the enterProd method:

\[
\varphi_{P} = G \neg \text{enterProd}
\]

The variants Keyboard, Scanner, KeyboardOrScanner, Cash, Card and CashOrCard inherit the specification of their SHVM node from the respective variation point specification. The specification of the public methods enterProd and payment is similar to the specification of the @EnterProd and @Payment variation points.

Next, we have to verify that all public methods satisfy their behavioural property. For the sale method, we have to inline the private methods writeReceipt and updateStock to obtain the method graph of the sale method, Then we check that the method graph satisfies the property \( \varphi_{sale} \) by finite-state model checking. Similarly, we verify the enterProd and payment methods defined in the variants Keyboard, Scanner, KeyboardOrScanner, Cash, Card and CashOrCard.

Finally, we have to establish that all SHVMs satisfy their SHVM specification. For the top-level SHVM, we construct the maximal models for the specifications of the variation points @EnterProducts and @Payment and for the public method \( \varphi_{sale} \), and model check \( \varphi_{CD} \) against the composition of these maximal models. The properties of the variants Keyboard, Scanner, KeyboardOrScanner, Cash, Card

---

1This 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.
6.5. EXPERIMENTS

Figure 6.2: Annotations for variation point @EnterProd and its variant Keyboard

and CashOrCard are easy to verify because each of them contains only one public method. A maximal model for the specification of this public method is constructed and checked against the inherited variation point property.

6.5 Experiments

In this section, we explain the tool support for compositional verification of SHVMs and also we evaluate our technique and its tool support by performing some experiments on some variations of the cash desk example.

We have adapted ProMoVER for verifying properties of SHVMs according to the compositionality principle described in Section 6.4. For this adaptation, we have extended the annotation language to support the definition of core methods, 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, the variation point properties and the specifications of the public core methods are also provided by annotations. Figure 6.2 shows in the left column the annotation for the @EnterProd variation point, while the annotations for its Keyboard variant with core method enterProd are 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 6.3 for different versions of the trading system product line [59]. 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 6.1 and extended it by
an optional coupon handling functionality within the \texttt{sale} method, and a variation point for accepting different card types as a hierarchical refinement of variant \texttt{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.\footnote{The focus of the evaluation is on comparing the times required for verification, and not on the total times themselves.}

The results are summarized in Table 6.1 where \texttt{CD} denotes the product line of Example 6.1, \texttt{CD/CH} the version with coupon handling, \texttt{CD/CT} the version with different card types and \texttt{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 6.3. 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 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.

<table>
<thead>
<tr>
<th>Product Line</th>
<th>Depth</th>
<th># Modules</th>
<th># Products</th>
<th>$t_{ind}$[s]</th>
<th>$t_{comp}$[s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>CD</td>
<td>1</td>
<td>7</td>
<td>9</td>
<td>79</td>
<td>9</td>
</tr>
<tr>
<td>CD/CH</td>
<td>1</td>
<td>9</td>
<td>18</td>
<td>177</td>
<td>10</td>
</tr>
<tr>
<td>CD/CT</td>
<td>2</td>
<td>15</td>
<td>27</td>
<td>278</td>
<td>11</td>
</tr>
<tr>
<td>CD/CH/CT</td>
<td>2</td>
<td>17</td>
<td>54</td>
<td>652</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 6.1: Evaluation Results
variant constraints. In this case, an additional check of the excluded products would be required.

6.6 Conclusion

In this chapter, we introduce software product line as a suitable application area for the cvpp framework and ProMoVer. In particular, we propose a new formalization for modeling software product lines called SHVM which matches the compositional verification method. Moreover, we propose a verification procedure for SHVMs and prove its soundness. In addition, we provide tool support for verification of software product lines by adapting ProMoVer. The proposed compositional verification procedure is linear in size of the SHVM, while, in contrast, the number of products is exponential. At the end, by performing experiments we evaluate our approach and the tool support practically.

One possible direction for future work is to enrich the hierarchical variability model by allowing optional variants and constrains on their composition, e.g., a constraint that prohibits products to include variants $V_1$ and $V_2$.

One other possible adjunct is to provide support for constructing a hierarchical variability model from an existing product line.
Chapter 7

Boolean Flow Graphs

Despite that the cvpp framework and ProMoVER are shown to be practically useful and technically flexible for programs with static and dynamic variability, the main 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 the properties this method can handle is limited to control flow properties in absence of data.

In this chapter, we develop a first step towards extending the cvpp framework with data. For this, we suggest an extension to the framework, described in Chapter 3, by adding Boolean data to the program model and the compositional verification framework.

For the examples of this chapter, we use Boolean programs, that are imperative programs which allow Boolean variables only. The reason is that Boolean programs can be precisely modelled by Boolean flow graphs, i.e., they do not need to be abstracted. Thus, their compositional verification by the cvpp framework is complete.

We do not give a formal syntax and semantics of Boolean programs here. In fact, we use the standard syntax that is described in [62] and used by MOPED [45].

7.1 Program Model

As before, a program model is either the program’s structure or its behaviour.

7.1.1 Program Structure

Intuitively, to lift the cvpp framework to include Boolean data, we need the following extensions.

- The definition of method graph should be extended to capture the values of the Boolean variables at each program point, so that we capture data through control points. This can be achieved by extending the set of atomic
propositions with the names of all program variables and parameters. If a program point is tagged with a variable or parameter name, it means that the value of the variable is \texttt{true} in that program point and \texttt{false} otherwise.

- Methods should be called by passing Boolean arguments and returning Boolean values. This can be accomplished by modifying the set of labels such that the method invocations and returns include Boolean values for parameter and the return value, respectively. To return Boolean values, we introduce a reserved global variable \texttt{ret} and assign the return values to it.

Before giving a formal definition, let us illustrate the suggested program model on an example.

**Example 7.1.** Figure 7.1 shows a Boolean version of the Java program from Figure 3.1 and the Boolean flow graph it induces. Here, Boolean method graphs have two entry nodes, one for each value of the formal parameter \( n \).

Let us now adapt the formal definitions of the original framework to the program model that is described intuitively above. We remind that the basic definition of model is given in Definition 3.1. The definition of simulation is an instance of Definition 3.2 and Theorem 3.3 holds for Boolean program models as well. However, the method graph definition has to be adapted with the new Boolean variables.

We define \( \text{Bool} = \{ \text{true}, \text{false} \} \) as a set for Boolean values, and

\[
\text{Var}_m = \text{Glob} \cup \text{Loc}_m \cup \text{Param}_m
\]
as the set of variables of the graph of method \( m \) where \( \text{Glob} \) denotes the global variables of the program, \( \text{Loc}_m \) denotes the local variables of method \( m \) and \( \text{Param}_m \) denotes the parameters of method \( m \).

We also define a countably infinite set of signatures of methods as follows.

\[
\text{Meth} \ni m(x_1, \ldots, x_k): (l_1, \ldots, l_n) : t, \quad t \in \{ \text{void, bool} \}
\]
7.1. PROGRAM MODEL

where $m$ is a method name, Boolean variables $x_1, \ldots, x_k$ are the parameters, Boolean variables $l_1, \ldots, l_n$ are the local variables and $t$ is the return type of method $m$.

We let $\text{names}(M)$ for $M \subseteq \text{Meth}$ denote the set of method names in $M$.

**Definition 7.1 (Boolean Method graph).** A Boolean method graph for method $m(x_1, \ldots, x_k) : (l_1, \ldots, l_n) : t_m \in \text{Meth}$ over a set $M \subseteq \text{Meth}$ of method signatures is an initialized model $(M_m, E_m)$ where $M_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$ 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 $m$.

$$L_m = \{\varepsilon\} \cup \{m'(b_1, \ldots, b_k) : m' \in \text{names}(M), b_i \in \text{Bool}, k' \text{ arity of } m'\},$$

$$A_m = \{m, r, \text{ret}\} \cup \text{Var}_m, \text{ and } \lambda_m : V_m \rightarrow \mathcal{P}(A_m) \text{ so that } m \in \lambda_m(v) \text{ for all } v \in V_m. \text{ The nodes } v \in V_m \text{ with } r \in \lambda_m(v) \text{ are return points, variable } \text{var} \in \text{Var}_m \text{ is true if } \text{var} \in \lambda_m(v), \text{ and return value is true if } \text{ret} \in \lambda_m(v).$$

As before, every flow graph $G$ is equipped with an interface $I = (I^+, I^-)$, denoted $G : I$, where $I^+, I^- \subseteq \text{Meth}$ are the signatures of the provided and externally required methods, respectively.

7.1.2 Program Behaviour

The definition of behaviour has to be extended to capture method invocations with the values of parameters and return values. Also, the program points that are reached by a call action should be stacked in order to be used upon the return from the call. For the latter, we have decided to store a set of all possible program points and choose the correct one at the return from the call.

We define the behaviour of a closed Boolean flow graph as a labeled transition system. Transition label $\tau$ is used for internal transfer of control, label $m_1 \text{ call } m_2(b_1, \ldots, b_k)$ for the invocation of method $m_2$ by method $m_1$ by passing parameter values $(b_1, \ldots, b_k)$, $m_2 \text{ ret } m_1$ or $m_2 \text{ ret(b) } m_1$ for the corresponding return from the call. Transitions $m_2 \text{ ret } m_1$ and $m_2 \text{ ret(b) } m_1$ are used when the return type of method $m_2$ is $\text{void}$ or $\text{bool}$, respectively.

**Definition 7.2 (Behaviour of Boolean Flow Graph).** Let $G = (M, E) : (I^+, I^-)$ be a closed flow graph such that $M = (V, L, \rightarrow, A, \lambda)$. The behaviour of $G$ is defined as the initialized model $b(G) = (M_b, E_b)$, where $M_b = (S_b, L_b, \rightarrow_b, A_b, \lambda_b)$, such that $S_b = V \times \mathcal{P}(V)^+$, i.e., states (or configurations) are pairs of control points $v$ and stacks $\sigma$,

$$L_b = \left\{ \begin{array}{l}
    \{m_1 \text{ call } m_2(b_1, \ldots, b_k) : m_1 \in \text{names}(I^+), m_2 \in \text{names}(I^-), b_i \in \text{Bool}, k \text{ arity of } m_2\}
    \cup
    \{m_2 \text{ ret(b) } m_1 : m_1 \in \text{names}(I^+), m_2 \in \text{names}(I^-), b \in \text{Bool}, t_{m_2} = \text{bool} \}\cup
    \{m_2 \text{ ret(b) } m_1 : m_1 \in \text{names}(I^+), m_2 \in \text{names}(I^+), t_{m_2} = \text{void} \} \cup
    \{\tau\},
\end{array} \right.$$
Consider the Boolean flow graph from Figure 7.1. One example

$\lambda_0((v, \sigma)) = \lambda(v)$, and $\rightarrow_b \subseteq S_b \times L_b \times S_b$ is defined by the rules:

\[
\begin{align*}
\text{[transfer]} & \quad (v, \sigma) \xrightarrow{\tau}(v', \sigma) \quad \text{if} \quad m \in \text{names} (I^+), \quad v \xrightarrow{\tau} v', v \not= \tau \\
\text{[call]} & \quad \left( v_1, \sigma \right) \xrightarrow{\text{m}_1 \ \text{call} \ \text{m}_2 (b_1, \ldots, b_k)} \left( v_2, V_1, \sigma \right) \quad \text{if} \quad m_1, m_2 \in \text{names} (I^+), \quad V_1 = \{ v_1' \mid v_1 \xrightarrow{\text{m}_2 (b_1, \ldots, b_k)} v_1' \}, \quad v_1 \not= \tau, v_2 \not\in E, \quad \lambda(v_1) \cap \text{Glob} = \lambda(v_2) \cap \text{Glob}, \quad \forall i : 1 \leq i \leq k, \quad b_i \iff x_i \in \lambda(v_2) \\
\text{[retbool]} & \quad \left( v_2, V_1, \sigma \right) \xrightarrow{\text{m}_2 \ \text{ret} (b)} \left( v_1, \sigma \right) \quad \text{if} \quad m_1, m_2 \in \text{names} (I^+), \quad v_2 \not\in E, v_1 \not= \tau, v_1 \not\in V_2, \quad \lambda(v_1) \cap \text{Glob} = \lambda(v_2) \cap \text{Glob}, \quad \text{ret } \in \lambda(v_1) \iff \text{ret } \in \lambda(v_2), \quad b \iff \text{ret } \in \lambda(v_2), \quad t_{m_2} = \text{bool} \\
\text{[retvoid]} & \quad \left( v_2, V_1, \sigma \right) \xrightarrow{\text{m}_2 \ \text{ret} (\text{void})} \left( v_1, \sigma \right) \quad \text{if} \quad m_1, m_2 \in \text{names} (I^+), \quad v_2 \not\in E, v_1 \not= \tau, v_1 \not\in V_2, \quad \lambda(v_1) \cap \text{Glob} = \lambda(v_2) \cap \text{Glob}, \quad t_{m_2} = \text{void}
\end{align*}
\]

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

**Example 7.2.** Consider the Boolean flow graph from Figure 7.1. One example execution of the program, corresponding to an invocation of $\text{even} (\text{true})$ is:

\[
\begin{align*}
(v_0, \epsilon) \xrightarrow{\text{even} \ \text{call} \ \text{odd} (\text{false})} b(v_{12}, \{ v_1, v_3 \}) \xrightarrow{b} b(v_{13}, \{ v_1, v_3 \}) \xrightarrow{\text{odd} \ \text{ret} (\text{false}) \ \text{even}} b(v_2, \epsilon)
\end{align*}
\]

The control point following the call can only be $v_{12}$ since the value of the formal parameter $n$ has to agree with actual parameter $\text{false}$ of the call; the control point following the return can only be $v_1$ since it has to agree on $\text{ret}$ with the return value $\text{false}$.

The instantiation of the definition of simulation for behaviour $\leq_b$ is again:

$G_1 \leq_b G_2 \iff b(G_1) \leq b(G_2)$.

We show that structural simulation implies behavioural simulation.

**Theorem 7.3.** If $G_1 \leq_s G_2$ then $G_1 \leq_b G_2$.

**Proof.** Let $R_b$ be a structural simulation between $G_1$ and $G_2$. Define relation $R_b$ by (where $|\sigma|$ denotes the length of $\sigma$, and $\sigma(i)$ the $i^{th}$ element in $\sigma$):

\[
(v, \sigma)R_b (v', \sigma') \iff vRv' \land |\sigma| = |\sigma'| \land \forall i < |\sigma|, \forall w \in \sigma(i). \exists w' \in \sigma'(i). wRw'
\]

It is easy to check that \( R_b \) is a behavioural simulation between \( G_1 \) and \( G_2 \).

Simulation logic is as induced by Definition 3.4 on the sets of labels \( L_b \) and atomic propositions \( A_b \).

The definition of the logic is the same as Definition 3.4. Note that sets \( A_b \) and \( L_b \) include the atomic propositions and transition labels of Boolean program model, respectively.

**Example 7.3.** Consider again the flow graph in Figure 7.1. One example property is “if program execution starts in method even with value true for parameter \( n \), the first call can only be to method odd with value false for parameter \( n \)”. This property can be formulated in simulation logic as follows.

\[
even \land n \Rightarrow \nu X_1. \ [\text{even call even(true)]f}f \land \ [\text{even call even(false)]f}f \land \ [\text{even call odd(true)]f}f \land \ [\tau]X_1
\]

To simplify the specification we can present the above formula as follows.

\[
even \land n \Rightarrow \nu X_1. \left( \bigwedge_{b \in \text{Bool}} [\text{even call even}(b)]f \right) \land [\text{even call odd(true)]f}f \land [\tau]X_1
\]

### 7.2 Maximal Boolean Flow Graphs

Maximal models are constructed as explained in Section 3.3, but for the sets of atomic propositions and labels of Boolean program model. For structural formulas, the characterization formula should be adapted to restrict the constructed maximal Boolean flow graphs to the legal Boolean flow graphs. However, as before, for behavioural properties the maximal models are infinite-state and in general not unique. Therefore, behavioural properties should be translated to structural ones to be used for maximal Boolean flow graph construction. In this section, we define a set of atomic propositions and transition labels for Boolean flow graphs and adapt the characteristic formula. The translation of behavioural to structural properties fall into our future work.

#### 7.2.1 Maximal Model Construction

The definition of the auxiliary functions \( \chi \) and \( \theta \) remain the same as the definitions in Chapter 3. The Galois connection between finite specifications and formulas in simulation logic is also defined by these functions. We denote by \( \mathcal{L}(\mathcal{I}) \) the set of labels that can be produced from the set of method signature \( \mathcal{I} \subseteq \text{Meth} \), formally defined as follows.

\[
\mathcal{L}(\mathcal{I}) = \{ m'(b_1, \ldots, b_{k'}) \mid m' \in \text{names}(\mathcal{I}), b_i \in \text{Bool}, k' \text{ arity of } m' \}
\]
Also, we denote by $\mathcal{A}(I)$ the set of atomic propositions of method graphs of $I$, formally

$$\mathcal{A}(I) = \text{names}(I) \cup \{\text{ret}, r\} \cup \text{Global} \cup \left( \bigcup_{m \in \text{names}(I)} (\text{Local}_m \cup \text{Param}_m) \right)^1.$$

The transformation to SNF is defined for any set of labels and any set of atomic propositions and literals, therefore for the new set of labels and atomic propositions, the SNF transformation works as it was defined for the model in absence of data.

### 7.2.2 Maximal Boolean Flow Graph Construction

The characteristic formula for the Boolean flow graphs is defined as follows.

$$\sigma_I = \bigvee_{m \in \text{names}(I^+)} (\nu X \cdot P_m \land V_m \land [\mathcal{L}(I^-), \epsilon] X)$$

where,

- $P_m = m \land \bigwedge_{m' \in \text{names}(I^+) \setminus \{m\}} \neg m'$
- $V_m = \forall m' \in \text{names}(I^+) \setminus \{m\}. \bigwedge_{v \in \text{Local}_m \cup \text{Param}_m} \neg v$

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

$$\begin{align*}
\sigma_I &= \bigvee_{m \in \text{names}(I^+)} X_m \\
\Sigma_I &= \{X_m = [\mathcal{L}(I^-), \epsilon] X_m \land P_m \land V_m \land m \in \text{names}(I^+)\}
\end{align*}$$

where $P_m$ and $V_m$ are defined above.

The following theorem proves that the characteristic formula for interface $I$ characterizes the Boolean flow graphs with interface $I$.

**Theorem 7.4.** Let $I$ be an interface for Boolean flow graphs. For any initialized model $S = (M, E)$ over labels $L = \mathcal{L}(I^-) \cup \{\epsilon\}$ and atomic propositions $A = \mathcal{A}(I^+)$, we have

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

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

The proof is similar to the one of Theorem 32 in [34].

As before, maximal Boolean flow graph of a module is constructed on the conjunction of the structural property and an instantiation of the above characteristic formula.

---

1The global and local variables should be provided by a declaration similar to interfaces or should be extracted automatically from the flow graphs of $I$. 

---
7.3 Compositional Verification

The maximal flow graph constructed for a structural property $\phi$ and interface $I$ as explained above, is a valid Boolean flow graph that satisfies the property $\phi$ and interface $I$, and simulates all Boolean flow graphs satisfying them. So, the compositional verification principle 3.1 can be directly used to verify properties of Boolean flow graphs. However, Principle 3.2 requires a translation from behavioural to structural properties of Boolean programs. As mentioned, this will fall into our future work.

7.4 Examples

In this section, we present two examples to illustrate the use of Boolean flow graphs for verification of Boolean programs. In the first example, we focus on the global property and the local specifications, while in the second example we present the maximal Boolean flow graph construction of one of the components as well. In both of the examples, we use simulation logic as specification language because it is the internal language of the cvpp framework. Moreover, we use procedure-modular verification and specify each of the procedures by a local specification containing a property and an interface. Our focus is on the global relativization. For the local checks, the framework is based on external model checkers and therefore, this part is not our focus here.

7.4.1 Use-After-Ask Example

In this section, we illustrate the use of the compositional verification technique described above on the Boolean program shown in Figure 7.2. The interface of the program is

$$I = \{\{\text{main()}, \text{use}(), \text{ask}())\}, \{\text{main()}, \text{use}(), \text{ask}())\}.\]$$

It is closed and the program does not require any external methods. The program has no global variables. Function $\text{main}$ has a local variable $r$. In $\text{main}$, first, $\text{ask}$ is called before calling the critical method $\text{use}$ which accesses some critical information. In the program syntax, $F$ and $T$ are representing the Boolean values $\text{false}$ and $\text{true}$, respectively. We are interested in verifying the property “if program execution starts in method $\text{main}$, method $\text{use}$ can only be called if and after method $\text{ask}$ is called by $\text{main}$ and returns $\text{true}$. This global property can be formalized in
simulation logic as follows:

\[
\phi_{\text{glob}} = \text{main} \Rightarrow \nu X. \left( \bigwedge_{m \in \text{names}(I^+)} [m \text{ call use()}\text{ff}] \right) \land \left( \bigwedge_{m \in \text{names}(I^+)} [m \text{ call main()}\text{ff}] \right) \land \\
[\text{main call ask()}X \land \\
[\text{ask ret(false)}\text{main}]X \land \\
[\tau]X.
\]

In this global property, we prohibit calls to methods `use()` and `main()` until a call to method `ask()` returns with `true` value.

**Method use.** This method has no local variables and no parameters. The required interface of this method is empty: \( I_{\text{use}} = \{\{\text{use()}\}, \{\} \} \). The property of this method is that it does not invoke any method, which is formalized as follows.

\[
\psi_{\text{use}} = \nu X. [\text{use()}\text{ff}] \land [\varepsilon]X
\]

**Method ask.** Like Method `use`, this method also has no local variables and no parameters. The required interface of this method is empty: \( I_{\text{ask}} = \{\{\text{ask()}\}, \{\} \} \).
7.4. EXAMPLES

The property of this method is that also it does not invoke any method.

\[ \psi_{\text{ask}} = \nu X. [\text{ask}()] \text{ff} \land [e] X \]

**Method main.** Unlike Method use and method ask, this method has a local variable \((r)\) but no parameters. The interface of this method is: \( I_{\text{main}} = \{\{\text{main}()\}, \{\text{use}()\}, \{\text{ask}()\}\} \). The property of this method is that method use will be called only if method ask is called and returns \text{true}. To formalize this property in structural simulation logic, we can prohibit the calls to methods use() and self-calls to main() and after a call to method ask(), if the value of ret is true then calling method use() is possible; otherwise, no other method can be called.

\[ \psi_{\text{main}} = \nu X. [\text{main}()] \text{ff} \land [\text{use}()] \text{ff} \land [e] X \land [\text{ask}()] \phi \]

where \( \phi \) is:

\[ \phi = \left( \begin{array}{c} \text{ret} \Rightarrow \nu Y. [\text{ask}()] \text{ff} \land [\text{main}()] \text{ff} \land [e] Y \land [\text{use}()] \text{ff} \land [\text{ask}()] \text{ff} \land [\text{main}()] \text{ff} \land [e] Y \\ \neg \text{ret} \Rightarrow \nu Y. [\text{use}()] \text{ff} \land [\text{ask}()] \text{ff} \land [\text{main}()] \text{ff} \land [e] Y \end{array} \right) \]

(Notice that the local property can be specified without mentioning local variable \(r\).)

In this property, we prohibit the calls to methods main() and use(). And after a call to method ask(), if in method main the value of ret is true we prohibit all calls except a call to method use(), while if ret has false value in main(), we prohibit all calls.

Notice that since the property is structural and the provided interface only contains method main(), after a call to a method (e.g., ask()) the control stays in method main(). Therefore, it is accurate to check ret exactly after the call.

**Compositional Verification.** As mentioned above, local checks are performed with external tools and therefore we are not focusing on them here. For global relativization, after constructing maximal Boolean flow graphs for all methods, we compose these and model check the resulting Boolean flow graph against \( \phi_{\text{glob}} \). The result of the model checking will be the result of the verification.

7.4.2 Lock Example

As a second example, consider the Boolean program that is shown in Figure 7.3. This example is inspired by the lock example provided by the MOPED distribution and explained in [62]. The interface of the program is

\[ I = \{\{\text{main}(), \text{lock}(), \text{unlock}(), g(b)\}, \{\text{main}(), \text{lock}(), \text{unlock}(), g(b)\}\} \]

As it is clear from the interface, the interface is closed and the program does not require any external methods. The program has global variables \(l\) and \(b\). Variable \(l\) is used to specify that the program has exclusive access (lock) to variable \(b\) (which
here models a critical resource. Functions lock and unlock are to acquire or release access to the critical section, respectively. Function g takes one Boolean argument and returns the negation of the given value. Function main has a local variable a. In main, the lock is acquired before assigning value to the critical variable b, and after the assignment is performed the lock is released. In the program syntax F and T are representing false and true Boolean values, respectively.

Our task is to prove that a behavioural global property holds for the program, given the local specifications of the components of the program. The global property we use for this example is: “if program execution starts in method main, method g is only called when global variable l is true”. Since the global variable l is the lock in this example, the proposed global property essentially states that the lock has to be obtained before method g is called.

Then the global property can be formalized in behavioural simulation logic as follows:
7.4. EXAMPLES

\[ \phi_{\text{glob}} = \text{main} \Rightarrow \nu X. \left( \bigwedge_{m \in \text{names}(I^+)} \bigwedge_{b \in \text{Bool}} [\text{call } g(b)] l \right) \land \left( \bigwedge_{m \in \text{names}(I^+)} \bigwedge_{m' \in L} \left[ [m \text{ call } m'] X \right] \land [r] X \right) \]

where \( L = L(I^- \setminus \{\text{sig} \mid \text{sig} \in I^- \land \text{names}(\{\text{sig}\}) = \{g\}) \).

In the above property, the first conjunct captures that calls to method \( g \) are prohibited when \( l \) is false. We express this by capturing that the value of variable \( l \) is true after method \( g \) is called \( ([x \text{ call } g(y)] l) \). This complies the global property because the call action preserves the value of the global properties.

**Method lock.** The interface of method lock is \( I_{\text{lock}} = \{\{\text{lock}\}, \{\}\} \) and the local property of method lock is: “method lock does not call any method and at its return point, global variable \( l \) has value true.” This property can be formalized in structural simulation logic as follows:

\[ \psi_{\text{lock}} = \nu X. (r \Rightarrow 1) \land [\text{lock}()] \land [e] X \]

The local property is accurate because method lock does not have any self-calls, and if global variable \( l \) is false then it is changed to true, otherwise the value of global variable \( l \) is not changed.

**Method unlock.** The interface of method unlock is \( I_{\text{unlock}} = \{\{\text{unlock}\}, \{\}\} \) and the local property of method unlock is: “method unlock does not call any method and at its return point, global variable \( l \) has value false” which can be formalized as follows:

\[ \psi_{\text{unlock}} = \nu X. (r \Rightarrow \neg 1) \land [\text{unlock}()] \land [e] X \]

This local property is also accurate because method unlock does not have any self-calls, and if global variable \( l \) is true then it is changed to false, otherwise the value of global variable \( l \) is not changed.

**Method g.** The interface of method \( g \) is \( I_g = \{\{g(b)\}, \{\}\} \) and the local property of method \( g \) is: “no self-calls and if the parameter \( x \) is true, return false and vice versa” which can be formalized as follows:
ψ_g = \left( x \Rightarrow \nu X. (r \Rightarrow \neg \text{ret}) \land \left( \bigwedge_{b \in \text{Bool}} [g(b)] \land [\varepsilon] X \right) \land \neg x \Rightarrow \nu X. (r \Rightarrow \text{ret}) \land \left( \bigwedge_{b \in \text{Bool}} [g(b)] \land [\varepsilon] X \right) \right)

Method main. The interface of method main is \( I_{\text{main}} = \{ \{ \text{main}() \}, \{ \text{lock}(), \text{unlock}(), g(b) \} \} \). We specify a local property for method main that captures the complete structure of the method. We do this to give readers a view on how to specify different properties in simulation logic\(^2\). The local property of method main is: "no function can be called until global variable \( l \) and local variable \( a \) are false. Afterward, method lock is called and after returning from the call, method \( g \) is called with parameter \( a \) (which has value false) and the return value from the call is stored in global variable \( b \). Then method unlock is called and no other method can be called.". This property can be formalized as follows:

\[
\psi_{\text{main}} = \left( \begin{array}{c}
\nu X_1. \left( \bigwedge_{y \in L'} [\text{ff}] \land [\varepsilon] X_1 \land ((\neg 1 \land \neg a) \Rightarrow \right.
\nu X_2. \left( \bigwedge_{z \in L' \cup \{ \text{lock()} \} \setminus \{ \text{sig} \} \}} \land [\varepsilon] X_2 \land \right.
\nu X_3. \left( \bigwedge_{y' \in L' \cup \{ \text{false} \} \setminus \{ \text{sig} \} \}} \land [\varepsilon] X_3 \land \right.
\nu X_4. \left( \bigwedge_{y'' \in L' \cup \{ \text{true} \} \setminus \{ \text{sig} \} \}} \land [\varepsilon] X_4 \land \right.
\nu X_5. \left( \bigwedge_{y'''} \land [\varepsilon] X_5 \land \right.
\nu X_6. \left( \bigwedge_{y''''} \land [\varepsilon] X_6 \land \right.
\nu X_7. \left( \bigwedge_{y'''''} \land [\varepsilon] X_7 \right) \right)
\right)
\right)
\]

where \( y \in L(I^-) \), \( z \in L(I^- \setminus \{ \text{lock}() \}) \), \( z' \in L(I^- \setminus \{ \text{true} \}) \), \( z'' \in L(I^- \setminus \{ \text{false} \}) \), \( z''' \in L(I^- \setminus \{ \text{true} \}) \), \( z''''' \in L(I^- \setminus \{ \text{false} \}) \).

After specifying all local interfaces and properties, maximal flow graphs are constructed for each of the methods from their local interfaces and properties. Here, we will illustrate the construction of maximal flow graph of method lock only. The maximal flow graphs of the remaining methods are constructed in a similar way.

Characteristic Formula. As a first step for constructing a maximal Boolean flow graph for method lock, the following characteristic formula form interface

\(^2\)But this is not necessary at all. In practice properties can be specified by abstracting away immaterial variables and method invocation orders to prove the global property. This also allows more convenient evolution of the code.
7.4. EXAMPLES

$I_{\text{lock}}$ is constructed.

$$(\sigma_{I_{\text{lock}}}) = \nu X. \ \text{lock} \land \neg x \land \neg a \land [\varepsilon]X$$

The above formula is converted to the following modal equation system.

$$(\sigma_{I_{\text{lock}}}) = X \ [X = \text{lock} \land \neg x \land \neg a \land [\varepsilon]X]$$

Construction of Maximal Flow Graph. The conjunction of characteristic formula $\sigma_{I_{\text{lock}}}$ and $\psi_{\text{lock}}$ is transformed to SNF. The result after transformation is $(A \lor B \lor C \lor D \lor E)[\phi]$ modal equation system where $\phi$ is the following equation system.

$$\phi = \begin{cases} 
A = ([\varepsilon](A \lor B \lor C) \land [\text{lock}]ff \land \neg r \land \text{lock} \land p) \\
B = ([\varepsilon](A \lor B \lor C) \land [\text{lock}]ff \land r \land \text{lock} \land 1 \land p) \\
C = ([\varepsilon](A \lor B \lor C) \land [\text{lock}]ff \land \neg r \land \text{lock} \land 1 \land p) \\
D = ([\varepsilon](A \lor B \lor C \lor D \lor E) \land [\text{lock}]ff \land r \land \text{lock} \land 1 \land p) \\
E = ([\varepsilon](A \lor B \lor C \lor D \lor E) \land [\text{lock}]ff \land \neg r \land \text{lock} \land 1 \land p) 
\end{cases}$$

where $p$ abbreviates $\neg b \land \neg a \land \neg x$. In fact, $\phi$ also includes all the equations above for true values of variable $b$. But here we just consider false values for global variable $b$ to have a smaller equation system and maximal flow graph and simplify the understanding of the maximal flow graph construction process.

By using function $\chi$ defined in Definition 3.10, the above modal equation system is represented as the Boolean flow graph shown in Figure 7.4 which is the maximal Boolean flow graph of method $\text{lock}$. In the figure, states that are tagged with atomic proposition $r$, are also tagged with atomic proposition 1. This shows that in every return point of the Boolean flow graph, the value of variable 1 is true.

Compositional Verification. After constructing maximal Boolean flow graphs for all other methods, we compose these and model check the resulting maximal flow graph against $\Phi_{\text{glob}}$. The result of the model checking will be the result of the verification. Notice that the focus is on the global relativization and not on the local checks.

![Figure 7.4: Maximal Boolean flow graph of method lock](image-url)
7.5 Conclusion

As a first step towards adding data to the CVPP framework, we suggested an extension of the program model with Boolean variables as a new class of atomic propositions associated with control points of the program model. Also the labels of the program model are changed to be able to express method invocations by passing parameter values. All the relevant definitions and theorems are adapted from the original set-up and (if needed) proved with the new definitions of the program model. To evaluate the suggested extensions, two examples are described and verified step-by-step.

In this chapter, we show that data from finite domains can be encoded through control. The advantage is that the CVPP framework can be easily adapted and used. However, the drawback is that there is no direct correspondence between control points of a program and its flow graph. This makes the flow graph less understandable. Moreover, by capturing concrete data values, the number of the control points grow quickly and we face the state-space explosion problem. An alternative approach of capturing data is to capture data symbolically. We have already started to investigate the encoding of symbolic data into the program model.
Chapter 8

Conclusion

The detailed conclusions and future work are given in the respective chapters. Here, we present some general discussion and directions for future work.

8.1 Discussion

This thesis provides an overview of the CVPP compositional verification framework based on maximal flow graph construction and its tool set. In the framework, the structure of programs are modelled by flow graphs. These represent the (over-approximated) structure of a program by abstracting away all its data. The framework is developed for verification of open systems, and can be used for modular verification and non-compositional verification as well.

The CVPP framework brings various advantages. It is modular, thus can deal with static and dynamic variability, and is in general sound and complete at the flow graph level. However, these advantages come at the price of having to provide local specifications, computationally expensive maximal flow graph construction and PDA model checking, and a limited range of properties because of data abstraction.

We introduce ProMoVer which is a tool to automate procedure-modular verification with CVPP. It brings the whole compositional verification framework into practical usage. It is easy to use in the sense that it requires a single input file including annotations. Moreover, it supports verification in presence of variability, and therefore can be used in code evaluation and customization scenarios such as open systems, mobile code, and software families.

ProMoVer provides pragmatic solutions for the limitations of the compositional verification framework. It is equipped with the specification extractor which extracts candidate specifications from the implementation of methods. By using the specification extractor, it is possible to verify large programs with minimal manual effort. In addition, ProMoVer’s proof storage and reuse mechanism minimizes the use of the flow graph extraction, maximal flow graph construction, and PDA model checking.
As a particular application area, the modular verification of software families is studied, and a novel hierarchical variability model is proposed. ProMoVer is adapted and evaluated by a case study. It is 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 modular verification method.

Experiments with ProMoVer show that procedure-modular verification of temporal safety properties can be done automatically, and temporal logic provides a meaningful abstraction for individual program methods.

The main restriction of the cvpp framework is the abstraction from all program data. This restriction is alleviated by theoretically extending the definitions of flow graph, program behaviour, and specification languages to include Boolean values. Also the maximal flow graph construction is adapted accordingly. It is shown that most of the previous definitions and theorems are conveniently adapted for the Boolean framework as well. By this extension, a wider range of properties is supported.

To easily adapt the cvpp framework, the Boolean values are encoded through the control points of flow graphs. As a result, flow graphs are less understandable and significantly larger. To resolve these, data should be encoded symbolically.

8.2 Future Work

We believe graphical specifications, such as automata based languages, are more convenient than LTL and μ-calculus to express some properties. Such specification languages may also allow more efficient maximal flow graph construction. We are currently working on providing support for security automata as a specification language. Our ultimate goal is to support (local and global) specifications written in various temporal logics and notions, or to use patterns to abbreviate common specification idioms.

For product families, our plan is to enrich SHVM with optional variants and constraints and rules between variants in order to facilitate the direct verification of more expressive hierarchical variability models. Also, in order to further evaluate our approach, we plan to perform more and larger case studies.

Work has begun on extending ProMoVer with the support for Boolean programs. Moreover, we are currently investigating the extensions of the verification framework with symbolic data. Our plan is to generalize our method for the program model of Rot et al. that models object reference in the presence of unbounded object creation [60].
Bibliography


