UNIVERSITÄT
PASSAU
Fakultät für Informatik und Mathematik

Analysis Strategies for Configurable Systems

Alexander von Rhein

June 9, 2016

Dissertation zur Erlangung des Doktorgrades
der Naturwissenschaften (Dr. rer. nat.)
eingereicht an der Fakultät für Informatik und Mathematik
der Universität Passau

Dissertation submitted to
the Department of Informatics and Mathematics of
the University of Passau
in Partial Fulfillment of Obtaining
the Degree of a Doctor in the Domain of Science

Betreuer / Advisor:
Prof. Dr. Sven Apel

Externe Gutachterin / External Examiner:
Prof. Dr. Stefania Gnesi

Die Dissertation wurde eingereicht am:
09.11.2015

The dissertation was submitted on:
12/09/2015

Unterschrift / Signature:

Alexander von Rhein

Abstract

A configurable system enables users to derive individual system variants based on a selection of configuration options. To cope with the often huge number of possible configurations, several analysis approaches (e.g., for verification of configurable systems) implement different strategies to account for configurability.

One popular strategy—often applied in practice—is to use sampling (i.e., analyzing only a subset of all system variants). While sampling reduces the analysis effort significantly, the information obtained is necessarily incomplete as some variants are not analyzed. A second strategy is to identify the common parts and the variable parts of a configurable system and analyze each part separately (called feature-based strategy). As a third strategy, researchers have begun to develop family-based analyses. Family-based approaches analyze the code base of a configurable system as a whole, rather than the individual variants or parts of the system, this way exploiting similarities among individual variants to reduce analysis effort. Each of these three strategies has advantages and disadvantages, which might even prevent its application (e.g., the family-based strategy typically needs much main memory).

The goal of this thesis is to enable the efficient analysis of configuable systems, even if existing strategies fail (e.g., the family-based strategy, because of memory limitations). To this end, we designed a framework that models the key aspects of configurable-system analysis strategies, independent of their implementation and of the analyses techniques (e.g., type checking or model checking). Guided by our model, we developed a number of analysis strategies for configurable systems. To learn about advantages and disadvantages of individual strategies, we compared these in a series of empirical studies.

In particular, we developed and evaluated a model-checking analysis and a data-flow analysis for configurable systems. One of our key findings is that family-based analysis outperforms most sampling heuristics with respect to analysis time, while being able to make definite statements about all variants of a configurable system. Furthermore, we identified advantages and disadvantages of analysis strategies and how to mitigate them by combining strategies.

In our endeavor, we identified two key problems that are common to configurable-system analyses, and we developed supporting techniques to solve them. These techniques are general and are applicable beyond our research. In particular, we developed presence-condition simplification and variability encoding. Presence-condition simplification provides a simple method to reduce the size of the output or the internal data structure of configurable-system analyses. Variability encoding provides a means for transforming compile-time variability to run-time variability, which enables many family-based analyses.

Our key contributions are the model of analysis strategies for configurable systems and the corresponding empirical comparisons of strategies. Our findings are backed by empirical studies, which helped broaden the community knowledge on analyses of configurable systems (indicated by citations). For these evaluations, we prepared several subject systems, which have also been used already by other researchers. Furthermore, we developed several analysis tools and demonstrated their feasibility in practical application scenarios based on code from, for example, the Linux kernel. Our tools are based on variability-aware optimizations that enable levels of scalability on configurable systems that were not possible with other tools before.

Acknowledgements

Writing a dissertation in computer science is a long and arduous endeavor, which requires the help of many persons. I am deeply appreciative of the many individuals who have supported my work and continually encouraged me through the writing of this dissertation.

Foremost, I would like to express my sincere gratitude to my advisor Sven Apel for his patience, motivation, and immense knowledge. Without his guidance the research and writing of this thesis would not have been possible. I cannot imagine having a better advisor and mentor for my dissertation.

I would also like to thank Stefania Gnesi, who agreed to review my thesis, although she knew it would be more than 200 pages to read. Stefania also introduced me to the difference between a bisimulation proof and a trace-equivalence proof.

In the same way, I want to thank Thorsten Berger, Christian Kästner and Thomas Thüm. We collaborated on several research papers and our discussions helped me understand the bigger picture of my work, directed us to new research questions, and improved my scientific writing. I want to thank my fellow students Kathrin, Olaf, Philipp, and Severin who accompanied me since I enrolled at the university for the first time.

I want to thank my working group at Passau for a wonderful time. Especially, I want to thank Sergiy Kolesnikov, for many discussions about language grammar and presentation of results. We shared our office during my 5 years of doctoral study and had a lot of recreational trips to the ice cream vending machine in that time. I want to thank my friends and colleagues for proofreading my thesis and for general support: Alex Grebhahn, Jörg Liebig, Stefan Löwe, Norbert Siegmund, Andreas Simbürger, and Andreas Stahlbauer. Furthermore, I want to thank Eva Reichhart for her invaluable administrative support.

A huge part of the tool support for this thesis has been implemented by student assistants. In particular, I want to thank Florian Garbe for his tireless work on Hercules, Andreas Buchecker for his experiments with various model-checking strategies, and Niklas Schalck Johansson and Mikael Mark Hardø for the implementation of Sifta.

Last but not least, I want to thank my family. My grandmothers Anni and Orla cooked delicious bavarian meals once a week and made me feel at home in Passau. My parents Elisabeth and Volker, my sister Nadja and my partner Melli shared happy moments and motivated me when times were bad (especially after rejected papers).

Contents

List of Figures

List of Algorithms

List of Tables

List of Abbreviations

ASE International Conference on Automated Software Engineering

BDD Binary Decision Diagram

HVC Haifa Verification Conference

ICSE International Conference on Software Engineering

JLAMP Journal of Logical and Algebraic Methods in Programming

PLA cube Product-Line–Analysis Cube

PLA model Product-Line–Analysis Model

VAMOS International Workshop on Variability Modelling of Software-Intensive Systems

CHAPTER 1

Introduction

1.1 Problem and Motivation

Software plays an essential role in the modern world. National infrastructure, transportation, communication, finance, and entertainment depend on reliable software. Even though software is often invisible, it has real effects on our daily life, especially if a system does not work as expected.

In the 1960s, it became clear that large-scale software must be developed by teams rather than individuals and in a structured process [NR69]. Since then many *software-engineering* techniques and processes, such as the waterfall model and scrum, have been introduced to keep up with the increasing demand for more complex and powerful systems.

The key challenges for software engineering are the increasing diversity of application domains, demands for more reliable software, and reduced development times [Som10]. For example, the LINUX operating system (LINUX kernel)[1] is used in many different application scenarios, from embedded systems in cars and planes to servers and desktop computers. This diversity of application scenarios requires not only a single variant of the system, but a *family* of many different LINUX kernel variants, which must be developed and maintained simultaneously. Supporting many variants of a system is usually in conflict with reducing development time (and cost) and with providing a high level of system reliability.

[1] `https://www.kernel.org/`

Software product lines (or more general, *configurable systems*) address these key challenges by reusing software artifacts across system variants. Configurable systems provide dedicated configuration options—corresponding to system *features*—that control aspects of the system behavior and functionality. The configuration options of a configurable system control variability in the system implementation. To build a system variant, a user chooses one value for each configuration option, corresponding to the desired application scenario and, thereby, resolves the configuration-related variability in the system. Configurable systems often have huge numbers of variants, which leads to scalability problems when a developer wants to assure a property for every variant (e.g., all variants implement a given specification correctly). For this thesis, we developed, evaluated, and compared different approaches for configurable-system analysis, and we identified strategies for efficient analyses for highly configurable systems. Our goal is to enable the efficient analysis of highly configurable software systems.

As a motivating example for the practical use of configurable systems, we discuss how configurable systems are used in practice. Several well known companies, such as Eurocopter, Bosch, Nokia, Philips, and Siemens have developed and used configurable systems successfully [DH09; vLSR07; WSA$^+$15]. Often configurable systems are implemented in domains with rich variability, such as avionics, automotive, financial software, mobile phones, medical systems, and network management. Dordowsky and Hipp [DH09] described the development of variants of the NH90 helicopter. At the time of their writing, the helicopter was produced in 23 variants for 14 nations. Each customer had her own requirements, which were modelled as sets of features. One important project limitation was the non-disclosure requirement between customers. A feature that is developed for one customer must not be included in the package delivered to other customers. This implies that one cannot just pack all features and select dynamically which should be used. Instead, the variants of the system must contain exactly the requested functionality.

the NH90 system

the Linux kernel

As another example of a highly configurable system, we discuss the LINUX kernel in more detail. Maintained by over 7800 individual engineers from almost 800 different companies, it is one of the largest and most active open-source software projects in existence [CKM12; SSSS07]. It has been in development for over 20 years and it is highly configurable. The number of configuration options in the LINUX kernel has risen from 4752 (June 2005) to 13 165 (April 2013) [PPB$^+$15]. The options allow users to configure variants of the kernel according to their application scenario. For example, the kernel supports more than 60 different hardware platforms. Furthermore, the kernel contains subsystems, such as network protocols and file systems, which can be configured by the user. Finally, the biggest subsystem contains the drivers, which allow the kernel to support a vast amount of different hardware devices.

feature-interaction problem

The huge number of variants in configurable systems, such as the NH90 or LINUX, leads to defects that occur only in some of the variants and are therefore hard to detect. To avoid such configuration-dependent defects, configurable-system developers and designers spend much effort to find and document feature dependencies [ABKS13]. For example, in the NH90, choosing an advanced navigation computer might require a bus system that is able to handle higher data throughput. If dependencies between features have not been documented, the features can still interact and change system behavior in potentially undesired ways. A *feature interaction* between two features causes a changed, possibly unexpected or undesired system behavior (behavioral defect) [ABKS13]. The behavior can be observed only in variants in which both features are present, and it cannot easily be deduced from the behaviors of the individual features involved. If only one or none of the features is present, the interaction behavior does not arise. For example, an intelligent door-locking feature in a car might well interact with a feature that controls lighting and with another feature that detects and responds to a car crash. Ideally, the safety feature (unlock doors after crash) should take precedence over the security feature (door locking). But if this interaction is publicly known, a thief might intentionally hit a parking car to trigger the crash sensor and unlock the doors [Dom12]. So, the developers must either develop a dedicated routine that deals with this specific interaction or they must forbid that the crash sensor and door-locking features are chosen together. In general, it is often difficult to predict how features will interact and how to resolve detected feature interactions.

analysis challenges

The car example illustrates that feature interactions can cause misbehavior. In software systems, feature interactions can cause a whole range of problems, including compile errors (e.g., due to wrong types), undesired run-time behavior, or even performance bugs. Each interaction might occur in only a few variants, which makes the interaction hard to detect. To detect and fix such problems, developers in practice use *software-analysis* techniques, such as type checking and model checking. For example, the NH90 system is based on safety-critical avionics software; they must be qualified against many strict regulations concerning safety, security, and verification [WSA^{+}15]. There are documents that guide certification authorities on the approval of safety-critical airborne software (e.g., the RTCA DO-178B standard [DO-178B]). One recommendation in these guidelines is that dead code (code that can never be reached in any execution) is required to be completely removed [DBT11]. There are well-known techniques to eliminate dead code in traditional single-variant programs [ALSU06]. However, in a configurable system, code might be dead only in some variants [Tar13], which requires specialized analyses that eliminate code only from those variants in which it is unreachable. Other requirements in the NH90 are that the systems must not crash due to memory exhaustion and that subsystems meet their real-time deadlines [DBT11]. Such requirements usually need to be proved with automatic verification techniques (e.g., model checking or theorem proving).

There are several strategies to apply standard software-analysis techniques to configurable systems. A naive approach is to apply traditional analyses by configuring and building every variant and analyzing the variants in isolation (*variant-based* strategy). Each variant is a program without variability, so applying this strategy is quite simple, since it requires no modification of the existing analysis tools. However, a real-world configurable system may have a high number of variants. A system with 33 optional, independent features has more variants than there are humans on our planet. Generating all these variants is infeasible, not to mention analyzing them. In large systems, such as the Linux kernel, with thousands of features, it is even difficult to compute how many valid configurations the system has [Lie15].

variability-aware analysis

Consequently, applying traditional analyses to each variant of a configurable system does not scale. However, there are properties specific to configurable systems that help us to avoid a full analysis of all variants. For example, many parts of the code are shared among many variants. This sharing is caused by the very goal of configurable-system engineering: reuse as many artifacts as possible between variants to reduce development effort. This goal typically leads to many variants of configurable systems that share substantial common parts. The *family-based* approach, analyzes the entire system family at once, which often implies that common parts are analyzed only once. Usually, *family-based* analyses are *variability-aware*, which means that the implementation is optimized to exploit common parts and reuse analysis results for these parts across the analysis of individual variants. We have developed this idea in various *variability-aware family-based* approaches [ASW+11; AvRW+13; LvRK+13].

Another approach is to reduce the number of analyzed configurations using a sampling strategy: a traditional (not variability-aware) analysis runs on a selected, representative subset of configurations. The crux is how to select a good sample set. It must be small enough to avoid the combinational explosion of the naive strategy (i.e., too many variants to analyze), but large enough to cover as many different use cases as possible. Depending on the goal of the analysis and the system (e.g., Linux) one might want to use implementation knowledge and, for example, cover all platform variants (e.g., x86 and arm), at least, once.

Yet another strategy is to analyze every feature in isolation [LKF02a; TAK+14]. Such *feature-based* analyses can be applied even if only part of the system is known (in an open-world scenario). However, feature-based analyses can detect only defects that are caused by individual features, not by interactions. We develop an analysis approach that mitigates this disadvantage by combining the feature-based and family-based strategies.

thesis goal and focus

Overall, our goal is to enable the efficient analysis of highly configurable software systems. In particular, we address the question of how program analyses (e.g., data-flow analysis and model checking) can be applied to configurable software systems. We developed, evaluated, and compared different approaches for configurable-system analysis. We focus on the analysis of software implementation artifacts with automatic tools as opposed to manual analysis, analysis of feature-design documentation, or analysis of hardware–software systems such as NH-90. Compared to analysis of design documentation (e.g., UML diagrams), analysis of the final implementation artifacts (e.g., source code) has the potential for reliable statements about the final variants of a system that are delivered to end-users. Furthermore, we focus on software implementation artifacts, because systems with configurable hardware (e.g., cars) are very expensive to analyze. One either needs many different variants of the hardware or a configurable simulator of the hardware [LALL09; OSC+14].

1.2 Contributions

In this thesis, we present different contributions concerning the analysis of configurable systems. Most contributions have been published in conferences, journals or workshops that are considered first-grade in our field of research (such as ICSE, ESEC/FSE, or ASE). Our contributions contain technical insights, practical tool implementations, empirical evaluations, formalisms, and several subject systems used in experiments. Our techniques have already been reused in related publications and tools [BLB+15; KvRE+12; Mei14]. Analysis tools used in practice could be improved based on our technical contributions and experiment results. In this section, we give a short overview of the main contributions.

- We present the **product-line analysis (PLA) model** [vRAK+13], a model that covers the spectrum of configurable-system analysis. It describes basic analysis strategies for configurable systems, and it guides developers to combine these strategies to derive more efficient ones. Based on the PLA model, we proposed and implemented several different analyses for configurable systems and we discuss them in the context of the PLA model. We demonstrate the usefulness of the model by taking a closer look at existing analysis approaches and by classifying them according to our model. In the light of our model, key ideas that make these approaches efficient become apparent. We further used the model as a driver to devise new analysis strategies, which we discuss in this thesis.

- Our technical contributions include two fundamental techniques: presence-condition simplification and variability encoding. **Presence-condition simplification** [vRGA$^+$15] is a method for reducing the size of expressions (*presence conditions*) that denote when a certain artifact is present in a variant. The method can improve variability representation in source code, in reports from analysis tools, and in data structures of analysis and transformation tools. We formally describe presence-condition simplification and refer to existing algorithms from different researchers. We evaluated these algorithms in different application scenarios, such as simplification of presence conditions in internal data structures in the tool TypeChef.[2] The results of our evaluation show that presence-condition simplification can be used to improve tool output and data structures. Based on our work, presence-condition simplification has been integrated in the main branch of TypeChef.
- **Variability encoding** is a technique that encodes the compile-time variability of a configurable system in terms of load-time variability in a corresponding *variant simulator*. We use variant simulators to efficiently run variability-aware analyses (e.g., Chapter 6). In a simulator, configuration options are encoded as global variables that can be set at program start. Variability encoding ensures *behavior preservation*. This means that the behavior of a variant simulator and the behavior of a variant derived at compile time are equivalent if the global variables of the simulator and the configuration options of the variant are set to the values. Variability encoding enables us to change the binding time of configuration options from compile time to load time. This is relevant in practice as similar approaches are used in industrial contexts (e.g., in the development of Mercedes passenger cars [BW09], cf. Section 5.6).

  We present a formal definition of variability encoding based on a subset of Java and give a formal proof that variability encoding is behavior preserving. Based on this formal work, we implemented Hercules, a tool that provides variability encoding for C systems that express and control variability with the C preprocessor. We evaluated Hercules by testing it on real-world systems, such as Linux and SQLite.[3] The

[2] A tool for variability-aware parsing and type checking of compile-time–configurable systems [KGR$^+$11].

[3] https://www.sqlite.org/

result is that HERCULES can correctly encode variability except for a few special situations, which we discuss separately. We also implemented variability encoding for module-based configurable JAVA systems in the tool FEATUREHOUSE [AKL13].

- We developed and evaluated an approach for **variability-aware model checking** and implemented it on top of two software-model-checking tools.[4] We published our model-checking implementations in the tool suite SPLVERIFIER [AvRW$^+$13]. Our variability-aware model-checking approach uses variability encoding to generate variant simulators for the analyzed systems; it uses variability-aware optimizations to improve the model-checking process on the simulators. We evaluated this optimized, variability-aware model-checking approach and compared it to variant-based and sample-based approaches. Our evaluation is partly based on configurable-systems that we implemented based on community specifications [Hal05; KMSL83; PR01]. These systems have been used by other researchers in many related publications since then [Bey15; BLB$^+$15; Mei14]. Our evaluation shows that variability-aware model checking with our extensions is faster than variant-based and sample-based approaches. However, verification of variant simulators also consumes more main memory than variant-based verification, because simulators comprise more functionality than system variants.
  To mitigate this problem, we developed and evaluated a set of strategies that are partly variant-based and partly family-based, guided by our PLA model. We partitioned the set of all configurations of a subject system to generate multiple variant simulators that simulate mutually exclusive variant sets. Then, we compared verification of these partition-based simulators against other verification strategies. Our results show that a combination of strategies (family-based and variant-based in this case) can improve analysis performance. In particular, the combined strategies were faster than the variant-based strategy and consumed less main memory than the family-based strategy.
- Guided by the PLA model, we developed an approach that combines the feature-based and family-based strategies to improve **data-flow analysis between ANDROID apps**. Applications on an ANDROID device can often access private user data and apps can also communicate with other apps, which induces a potential for data-flow leaks. Leaks might involve several apps that pass on private data before it is leaked to,

[4] The tools are JPF-BDD [vRAR11], an extension to the model checker JAVA PATHFINDER [VHB$^+$03], and a similar unnamed implementation in the model checker CPACHECKER [BK11].

for example, untrusted internet services. For instance, an app that has sufficient rights to obtain a password could pass it to another app, which sends it somewhere on the internet. As ANDROID users can choose which apps they want to install on their devices, this scenario is highly configurable (apps are like features), and variability-aware approaches promise good results. We implemented an approach that models this app-analysis scenario as a data-flow problem and reports potentially malicious data leaks. Our approach first analyzes all considered apps in isolation (feature-based) and then combines the feature-based information to build a global communication graph (family-based). We evaluated our implementation on a set of 51 935 ANDROID apps. Our evaluation shows that our variability-aware analysis scales much better than similar analyses that do not consider variability.

1.3 Research Methodology

empirical results

In most studies that lead to this thesis, we conducted empirical research. We formulated research questions such as “Is family-based verification more efficient than variant-based verification? (Chapter 6)” Based on such questions, we phrased our hypotheses and implemented tools needed to answer the question (family-based verification and variant-based verification in this case). Based on the initial experiment results, we developed theories on which general principles or properties result in good or bad performance of different tools. Then, we chose a representative set of case studies and evaluated the theories based on the tools and the case studies. Such theories can then guide the improvement of analysis tools. This research method is an adaption of the *Goal-Question-Metric* approach [BCR94]. We define an evaluation *goal* (“Improve the performance of model checking for configurable systems.”), a more precise *question* (“What is the influence of sharing between variants on model checking?”), and a *metric* that allows measument in an experiment (“maximum consumed main memory” or “execution time”).

theoretical results

Besides empirical research methods, we also developed models to formalize and study observations and theories gained in experiments. For example, we observed in initial experiments that family-based verification outperforms variant-based verification (Chapter 6). However, it was not obvious whether the results could be generalized, because the central transformation in our approach, variability encoding, might not be correct on programs that are more difficult than our case studies. To mitigate this problem, we formally defined variability encoding and proved that the process correctly encodes variant behavior (Chapter 5). Furthermore, we discussed how programming languages such as Java and C deviate from our formal definition and how these differences affect variability-encoding correctness.

1.4 Outline

In Chapter 2, we provide a background on common concepts on which we build the thesis. In particular, we give an overview of the concepts of configurable systems. Furthermore, we outline software analysis and basic strategies for configurable-system analysis. In Chapter 3, we describe the PLA model and discuss different combinations of basic analysis strategies. In Chapter 4, we present our work on *presence-condition simplification*, a problem that touches many areas of research on configurable systems. Chapter 5 describes our work on *variability encoding*, a technique we used to implement the *family-based* analysis strategy. Chapter 6 discusses our implementation of *family-based model checking* and our evaluation of this strategy in comparison to alternative strategies. We also describe an evaluation of analysis strategies that combine variant-based and family-based verification and how these strategies perform in comparison to each other. In Chapter 7, we discuss a combination of the *feature-based* and *family-based* analysis strategies in a practical setting; we implement a taint analysis for Android applications to detect private-data leaks in a large set of Android apps.

supplementary website

Several chapters of this thesis rely on experimental research. Therefore, the documentation of our experiments is important to enable replication studies and studies that build on our results. We provide a supplementary website `http://www.fosd.net/vonRhein/` to support such studies. The website contains experiment documentation, setup details, and results.

CHAPTER 2

Background

This chapter provides a conceptual background to the thesis, including selected topics of configurable systems and software analysis. This chapter is not intended to present an exhaustive overview and motivation of the respective research areas. Instead, we present a short, condensed introduction of necessary key concepts. For complete coverage of the research areas, we refer to key textbooks in the following sections.

Section 2.1 presents background information on configurable systems [ABKS13; CE00]. Section 2.2 presents different static software-analysis techniques: type checking [Pie02], testing [CDS07], software model checking [CGP99], and taint propagation (a data-flow analysis technique) [HCF05]. Section 2.3 describes software analyses for highly configurable systems and introduces different strategies used for improving analysis efficiency [TAK^{+}14].

2.1 Configurable Software Systems

In this section, we discuss configurable systems, related concepts, and tools. First, we introduce the basic ideas and goals driving configurable systems in Section 2.1.1. We use a small, configurable printing-device system as a running example. In Section 2.1.2 we discuss different binding times for configuration options. Then we discuss how configurable systems can be implemented in practice (Section 2.1.3). We conclude with an overview of examples of configurable systems that we developed for our experiments and that we reused throughout this thesis (Section 2.1.4).

2.1.1 Terms and Running Example

A configurable system is a system that can be tailored to fulfill different roles in different environments and application scenarios. A product line is a similar concept with a slightly different focus. The Carnegie Mellon Software-Engineering Institute (SEI) describes a software product line as follows[1]:

> A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

The concept of a *feature* is at the core of this definition. This concept is inherently hard to define as it captures, on the one hand, intentions of the stakeholders of a product line, and on the other, design- and implementation-level concepts [ABKS13]. In this thesis, features capture design- and implementation-level decisions that are part of the software construction phase. A feature is a characteristic or end-user-visible behavior of a software system.

product lines vs. configurable systems

Product lines focus on the fact that features are reused between variants and that a mapping from features to implementation assets is part of the software-engineering process. Ideally, features are planned during the requirements-engineering phase of the software life cycle and are later mapped to specific, configurable parts of the implementation [ABKS13; CN01; CE00]. A configurable system is a system that can be configured at some point in time. In configurable systems, we do not focus on how the software has been planned. In the end, software product lines with many variants shall be implemented as configurable software systems and the same analysis concepts can be applied.

When describing concepts or experiments in this thesis, we use the terms configurable systems (with configuration options) or product lines (with features) depending on which fits better in the concrete case. If a technique is used primarily in the context of highly-configurable systems (e.g., Linux), we use configurable systems. If a technique is used primarily in the context of product-line research, we use product lines. However, our research is in principle applicable to both worlds.

printing-device example

Next, we introduce the printing-device system as a running example of a configurable system. We use this toy example throughout the thesis to illustrate problems and solution approaches. In practice, a similar, larger configurable system has been implemented for Hewlett Packard printers by the Owen Firmware Cooperative [TCO00].

The printing-device system consists of features that implement different printing and scanning functionalities. Variants of the system might represent

[1] http://www.sei.cmu.edu/productlines/

drivers for a family of printers. Some of the printers provide functionalities such as scanning or duplex printing and some do not (e.g., for cost reduction). To avoid potential implementation bugs and to keep the binary size of the driver small, variant of the driver include only the functionality that is also supported by the respective printer hardware.

The configurable printing-device system has five features: *BasicPrinter*, *Duplex*, *Color*, *Scan*, and *Copy*. *BasicPrinter* provides core printing functionalities: single-side printing, a user interface, protocols to connect to the printer, etc. The feature *Duplex* enables automatic double-side printing. *Color* allows to print with colors. *Scan* allows users to use the scan hardware of some printers. *Copy* (which requires feature *Scan*) allows users to duplicate documents.

features vs. configuration options

Configurable systems and product lines are centered on the concept of configuration options and features. *Configuration options* allow users to control settings and functionalities of the system. The controlled aspects might lead to externally visible behavior or not (e.g., colored output versus a buffer size). *Features* represent functionalities of a system that are of interest to a stakeholder. They *can* be mapped to actual implementation artifacts. In product lines, features are often implemented using configuration options. We focus on features that are mapped to configuration options and we assume that all configuration options in our subject systems correspond to a feature. In systems where features can be mapped to source code, we also use the name of the feature to refer to this source code. A feature can be *mandatory* (it has to be enabled in all variants) or *optional* (its selection is not fixed). A feature can also *depend* on whether other features are enabled or disabled. In the printing-device example, *BasicPrinter* is mandatory, *Duplex*, *Color*, *Scan*, and *Copy* are optional, and *Copy* depends on *Scan*.

By choosing values for configuration options, a user builds a *configuration* of the *configurable system* according to its desired application. Given a *configuration* and the system implementation, the corresponding *variant* (also called *product*) of the system can be derived with tools such as FeatureHouse [AKL13]. The behavior and properties of the variant correspond to the chosen configuration.

feature dependencies

Often the user is limited in the ways she can select features due to dependencies among them. These dependencies are defined in a *variability model* (also called *feature model* [ABKS13; CE00; KCH+90]). For example, the printing-device system does not allow users to choose the feature *Copy* unless *Scan* is chosen, too. A configuration that conforms to the variability model is called a *valid configuration.* A configuration contains choices on all configuration options (each is either selected or deselected). Therefore, a configuration corresponds to exactly one variant of the configurable system. A configuration Φ

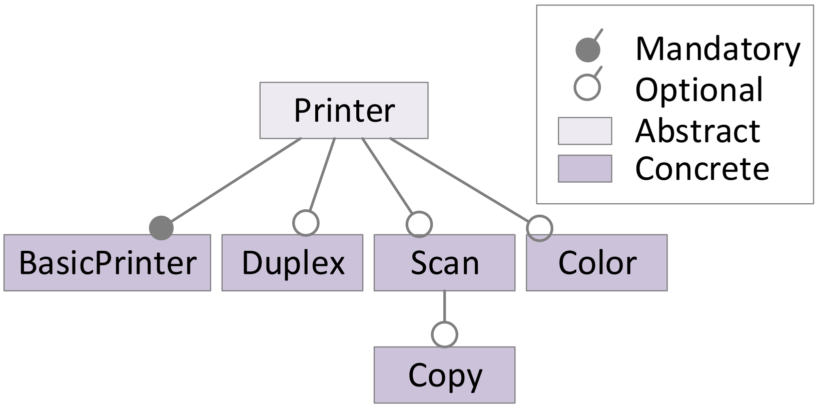


Figure 2.1: A tree-based variability model of the printing-device system

can be written as a propositional expression with one satisfying assignment (e.g., $\Phi = \mathit{BasicPrinter} \wedge \neg \mathit{Duplex} \wedge \mathit{Color} \wedge \mathit{Scan} \wedge \mathit{Copy}$). The set of all valid configurations defines the *configuration space* of the system.

The set of all variants that are derived from valid configurations is also called the system *family*. Every system variant in the family is supposed to satisfy the system's specification. That means that each variant could be built and deployed for a customer. A main motivation of this thesis is the fact that it is very difficult to ensure that this assumption holds on a given configurable system.

There is a number of different notations for variability models. The standard notations are *feature models* [ABKS13; CE00; KCH$^+$90] and propositional logic formulae. Feature models are useful if the model is intended to be read by a user. For example, if a user is supposed to configure a variant of a configurable program, she might want to know about constraints between the features. In this case, the variability model should be shown in one of several well-known tree-based notations such as a *feature model*. Figure 2.1 shows a feature model of the printing-device system. The circles on the feature boxes denote optional and mandatory features. The tree hierarchy denotes dependencies between features. This model contains an *abstract* feature, Printer, which is not mapped to code artifacts. It is used only for structuring the variability model. *Concrete* features are mapped to code artifacts.

Propositional logic formulae are better suited when the variability model needs to be processed by tools [Ber13; CW07; Men09]. A Boolean variable in a formula corresponds to one feature and states whether the feature is included in a variant. In our example, the variability model is $\hat{\Phi}$ with $\hat{\Phi} = \mathit{BasicPrinter} \wedge (\mathit{Copy} \rightarrow \mathit{Scan})$. The feature *BasicPrinter* is mandatory and the feature *Copy* depends on feature *Scan*. In case of non-Boolean features (with more than two possible settings) [PNX+11], the concept can be extended such that propositions over integer values are allowed (e.g., $print\_quality = 600dpi$).

Large configurable systems in practice, such as LINUX, often use dedicated languages to state feature dependencies. Due to the size and modular structure of LINUX, propositional logic and feature models would not scale. LINUX uses the KCONFIG language to define dependencies between configuration options. A KERNEL CONFIGURATOR guides the user through the configuration process based on the KCONFIG files.

2.1.2 Binding Times

Configuration choices can be bound at different points of time (*binding times*) during the application build process. In large configurable systems usually three binding times for configuration options are used: compile time, load time, and run time. For example, choosing on which operating system an application should run is a fundamental choice. In most cases, the operating system is known and therefore chosen at the start of the configuration process (compile time). Code that would be needed for alternative operating systems is not included in the running system, which improves efficiency. Another example is in which display resolution a graphical user interface should be displayed. The value of this option changes relatively often as users move to different display devices or connect their computers to projectors. Therefore, this option should be chosen as late as possible (e.g., at load time or at run time). An option that can be reconfigured after it has been chosen once has a *dynamic binding mode*, which improves flexibility [CE00]. For example display resolutions can be reconfigured at run time.

The different binding times are relevant in this thesis because our experiments show that analyses are more efficient if variability is handled as part of the analysis instead of being handled in a preprocessing step. Based on this insight, we develop an approach to transform compile-time options to load-time options (Chapter 5). This transformation enables efficient analysis techniques based on load-time variability (Chapter 6).

Because the printing-device system is too small to illustrate useful examples for different binding times, we use the SQLITE[2] database engine as illustrative example to describe all three binding times in the next paragraphs. SQLITE is a program that manages a database. During its run time SQLITE allows the user to manipulate a database with SQL queries and modification commands. sqlite

[2] http://sqlite.org/

```
  /* Shared library endings to try if zFile cannot be loaded as written */
  static const char *azEndings[] = {
#if SQLITE_OS_WIN
     "dll"
#elif defined(__APPLE__)
     "dylib"
#else
     "so"
#endif
  };
```

Figure 2.2: An example of compile-time options in the SQLITE3 amalgation version 3.8.1

compile-time configuration

Compile-time options are configuration options that are chosen by the user before the application is compiled (e.g., the target operating system). Variability implemented with compile-time options is also called *static* variability. The values of such options cannot be changed afterwards. In SQLite, compile-time variability is implemented with ifdef preprocessor directives. For example, Figure 2.2 shows an excerpt of sqlite3.c that determines the target operating system. Depending on the operating system, different file-name suffixes are expected for shared libraries ("dll" for Windows, "dylib" for Apple systems, "so" for UNIX). Variable code is enclosed in #if <condition> and #endif tokens where <condition> is a logic constraint on configuration options. #elseif <condition> (or #elif <condition>) and #else denote alternative code blocks.[3] In ifdef-based systems, users can define values for each configuration option. We say that a option is *enabled* if it has been defined to a non-zero value. If an option has been defined as zero or not defined at all, it is *disabled*.

The preprocessor removes all code that is enclosed by annotations for which the conditions are not satisfied. Therefore, the choice of the operating system cannot be reverted after ifdefs are processed. There are many implementation techniques that resolve variability before compilation, such as mixins [Bat04], superimposition [AL08], aspect-oriented programming [KLM^{+}97], or conditional compilation (e.g., ifdefs). The advantage of most compile-time variability techniques is that they avoid overhead at run time, since variability is removed before compilation. Also, for example ifdefs, allow to modify almost any token in the source code conveniently (including data types).

load-time configuration

If an option should be selectable after the application has been compiled and installed, it is not possible to resolve its variability before compilation. For

[3] #ifdef <option> is a shorthand for the command #if defined <option> which determines whether an option has been defined in a configuration. We use the term ifdef to refer to the concept of preprocessor directives.

some options, it is desirable to delay the choice until the application is started (load time); later, the values for such options can be fixed. These options are *load-time options*. Load-time options are usually implemented as command-line parameters or via configuration files. SQLite offers many parameters that can be specified when starting the program. An example is the -echo parameter, which causes the program to log each run-time command (e.g., SQL query) to the console before it is executed.

run-time configuration

Run-time configuration is necessary if the value of the option can be chosen only at run time or if it must be changeable at run time. Run-time options are often more expensive in terms of computation time or maintenance costs than compile-time or load-time options. Enabling or disabling an option at run time might require modifying values in different modules that depend on the enabled or disabled option. In SQLite, run-time options can be configured with either SQL commands or SQL-like PRAGMA commands. As an example, we use the PRAGMA locking_mode command from SQLite.[4] The option influences how SQLite handles database locking. If set to NORMAL, the database is unlocked after each write command and other database users (database connections) can write. If the option is set to EXCLUSIVE, the lock is held by one database connection until that connection is closed. The user can change the value of the locking_mode PRAGMA at run time. In this case, changing the value of the option is cheap because no data (e.g., the database stored on disk) has to be modified during the value change.

2.1.3 Implementation Mechanisms

There are two main mechanisms for implementation of configurable software systems: *module-based variability*, where feature implementations are separated in modules and *annotation-based variability*, where feature implementations are marked with variability annotations in source-code files [ABKS13]. Both mechanisms are used in the subject systems of our experiments.

module-based variability

Figure 2.3 shows an excerpt of the configurable printing-device driver implemented in Java with *module-based variability*. Each of the features is implemented in a separate code module and, in this example, each module contains parts of the Printer class. Variants of the configurable system are generated by combining the modules that correspond to selected features. There are different possible composition semantics for module-based variability, such as mixins [Bat04], superimposition [AL08], or aspect-oriented programming [KLM$^+$97]. In this example, we choose superimposition as implementation technique. Composition of each printer variant starts with the *BasicPrinter*

[4] http://www.sqlite.org/pragma.html#pragma_locking_mode

Feature *BasicPrinter*

```
 1 class Printer {
 2   void print(Page p) {
 3     ... // basic printing
 4   }
 5   void print(Page front, Page back) {
 6     printMulti(front, back);
 7   }
 8   void printMulti
 9     (Page front, Page back) {
10     ... // print both pages on one sheet
11   }
12 }
```

Feature *Duplex*

```
13 class Printer {
14   void print(Page front, Page back) {
15     printDuplex(front, back);
16   }
17   void printDuplex
18     (Page front, Page back) {
19     ... // duplex printing
20   }
21 }
```

Feature *Color*

```
22 class Printer {
23   void print(Page p) {
24     if (p.isColored()) {
25       ... // color printing
26     } else { original(p); }
27   }
28 }
```

Feature *Scan*

```
29 class Printer {
30   // scanning of one page
31   public Page scan() {
32     ...
33   }
34 }
```

Feature *Copy*

```
35 class Printer {
36   // scans a page and prints it
37   public void copy() {
38     print(scan());
39   }
40 }
```

Figure 2.3: Excerpt from a FEATUREHOUSE-based implementation of the printing-device system (module-based and feature-oriented). The class declarations of *Duplex*, *Color*, *Scan*, and *Copy* refine the corresponding declaration of *BasicPrinter*.

feature. Each feature that is added in the composition process refines existing implementation components (fields, methods) if they have the same name and type. For example, the method print(Page) of feature *BasicPrinter* is refined by print(Page) of feature *Color*. The superimposition approach we use in the example introduces a new keyword, original. By using this keyword refining implementations can call the refined implementations and thus *add* functionality to the refined features. In the printing-device system (Figure 2.3) the implementation of print(Page) in feature *Color* optionally calls (Line 26) the *original* implementation (which resolves to print(Page) in feature *BasicPrinter*).

superimposition

Figure 2.4 shows a variant composed based on a valid configuration. The variant contains the features *BasicPrinter*, *Duplex*, and *Scan*. The variant has been derived from the feature modules shown in Figure 2.3 using the superimposition mechanism. *Superimposition* starts with one feature (*BasicPrinter* in our example) and iteratively merges the source code with the code of other

```
class Printer {
  // basic printing method
  public void print(Page p) {
    ...
  }
  void print(Page front, Page back) {
    printDuplex(front, back);
  }
  public void printDuplex
    (Page front, Page back) {
    ... // duplex printing
  }
  // scanning of one page
  public Page scan() {
    ...
  }
  // scans one page and prints it
  public void copy() {
    print(scan());
  }
}
```

Figure 2.4: A variant of the printing-device system including features *BasicPrinter*, *Duplex*, *Scan*, and *Copy*

chosen features. In the example, it first combines *BasicPrinter* with *Duplex* and then combines the result with *Copy*. The resulting code is shown in Figure 2.4. Method print(Page,Page) from feature *BasicPrinter* is not included in the variant, because it has been refined during composition by the method from feature *Duplex* with the same signature. For more details, we refer to a description of the exact composition semantics for superimposition [AL08; ALMK10].

Another approach for static variability is to have all code in one code base and add annotations to selected parts of the code. The annotations contain propositional logic constraints. Each annotated code fragment is included in the variant only if the constraint is satisfied by the chosen configuration. Annotation-based variability can be implemented using tools such as the C preprocessor (ifdef annotations) or CIDE [KAK08]. In CIDE, features are mapped to colors and each element of a program's abstract syntax tree (AST) can be colored to bind it to a feature. An evaluation has shown that source-code background colors mapped to features improve program comprehension [FKA+13]. The C preprocessor is based purely on lexical annotations. Variability is resolved using the C preprocessor and the derived variant is usually given to a C compiler afterwards. However, the preprocessor is independent of the C language and works with any text-based artifacts that do not interfere with its keywords. Figure 2.5a shows our running example using ifdef directives. Features are represented by configuration options used in ifdef conditions (lines 3, 14, and 20). In contrast to the superimposition approach, ifdef directives allow to make any token of the host language optional (e.g., keywords like private or function parameters). Figure 2.5b shows a variant with features *BasicPrinter* and *Duplex* derived from the ifdef implementation.

annotation-based variability

```
class Printer {
  void print(Page p) {
    #ifdef Color
    if (p.isColored()) {
      ... // color printing
    }
    #else
    ... // basic printing
    #endif
  }
  void printMulti
    (Page front, Page back) {
    ... // print both pages on one sheet
  }
  #ifdef Duplex
  void printDuplex
    (Page front, Page back) {
    ... // duplex printing
  }
  #endif
  void print(Page front, Page back) {
  #ifdef Duplex
    printDuplex(front, back);
  #else
    printMulti(front, back);
  #endif
  }
}
```

(a) Excerpt from an ifdef implementation of the printing-device system

```
class Printer {
  void print(Page p) {





    ... // basic printing

  }
  void printMulti
    (Page front, Page back) {
    ... // print both pages on one sheet
  }

  void printDuplex
    (Page front, Page back) {
    ... // duplex printing
  }

  void print(Page front, Page back) {

    printDuplex(front, back);



  }
}
```

(b) A variant including *BasicPrinter* and *Duplex*

Figure 2.5: An ifdef-based implementation and a variant of the printing-device system. Blank lines in the variant illustrate the code that has been removed by preprocessing.

2.1.4 Examples of Configurable Systems

In this section, we introduce three configurable systems we used in case studies throughout the thesis. The described systems are well known and used in the research community [BLB$^+$15; CCH$^+$11; CHSL11; Hal05; Mei14; PR01; SvRA13]. We introduce the systems here to avoid repeating the introduction in different chapters. Based on system specifications and designs by others, we implemented C and JAVA versions of the systems. The designs and specifications of these three configurable systems have been used before to assess configurable-system verification (for example by Classen et al. [CCH$^+$11; CHSL11]). Our implementations have since been used as benchmarks by other researchers [Bey15; BLB$^+$15; Mei14], so they are a contribution to the research community by themselves. The systems are implemented with module-based variability (superimposition with FEATUREHOUSE) and they are available online from the

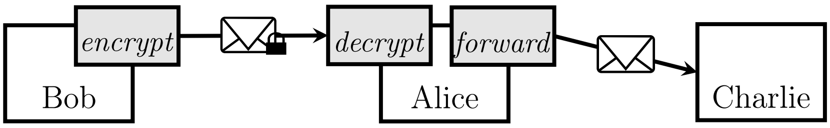


Figure 2.6: Interaction example in the E-MAIL system. An encrypted mail from Bob to Alice is forwarded (unencrypted) to Charlie.

SPL2GO database.[5] These systems are very small in comparison to real-world configurable systems such as the LINUX kernel. However, they are ideal for evaluating variability-aware analyses before investing the effort necessary to make the analyses applicable to real-world systems such as the LINUX kernel. The systems capture essential patterns of configurable systems which can be thoroughly studied with different analysis strategies.

E-Mail system The E-MAIL system models a configurable e-mail communication system comprising 40 variants. It is based on specifications of features and feature interactions by Hall [Hal05]. The system has 9 features, which provide functionality such as encryption, forwarding, and signatures. The system also has interactions between features; for example, feature *encrypt* encrypts all outgoing e-mails and specifies that the text from the e-mails must not be sent in plain text in future communications. However, feature *forward* is unaware of this specification and forwards the decrypted e-mail to another user as illustrated in Figure 2.6. This is a typical example for an interaction between features, which we detect with variability-aware verification (Chapter 6).

Elevator system The ELEVATOR system is a simplified model of a passenger elevator, based on designs by Plath and Ryan [PR01]. The model has 6 features and 20 variants. Features enable, for example, priority service for a special floor or stopping when the elevator is empty. Like the E-MAIL system, the elevator system has several interactions between features. For example, the priority-service feature enforces that the elevator goes directly to the special floor if it is called there. Even if there are passengers in the elevator car, it does not stop. This behavior violates several specifications for the normal elevator behavior (e.g., elevator doors must open when the car is at a floor with a requested stop). Of course this violation is intented from the view point of the priority-service feature. However, in practice, developers of different features are not necessarily aware of each other, so similar interactions can occur in practice, too.

[5] `http://spl2go.cs.ovgu.de/`

Mine-Pump system The MINE-PUMP system is based on work in the CONIC project [KMSL83]. The system simulates a water pump in a mining operation. The pump must keep the bottom of a mine shaft dry. Its operation is regulated by environmental conditions. For example, it must stop pumping when there is combustible methane gas in the mine. Features of the system are, for example, sensors for gas and water detection. The system has 5 features and 64 variants. The code base of the system is smaller than that of the E-MAIL and ELEVATOR systems, which makes it very well suited for preliminary experiments with variability-aware tools.

2.2 Software Analysis

Software analyses are techniques that take a given software implementation and extract high-level information from it. In this thesis, we address the question of how program analyses can be applied to configurable systems with a very large number of variants. We focus on analysis techniques that go beyond merely extracting statistics, such as the number of code lines or the Halstead code complexity metric [Hal77]. Instead, we focus on techniques that try to make statements about the program behavior: model checking, and taint propagation.

In this thesis, we adapted model checking and taint propagation for application on highly-configurable systems. In this chapter, we describe, in addition to model checking and taint propagation, the analysis techniques type checking and testing. Type checking has been extended to configurable systems by other researchers (e.g., [AH10; DCB09; KATS12; KOE12; KvRHA13; LvRK^{+}13]). We use type checking in illustrative examples (e.g., Table 2.2 on page 37) to show basic concepts of configurable-system analysis. Furthermore, we describe our extension of the type system of a simple programming language in Chapter 5. We use testing in Section 5.5.2, to show that our tool HERCULES correctly encodes program behavior on a specific test suite.

analysis goals

Model checking and testing assert that the run-time behavior of a program satisfies its specification. Type checking asserts that no type errors can occur at program run time (as far as the language allows to assert this statically). Taint propagation can be used to assert that private information that is queried by a program (e.g., passwords) are not leaked to the public (e.g., internet servers). Generally, all considered techniques focus on prevention of faulty program behavior. By reduction to the halting problem, one can prove that finding all possible run-time errors in an arbitrary program is undecidable [Dav58]. However, one can still give approximate answers or limit the set of possible input programs and, therefore, focus on checking specialized specifications.

Actually, the difference between the goals of type checking, testing and model checking lies in the specifications that are checked, which are discussed in the following sections (Section 2.2.1, Section 2.2.2, and Section 2.2.3).

analysis performance

Software analyses are typically characterized by their goals, their limitations, and their performance characteristics. In our experiments, we compared analyses with the same goals and limitations based on their different performance characteristics. Performance is typically measured in terms of *computation time* needed to finish the analysis, and the *maximum amount of main memory* needed during the analysis. From theoretical computer science, it is known that often computation time can be reduced if more memory is available and vice versa (*space–time tradeoff*; e.g., [Oec03]). The discussed analyses are usually implemented with a focus on optimizing the computation time, because main memory is easier to get than more time. We do not discuss the space-time tradeoff any further, because it is well known in computer science.

analysis accuracy

Another property of analyses is the accuracy of the analysis results. Given a set of analysis subjects (e.g., systems or system variants), and analysis results on these subjects, we divide the set of subjects, in four partitions (assuming that each subject has one or no defect):

TP	true positive	The set of subjects where a *real* defect was detected by the analysis.
TN	true negative	The set of subjects *without* defect where *no* defects were detected.
FP	false positive	The set of subjects *without* defect where (false) defects were reported by the analysis (a.k.a. type I error [AF96]).
FN	false negative	The set of subjects where a *real* defect was *missed* by the analysis (a.k.a. type II error [AF96]).

Based on these definitions, we define the properties *precision* and *recall* of an analysis as $precision = \frac{|TP|}{|TP|+|FP|}$ and $recall = \frac{|TP|}{|TP|+|FN|}$ [RP06]. Precision and recall of an analysis can be used to predict the quality of analysis results of future experiments with similar subjects. A high precision means that the analysis reports only few false positive results. A high recall means that few defects are missed.

section overview

We discuss different analysis techniques in the next sections. In particular, we discuss type checking (Section 2.2.1), testing (Section 2.2.2), software model checking (Section 2.2.3), and taint propagation (Section 2.2.4). In our work, we show how different analysis strategies can be applied to make these techniques applicable to configurable systems with a high number of variants.

2.2.1 Type Checking

Pierce [Pie02] states that

> A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

The definition emphasizes that phrases in the program code (e.g., variable names and function names) are classified. A type system assigns a type to each variable and function in the program and checks whether each usage of the variable or function is valid for its type.

Pierce further explains that type checking can guarantee that *well-typed* programs are free from *type errors* at run time. Depending on the language, type checking can protect from low-level faults such as missing function definitions, but also from errors that occur when a variable is used in an unintended way. For example, a program variable x can hold different values during the execution of a program. A type is a characterization of the set of values a variable can hold. Consider a variable x of type int that is supposed to hold only integer values. A *type error* occurs if there is a statement in the program that assigns a non-integer value to x or uses x in any way that requires a non-integer value.

type checking in this thesis

We use type checking in two chapters of this thesis: First, we use it to illustrate analysis strategies for configurable systems in Section 2.3. Second, we extended the statically-typed language FJ [Pie02] with new language constructs to implement load-time variability (Section 5). The resulting new language is still statically type safe.

2.2.2 Testing

> Software testing is a process, or a series of processes, designed to make sure computer code does what it was designed to do and that it does not do anything unintended. Software should be predictable and consistent, offering no surprises to users. [MS04]

In software testing, a given *test suite* (a set of *test cases*) is used to execute a system under test and to check its behavior. Each test case consists of the input to the system (e.g., parameters or user input) and the expected behavior of the system. During test execution, the system is fed with the input and the visible behavior of the system is compared to the expected behavior. If the visible behavior during the test matches the expected behavior in each test case, the system passes the test case. The specification checked with testing is that *one* run of the program shows the expected program behavior. One problem with testing is that, if the visible behavior of the program is influenced by non-determinism, a test case might pass or fail without obvious reasons. This makes it very difficult to work with the test results.

testing configurable systems

A good test suite executes behavior that is representative for the intended usages of the system. The difficulty of building such a test suite for a system without configuration options is multiplied when considering configurable systems. A test suite for a configurable system should cover all relevant system behavior in all relevant system configurations [CDS07]. The crux is to determine whether a certain behavior can be guaranteed to stay the same in different variants. For such behavior, one test case can ensure the behavior in multiple variants and, therefore, reduce the size of the test suite and the time for test-suite execution. For example, an approach to efficient test-suite generation for configurable systems [BLB+15] reuses the concepts we developed for family-based model checking (Chapter 6). In this thesis, we used an existing test suite for SQLite to evaluate our tool Hercules for variability-encoding (Section 5.5.2).

2.2.3 Software Model Checking

> *Model checking* is an automatic technique for verifying finite state concurrent systems. [CGP99]

state-space exploration

A model-checking algorithm explores a system's *state space*. In its basic form, a state space is a structure that consist of all states (e.g., a set of variable assignments) that a system can reach, and all transitions that the system can make between those states. During state-space exploration, the algorithm checks each path through the state space against a given *behavior specification*. When all paths have been explored and none of them violates the given specification, then the system is proven to be *safe* concerning the specification. Model checking verifies that *all* possible runs of the program show the expected program behavior (or it gives an example run that shows different behavior).

state-space generation

In *software model checking* the state space is generated based on the implementation of a software system. The state of a running program is defined by different data structures: the *call stack* stores the hierarchy of function calls, the *heap* stores field values, and the *program counter* stores a reference to the currently executed statement. Ideally, each state in the state space would faithfully represent these data structures. In practice, it is often impossible to generate all states at this level of detail. Infinite loops in a program can lead to non-terminating program paths and to an *infinite state space*. One popular technique to handle such infinite state spaces is *bounded model checking* which, for example, limits the length of explored program paths and ignores defects that occur beyond this limit. Even without infinite loops, growing complexity of a system leads to an exponential growth of the state space. For example, a Boolean variable in the system accounts for a duplication of the number of states, in the worst case. Even in small programs, this can lead to a combinatorial *state-space explosion* [CKNZ12]. To avoid this explosion, modern software model-checking tools use two main optimizations:

First, they use on-the-fly state-space generation [FMJJ92]. States are generated only when they are reached during the exploration and unreachable states are not considered at all. Second, they use abstraction to generate and explore an *abstract state space*. Generated *abstract states* usually do not contain exact or complete information about the elements of the stack or the heap. Instead, a set of *concrete states* can be represented by one abstract state such that the size of the abstract state space is reduced. This approach can be further optimized by *lazy abstraction*, where the level of abstraction is adjusted for different parts of the state space, driven by the model checker [HJMS02].

limitations

Software model checking suffers from some limitations that restrict its applicability to arbitrary programs. However, if it can be applied, the reward is a proof of the system's correctness, which might be worth the investment. The main limitation is the state-space explosion discussed above. Additionally, software model checking requires that the order in which paths are explored has no influence on the concrete states in the paths. Consider a side effect in an execution path A that is not covered by the model checker; for example the closing of a network connection. Once the end of one execution path A is reached, another, previously unexplored, alternative path B should be explored. If path execution of path B depends on the network connection, the reachable states in this path are different than they would have been if B had been explored first. This means that each exploration step must be deterministic, which is difficult to achieve if the system is based on an environment that has a modifiable state (e.g., network communication or file access). To allow the model checker to capture the environment state too, the environment has to be modelled as part of the system [TDP03]. In an extreme case, some statements modify the behavior of statements in the program itself (e.g., using JAVA reflection) and the program source code must be part of the states. Due to these difficulties software model checking is usually limited to systems that do not use network communication, file writing, or reflection.

software model checking in this thesis

We used model checking in Chapter 6: We extended two software model checking tools (CPACHECKER [BK11] and JAVA PATHFINDER [VHB$^+$03]) such that they perform well on configurable systems. We describe these extensions in detail and we compared the performance of several strategies for verification of configurable systems (Chapter 6).

2.2.4 Taint Propagation

Taint propagation is a static, data-flow analysis technique that determines how a certain input value to a program is used during its execution.

taint mode in PERL

The origin of the technique is the *taint mode* in the PERL programming language [WCO04]. The taint mode protects the computer from malicious program input. When in taint mode, perl marks data originating from outside the program (e.g., user input and environment variables). The language prevents such data from being used as parameter to sensitive functions, such as running local commands on the computer. Similar techniques have also been implemented for other programming languages, such as JAVA [HCF05].

privacy protection

Besides protecting the computer from malicious input, taint propagation can also be used to protect private user data from being misused. The idea is to mark (taint) private data, such as a password, once it enters the analyzed system. Then, all variable values that originate from this data through combination or copying are also tainted. Once a tainted value is used in a potentially unsafe function, such as an unsecure internet connection, a privacy breach is found.

taint propagation in this thesis

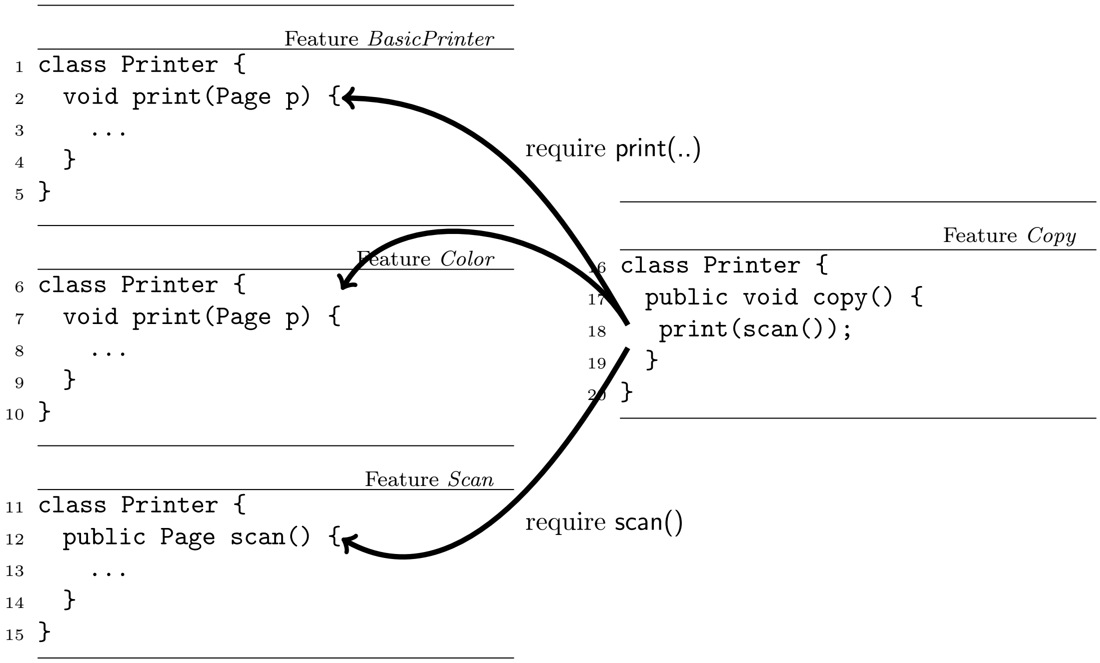
We used taint propagation in a study where we tried to find privacy data breaches in the communication between apps in the mobile-phone operating system ANDROID (Chapter 7). We designed an approach that makes taint propagation scalable using techniques developed for other configurable-system analyses (e.g., presence conditions).

2.3 Configurable-Systems Analysis

analysis of configurable systems

Configurable systems are used in security relevant domains such as avionics, automotive products and medical systems. We give a short recapitulation of the NH90 helicopter example described in the introduction. The NH90 project has two goals that are of particular interest here. The first goal is that each customer receives only code that is used in the respective system configuration. This requires, at least, a dead-code analysis for each relevant system variant. The second requirement is that all variants must fulfill certain specifications concerning memory usage or real-time deadlines. These project goals show that software analyses play an important role in configurable systems in practice.

The huge number of variants of configurable systems promotes a system design where features have as little impact on each other as possible. If a system developer can guarantee that features do not interfere with each other, it would suffice to test some configurations, such that each feature is covered at least once. Unfortunately, *feature interactions* can be very subtle and they are generally hard to rule out. Many feature interactions occur between two



```
class Printer {
  void print(Page p) {
    ...
  }
}
```

```
class Printer {
  void print(Page p) {
    ...
  }
}
```

```
class Printer {
  public Page scan() {
    ...
  }
}
```

```
class Printer {
  public void copy() {
   print(scan());
  }
}
```

Figure 2.7: Type dependencies in the printing-device example

features (*pair-wise interaction* or *first-order interaction*), but also interactions between more than two features (*higher-order interactions*) are possible and might be even harder to detect.

The research community has developed many analyses for configurable-systems which take variability in configurable systems into account (*variability-aware*). Such analyses need to reason about variability in the systems and efficiently represent variability in data structures. We use example type dependencies from the printing-device system as illustration in the next sections. Figure 2.7 shows the type dependencies of the feature *Copy*. The feature requires that some other feature implements a function print(..) with one parameter and that a feature implements function scan(). Furthermore, the parameter of print must have the same type as the return type of scan. If one of these requirements is not met in a variant, the variant has a type error. In the printing device system, print(Page) is implemented in *BasicPrinter* and in *Color*, and scan() is implemented in feature *Scan*. Therefore, the feature *Copy* depends on the presence of features *BasicPrinter* and *Scan* (which is documented in the variability model). To use the example as illustration for different analysis strategies, we assume, in Sections 2.3.2–2.3.5, that these dependencies have been omitted in the variability model.

In Section 2.3.1, we give a short overview over two common techniques used for reasoning about variability-induced constraints in analyses. Confronted with code reuse between variants and interactions between features, several strategies for efficient analysis of configurable systems have been developed. Then, we describe the four basic analysis strategies based a survey paper [TAK$^+$14]. We illustrate the strategies with our running example (printing system) in Sections 2.3.2–2.3.5. Finally, we discuss combinations of these strategies, also based on the survey, in Section 2.3.6 [TAK$^+$14].

2.3.1 Computational Problems in Variability-aware Analysis

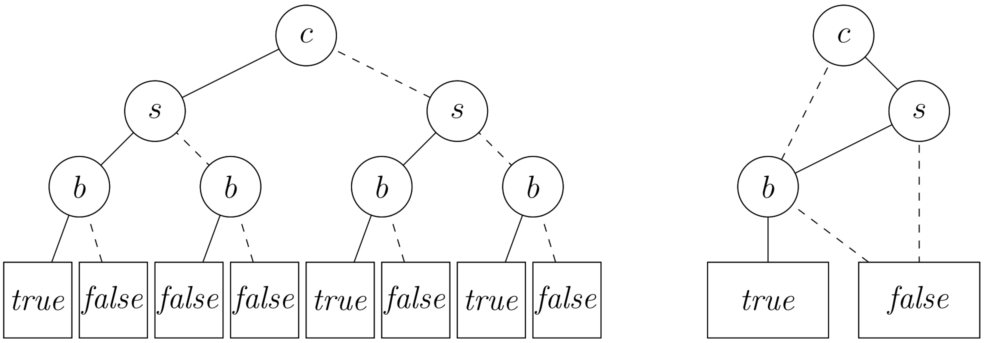
Variability-aware analyses have been developed to deal with very large configuration spaces. These analysis approaches are aware of variability in the system and exploit commonalities among variants. Exploiting commonalities allows such approaches to be more efficient on configurable systems than approaches without variability-awareness.

To deal with variability, analysis tools must have efficient means for representation and processing of constraints on configuration options. For example, an analysis might determine that a failure occurs in a system if a certain combination of options is set. Then it must validate that this combination can be selected by a user, according to the system's variability model. If users cannot select the combination, the error does not need to be reported.

Typically, constraints on configuration options are expressed as propositional logic expressions. In the printing device system, we might want to represent the condition for the absence of the Printer class to make sure we do not build variants without the class. The class is implemented in all features, so the expression is $\phi = \neg \mathit{BasicPrinter} \land \neg \mathit{Copy} \land \neg \mathit{Scan} \land \neg \mathit{Color} \land \neg \mathit{Duplex}$.

Now, we would want to check if there is a valid variant in which the class is absent. The variability model of the printing device system is denoted by $\hat{\Phi}_{ps}$ with $\hat{\Phi}_{ps} = \mathit{BasicPrinter} \land (\mathit{Copy} \rightarrow \mathit{Scan})$. We use the function sat to check if there are variants without a Printer class: $\mathrm{sat}(\phi \land \hat{\Phi}_{ps})$ returns true iff there is a variable assignment $\mathcal{A}$ under which ϕ and $\hat{\Phi}_{ps}$ hold (i.e., $\exists \mathcal{A} : \mathcal{A} \models \phi \land \hat{\Phi}_{ps}$). In our example, no such variable assignment exists. To implement this sat check in analyses, developers typically use SAT solvers or BDD libraries.

SAT was the first known NP-complete problem, as proven by Stephen Cook in 1971 and Leonid Levin in 1973. Since then many efficient heuristic tools have been developed that enable us to solve practical SAT problems fast. We do not describe SAT solving in more detail, but we refer to a reference handbook [BHvMW09].



(a) Unreduced BDD (b) Reduced BDD

Figure 2.8: Two BDDs that represent the function $(\neg c \lor (c \land s)) \land b$. The solid (dashed) lines represent the *true* (*false*) values of the variables. All paths leading to the terminal *true* are satisfying assignments for the function.

Another approach to the SAT problem is offered by Binary Decision Diagrams (BDDs) [Bry92], which are less common and warrant a more detailed discussion. A BDD is a data structure that represents a Boolean function. The function is stored in a compressed format that also enables efficient logic operations on the function. In comparison to SAT solvers, BDDs typically require more time when doing operations on an expression (and, or, etc.) but they are faster in executing SAT checks.

As an example for a BDD, consider the Boolean function $f(c, s, b) = (\neg c \lor (c \land s)) \land b$, in which c, s, and b are Boolean variables representing the features *Copy*, *Scan*, and *BasicPrinter*. The function represents all valid configurations of the printing-device system (the features *Duplex* and *Color* are optional). The truth table of this function is eight lines long. Alternatively, as shown in Figure 2.8a, one could represent the truth value of f by means of a diagram with root node c and two outgoing edges (one representing the value *true* for c and the other the value *false*), each of which lead to a nodes for s, and similarly with the last variable b. The eight leaves of this diagram represent the truth values of f. This diagram can be simplified (i.e., *reduced*) by merging redundant nodes and by removing redundant tests [BRB91]. Figure 2.8b shows the Reduced Ordered Binary Decision Diagram (ROBDD) representing f. Notice how the ROBDD for f only has five nodes (instead of fifteen for the non-reduced diagram). This reduction in size is one of the key factors for the efficient manipulation of Boolean functions using ROBDDs.

ordered BDDs

If the order of variables is kept fixed (hence the "ordered" in the name), it is possible to combine ROBDDs of different functions by means of Boolean operators in polynomial time. We refer to Bryant's seminal work [Bry92] for more details on the reduction algorithm and for other operations on ROBDDs. In the remainder of the thesis, we use the term BDD instead of ROBDD, as all the diagrams are considered to be reduced and ordered.

Unfortunately, even reduced and ordered BDDs may still have an exponential size in the number of Boolean variables used. The size can be reduced by optimizing the order of variables in the BDD, but finding the best variable ordering is an NP-complete problem. Despite the theoretical problems, it has been shown [CGP99] that BDDs offer an efficient mechanism for the exploration of large state spaces in practice.

In our example, the BDD in Figure 2.8b could be used for different interesting tasks. In configurable systems, we might be interested in whether there is a configuration possible with this constraint (SAT check). There exists such a configuration if there is a path in the BDD leading from the root to the *true* terminal. Another question would be how many configurations are possible with the constraint. In the unreduced BDD (Figure 2.8a), each configuration corresponds one of the three *true* terminals in the graph.[6] In the reduced BDD (Figure 2.8b), we also have to count the number of paths leading to the *true* terminal, but we have to weight each path with the number of variables which are not used on the path. The left path ($c - b - true$) does not contain the node s, so it counts for two configurations, resulting in a total number of three configurations. Both operations (checking satisfiability and computing the number of satisfying assignments) can be computed very efficiently in BDD libraries [Bry92].

2.3.2 Variant-based Strategy

A simple approach to run analyses on variable software systems is to generate and analyze all valid variants. Normally, the generated variants are expressed in a general purpose programming language, so one can use off-the-shelf analysis tools. We call this approach *variant-based*, because it analyzes all variants isolated from each other. The approach is *exhaustive* in the sense that it guarantees that all variants of the system are analyzed. The strategy is also known as the *product-based* strategy [TAK$^+$14] or as the *product-by-product* strategy [BLB$^+$15].

scalability

The approach has two drawbacks: First, the analysis has to consider all valid variants of the system. For large systems, it is even impossible to enumerate all valid configurations, so this strategy is confined to systems having little variablity. Second, the variant-based strategy generates many similar variants

[6] For this example, we do not consider the *Duplex* feature, which would double the number of configurations.

and it has to analyze similar code or behavior repeatedly. In a type analysis of our example, the basic printing method of feature *BasicPrinter* would be type checked in all variants, although it is always identical. We would find the type error described in Figure 2.7 in all variants which do contain feature *Copy* but not feature *Scan*. With increasing number of options, the number of variants rises fast. In the E-MAIL system, a benchmark used throughout the thesis, 9 options give rise to 40 variants. For the LINUX kernel with over 13 000 options in release (v3.9, April 2013) [PPB+15], it is even impossible to determine the number of variants, not to speak of generating them.

2.3.3 Sample-based Strategy

As the variant-based strategy is often not feasible, many real-world configurable systems in practice are analyzed only on a selected subset of all variants (called the *sample set*) [NL11; OMR10]. We name this general stategy *sample-based*. In the type checking example of the printing-device system (Figure 2.7 on page 28), we would find type errors if (and only if) the sample set contains a variant with *Copy* and without *Scan*. The key question in the sample-based strategy is what variants to include in the sample set. There are many possible approaches of which we discuss some next.

interaction-based sampling

We assume that generated sample sets are used to search for feature interactions. Therefore, our sampling strategies aim to generate sample sets that cover as many feature interactions as possible. For example, if we focus on finding all first-order interactions we would want to cover all pairs of features at least once in the sample set. This is the *pair-wise sampling* strategy [LOGS12; PSK+10]. Assuming F is the set of features, then a simple requirement for a basic pair-wise sample set would be that for each valid pair of features f_1 and f_2 in F there is at least one configuration in the sample in which the features are enabled. If $\mathit{isValid} : F \times F \rightarrow \mathbb{B}$ is the function checking if two features can be enabled together (feature model). This requirement can be formalized as $\forall f_1, f_2 \in F : \mathit{isValid}(f_1, f_2) \rightarrow (\exists v \in \mathit{sample\_set} : (f_1 \in v \wedge f_2 \in v))$. The type error in the printing-device system (Figure 2.7) is caused by an interaction of the features *Scan* and *Copy*, so it can be found with pair-wise interaction sampling.

The above requirement for pair-wise sample sets ensures that all valid pairs of features are covered, but it is far from perfect. First, it is very imprecise. For a system without constraints between features, the configuration in which all features are enabled would fulfill the requirement. Such a degenerated sample set is often undesired because it ignores possible *negative interactions*. A *negative interaction* occurs if a feature has to be deactivated to trigger the interaction. This can be fixed by ensuring coverage of all four combinations

((f_1, f_2), $(\neg f_1, f_2)$, $(f_1, \neg f_2)$, and $(\neg f_1, \neg f_2)$) in which features can interact. Second, the requirement accepts a very broad range of solutions. One possible fix would be to require that the sample set contains one of the *smallest* configurations for each pair of features. The size of a configuration could be defined as the number of enabled features. Even with this restriction, the sampling strategy can still be ambiguous and the problem of generating minimal test sets has been shown to be NP-complete [LT98]. Our experience is that sampling requirements should not be overrestricted because deriving sampling sets for large configurable systems gets even harder then.

Pair-wise sampling might also find *higher-order interactions* by chance if the interaction occurs in a variant in the sample set. However, the approach does not guarantee that all interactions between, for instance, three features are found. To guarantee triple-wise interaction coverage, the approach must be extended such that all triples (f_1, f_2, f_3) of features and (optionally) also their negative combinations are covered. In our experience, this approach quickly leads to large sample sets that are, again, infeasible to generate and to analyze. Such large sample sets diminish the advantage of sampling (fewer variants to analyze) over the variant-based strategy.

The main disadvantage of the sample-based strategy, however, is that the analysis can never guarantee that the entire configurable program is free from defects or errors (the analysis is not exhaustive). A second disadvantage is that similarities between variants are ignored (just as in the variant-based strategy).

2.3.4 Feature-based Strategy

A common problem with the variant-based and sample-based strategies is that similarities between variants are not used for optimization. The *feature-based* strategy avoids this problem by analyzing the features' implementation directly.

The main idea behind a feature-based analysis is to divide the entire analysis task such that each sub task targets a single feature. Then, the results of the sub tasks are aggregated and global knowledge about the entire configurable system is inferred. On this abstract level, the concept is similar to other divide-and-conquer algorithms such as merge sort.

In a practical implementation of this strategy, the main advantage would be that the per-feature analysis can abstract from the concrete feature implementations. Per-feature analysis would extract exactly the information that is of value during the aggregation. As an example, we describe a feature-based analysis [KvRHA13] of our printing-device system (Figure 2.7). The analysis does a type-check of each feature separately, such that it finds type errors that are caused by the feature itself (e.g., the feature declares an integer variable and uses it as a String). Furthermore, it generates for each feature a set of

Table 2.1: Results of feature-based type checking on the printing-device system. $\mathcal{T}_1$ and $\mathcal{T}_2$ are generic types. *Copy* imposes only the constraint that the parameter of print must be compatible to the return type of *scan*.

Feature	Requires	Provides
BasicPrinter	Type Page	void print(Page)
Color	Type Page	void print(Page)
Scan	Type Page	Page scan()
Copy	$\mathcal{T}_1$ print($\mathcal{T}_2$), $\mathcal{T}_2$ scan()	void copy()

program elements (functions, fields and classes) that are defined by the feature and a set of program elements that are required by the feature.

For our example, these sets are shown in Table 2.1. The features *BasicPrinter*, *Color*, and *Scan* each provide a function and require that some other feature defines a type Page which is used as parameter/return type. Feature *Copy* provides function *Copy* and requires that other features define functions print and scan. In a next step (not feature-based), one would use the requires and provides information to generate dependencies between the features. In our example, we would infer the type-correctness constraint ϕ with $\phi = (\mathit{Copy} \rightarrow (\mathit{BasicPrinter} \vee \mathit{Color})) \wedge (\mathit{Copy} \rightarrow \mathit{Scan})$. Next, we would check whether there are valid configurations which would have a type error. Such configurations have to be valid according to the variability model Φ_{ps} and violate the inferred constraint ϕ: $\mathrm{sat}(\Phi_{ps} \wedge \neg\phi)$. If we use the variability model from Section 2.1.1 then there are no such configurations.

The main problem of an analysis with a feature-based part is that the implementation of this part is very specific to the analysis and to the analyzed system. The feature-based part of the analysis must extract exactly the information required by the later aggregation part. Furthermore, the feature-based part should not require information from other features. After these considerations, a feature-based analysis can be seen as a form of preprocessing that encodes and filters the information of each feature for later analysis. A second problem is that the per-feature information generated by the feature-based analysis still has to be combined to gain global information. To implement this combination step, one has to choose a strategy to combine per-feature information. On this abstract level, one has to choose between the variant-based, sample-based, or family-based strategies again. One advantage of the feature-based strategy is that it can be applied in *open-world* scenarios where only part of the implementation is available. If only a subset of features is available, they can be analyzed, making assumptions about other features that are not available. In business scenarios where parts of source code might be protected by non-disclosure agreements, this is a huge advantage over the other strategies, which are limited to *closed-world* scenarios where the entire code base is known.

2.3.5 Family-based Strategy

The main problem of the variant-based and sampling strategies is that commonalities between variants are not exploited in the analysis. The *family-based* approach to this problem is to move the resolution of variability in the analysis process. Usually, features are chosen and variability is bound before the actual analysis tool (e.g., a verification tool or a type checker) is started. In a family-based approach, the variability remains such that the tool is able to process and reason about multiple variants of the system at the same time.

Family-based vs. variability-aware In literature, the concepts of family-based analyses and variability-aware analyses have not been clearly separated [ABKS13; TAK$^+$14]. Often the terms are used to as synonyms. We interpret the terms literally as two orthogonal concepts:

- A *family-based* analysis reasons about the entire family of systems (all variants) in one run. The main analysis tool (e.g., a model-checking tool) does not need to be aware of the variability that it processes. For example, the basic analysis approach which we extend in Chapter 6, is an off-the-shelf model checking tool that is *not* aware of variability in configurable systems. In Section 6.4, we use it to verify variant simulators that each represent *all* variants of a system (family-based).
- A *variability-aware* analysis is aware of the concepts of features and dependencies between them. It uses sharing of code between variants to optimize analysis performance. Often the concept is used to optimize family-based analyses, but it could also be used in other strategies. For example, a variant-based analysis is variability-aware if it reuses analysis results between analyzed variants. This example analysis could store and reuse control-flow graphs for functions that occur in multiple variants.

Implementation approaches Typically, there are two approaches to implement a family-based analysis: First, one can transform the static variability to another form of variability which exisiting analyses can handle. For example, one can transform ifdef options to load-time program parameters which can be handled by standard analyses. Second, one can extend a program analysis such that it is able to cope with the additional static variability constructs (this

Variant simulator {*BasicPrinter*, *Duplex*}

```
class Printer {
  static boolean _FeatureDuplex_enabled;
  // basic printing method
  public void print(Page p) {
    ...
  }
  public void printDuplex(Page front, Page back) {
    if (_FeatureDuplex_enabled) {
      //code from Duplex feature
    } else {
      print(front);
      ... // ask user to turn and re−insert page
      print(back);
    }
  }
}
```

Figure 2.9: A variant simulator of the printing-device product line covering the features *BasicPrinter* and *Duplex*

analysis is family-based and variability-aware). Both approaches have been explored by researchers, often for verification or testing of highly configurable systems [AvRW^{+}13; CCS^{+}13; CHL^{+}14; KKB12; LTP09]. Both approaches have in common that the analysis is applied to the whole system family at the same time. We illustrate both approaches in the following paragraphs.

Program transformation To illustrate the approach of transforming static variability, we show a family-based verification analysis. It first generates a *variant simulator* based on the configurable system. Static variability in the system is transformed to load-time variability (*variability encoding*). For example, Figure 2.9 shows a part of a simulator for the printing-device system. The simulator contains a variable for each encoded feature. In the example, we can use the *feature variable* _FeatureDuplex_enabled to control whether a variant with or without *Duplex* is simulated. As simulators use standard language constructs to express variability (if statements), they can be analyzed by standard verification tools. To verify all variants of the simulator we can force the verification tool to consider all possible values for the feature variables (this is explained in more detail in Chapters 5 and 6). To improve over earlier approaches in which variability encoding was done manually [PR01], we developed automatic tools for variability encoding (Chapter 5).

Table 2.2: Presence conditions during family-based type checking. A program element (e.g., void copy()) is defined in all configurations in which its presence condition (*Copy*) is satisfied.

Program Element	Presence Condition
class Printer	$BasicPrinter \vee Color \vee Scan \vee Copy$
void print(Page)	$BasicPrinter \vee Color$
Page scan()	*Scan*
void copy()	*Copy*

Variability-aware analysis extension We illustrate the approach of extending existing analyses, we show a basic family-based type-checking analysis using the type error in the printing-device system (Figure 2.7). A family-based type-checking analysis could be implemented in a two-phased approach: First, it would build a map of program element names to conditions that must be met such that the elements are defined (*presence conditions*). Second, it would analyze type dependencies in the code (e.g., function copy depends on function scan) and generate type-correctness constraints from the function's presence conditions. These constraints are shown in Table 2.2. In this case, the constraint would be identical to the constraint ϕ inferred in the feature-based analysis 2.3.4.

Comparison to other strategies The results of different comparative studies show that family-based analyses are often more efficient than variant-based and sample-based analyses [AvRW+13; CCS+13; CHL+14; KKB12; LTP09]. However, they require more implementation effort as either subject systems need to be transformed or the analysis has to be extended. We discuss detailed comparisons of the family-based strategy with other strategies in Chapter 6 and Chapter 7.

2.3.6 Combined Strategies

We discussed variant-based, sample-based, feature-based, and family-based analysis strategies. In practice, analysis strategies often combine these basic strategies to benefit from their advantages. In their survey paper [TAK+14] Thüm et al. discuss four simple combinations of these basic strategies: *feature-product-based analyses*, *feature-family-based analyses*, *family-product-based analyses*, and *feature-family-product-based analyses*. In each case, the name encodes which basic strategies are used and in which order they are applied.

For example, a feature-product-based analysis consists of two phases, similar to the type-checking analysis described in Section 2.3.4 on the printing-device system: First, the features are analyzed in isolation and, second, the properties that could not be checked feature-based are analyzed for each variant. This two-phase approach has three advantages. First, one can reuse information gained in the feature-based phase to speed up the variant-based phase. Second, as variant-based strategies often suffer from scalability problems, it is recommendable to do as much work as possible in the feature-based phase and thereby speed up the entire analysis. Third, this strategy can be applied in open-world scenarios where part of the features are not known. If such features are disclosed for the analysis at a later point, they can be easily added.

Thüm et al. conducted a literature survey to determine whether combined strategies are used in research. They found many approaches that use one of the first three combined strategies (feature-product, feature-family, or family-product) [TAK$^+$14]. However, currently, there is no approach that combines all three basic strategies (feature-family-product-based analyses). They note that future work should analyze and discuss the feasibility of this strategy in more detail.

Our impression is that, when designing a new analysis for configurable systems, one might be overwhelmed by possible development directions. Besides the configurable-system analysis strategies, one also has to consider implementation details of the analyses (e.g., type checking, theorem proving, or model checking). To make it easier to combine strategies and to predict advantages of certain combinations, we developed the product-line analysis (PLA) model and its visualization, the PLA cube. The PLA model allows free combination of the basic strategies for configurable-system analysis. It serves as illustration for possible combinations and as basis for research on feasibility of different combinations. We introduce the PLA model in Chapter 3.

CHAPTER 3

The Product-Line–Analysis Model

This chapter shares material with the publication "The PLA Model: On the Combination of Product-Line Analyses" in *VAMOS'2013* [vRAK+13].

In Section 2.3, we discuss three basic strategies for the analysis of configurable systems. Each strategy has advantages and disadvantages. The *variant-based* strategy is easy to implement, but does not scale for configurable systems with many independent features. The *sample-based* strategy improves over this scalability issue, but its analysis results are necessarily not exhaustive with respect to the configuration space. The *family-based* strategy is efficient and analyzes all variants, but requires variability-aware analysis tools.

There is evidence that combining strategies allows developers to exploit the advantages of multiple strategies in an analysis. For example, Siegmund et al. [SKK+12] developed an approach for predicting non-functional properties of variants using feature-based and sample-based strategies. Kästner et al. [KOE12] developed a type-checking approach that uses the feature-based and family-based strategies. In these approaches, the combination of strategies either improves performance compared to the variant-based strategy, or it makes the analysis feasible in the first place (when a variant-based analysis would not scale).

To better understand and compare the merits of combining strategies, we introduce the *Product-Line–Analysis Model* (PLA model) [vRAK+13]. The model is a formal framework for describing basic analysis strategies and combinations thereof. It guides analysis developers to compare existing strategies and design new strategies. Simple combinations of the strategies of Section 2.3

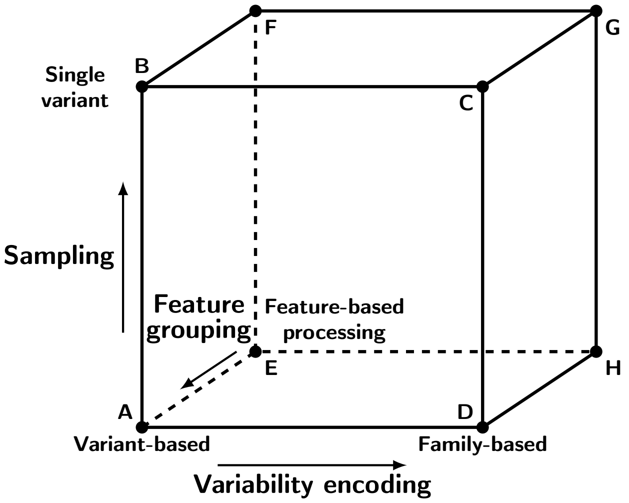


Figure 3.1: The PLA cube – visualization of the space of possible combinations of analysis strategies for configurable systems

have also been discussed by Thüm et al. [TAK^{+}14] Our model goes beyond and enables (1) more complex combinations and (2) strategy discussions based on a common, formally defined vocabulary.

The term "PLA model" stems from the focus on product lines in the original publication, but the model is applicable to configurable systems in general.

3.1 The PLA Cube

The *Product-Line–Analysis Cube* (PLA cube) is a visual representation of the PLA model. Each point in the cube represents a different analysis strategy. One key insight that underlies the design of the PLA model (and the PLA cube) is that the choice between basic strategies (e.g., variant- and feature-based) is not a black-and-white one. Instead, strategies can be combined in various ways. A strategy could for example combine features into groups and analyze these groups with a variant-based strategy (even though a group of features, such as *Copy* and *Duplex*, does not necessarily form a valid variant).

Such ranges of *strategy combinations* form three dimensions that give rise to the PLA cube, which is illustrated in Figure 3.1. The dimensions of the cube are (1) feature grouping, (2) variability encoding, and (3) sampling. Endpoints of the ranges include the pure variant-based, feature-based, family-based strategies and an extreme sample-based strategy, which analyzes only a single variant. In between, the strategies are combined to smaller or larger extents (cf. Figure 3.1).

We begin our discussion of the PLA cube with point **A** and then describe the dimensions feature-grouping, variability encoding, and sampling. Point **A** represents variant-based analyses, and we use it to describe how analyses along the three dimensions differ from these basic analyses. In point **A**, *all* variants are built and analyzed *separately* (as discussed in Section 2.3.2).

feature grouping

The *feature-grouping* dimension (e.g., range **E**–**A**) represents the granularity and size of feature combinations that are considered during analysis. For example, in point **A** whole variants are analyzed and analyses in point **E** consider only single features in isolation (pure *feature-based*). The range between **A** and **E** represents analyses that strategically group features to form compound units (not necessarily whole valid variants) and analyze these units as a whole. Examples include collecting metrics of code associated with individual features or certain intra-procedural static analyses that can operate on isolated code snippets [TAK+14]. Some software analyses depend on executable programs or search for feature interactions. Such analyses are difficult to implement on groups of features which might have dependencies on features that are not included in the group. To check for feature interactions, we could use a feature-based step to extract "feature interfaces". In the next step, we would check whether the interfaces fit each other (linker check, [AH10; LKF05]). Examples are analyses that search for critical feature interactions among combinations of certain subsets of features.

variability encoding

The *variability-encoding* dimension (e.g., range **A**–**D**) represents the extent to which variability is preserved and used during an analysis. Going from point **A** to point **D**, analyses encode some variability choices in the analyzed artifacts instead of analyzing different variants arising from these choices. Point **D** represents the *family-based* approach that encodes all features, information on variability, and valid configurations into one analysis subject that simulates all variants of the configurable system (e.g., for verification as discussed in Chapter 6). As the simulator contains information about all variants, only a single analysis pass (e.g., model checking) is necessary. To implement a family-based analysis, we need an analysis tool that can handle the variability information in the simulator. Some tools such as model checkers can handle the variability out-of-the-box by an exhaustive state-space exploration (e.g., [ASW+11]), but in other cases, a special analysis tool might be needed. In the range in between **A** and **D**, the analysis divides the configurable system in several subsystems and builds a smaller simulator for each subsystem. This way, still the whole configurable system is covered, but the simulators are smaller. Analyzing a smaller simulator will require fewer computing resources than analyzing an exhaustive simulator (simulating all variants). For example, we can use smaller simulators where an analysis of an exhaustive simulator would exceed main memory. We can also analyze multiple simulators in parallel to save time. In the printing-device system we could, for example, build a simulator with the *Duplex* feature and one without. Then we would analyze both simulators in parallel.

Analyses on the bottom plane of the cube (plane **A**–**D**–**E**–**H**) combine *feature grouping* and *variability encoding*. Such analyses group features to form compound units and analyze these units with a family-based analysis. Compared to a pure family-based analysis, this approach results in smaller simulators. Each simulator could yield, for example, an interface for a part of the system. The interfaces would still contain variability information which makes the interfaces more compact compared to variant-based interfaces. A family-based example result for the group of features *Scan* and *Copy* is that feature *Copy* requires that method print(Page) is implemented. For comparison, a variant-based result with the same information is that the combination $\{Scan, Copy\}$ requires print(Page) and the combination $\{\neg Scan, Copy\}$ also requires print(Page).

sampling The *sampling* dimension (e.g., range **A**–**B**) represents analyses that analyze only a subset of a system's configuration space (the analysis is not exhaustive). The subset of the configuration space, which is subject to analysis, determines where the analysis is located along the sampling dimension. For the printing-device system (Section 2.1.1), we might use domain knowledge and decide that only variants with feature *Duplex* have to be analyzed, as other variants are currently not requested by customers. So, only 6 of 12 variants must be analyzed, saving roughly half of the analysis time (compared to an exhaustive variant-based analysis; point **A**). However, sampling means that not all valid variants are considered, which may influence the conclusions one can draw from the analysis results (e.g., we may have missed a bug). Extreme cases of the sampling dimension are "all valid variants" (**A**) and "a single variant only" (**B**). When domain knowledge cannot be used for sampling, one can resort to *sampling heuristics* (e.g., pair-wise sampling) which select a representative subset of the variants.

Sampling can be combined with *variability encoding* or *feature grouping*–or with both dimensions, which has not been considered in previous work [TAK$^+$14]. For illustration, we start with an analysis that uses feature groups and analyzes them with a family-based approach (variability encoding). Now, we limit the analyzed feature groups such that they do not cover all variants of the system. For example, in the printing-device system, we could choose to include *Duplex* in all of the feature groups. This example shows a sampling approach because we would only cover half of the variants of the system. Alternative sampling approaches, such as pair-wise or triple-wise sampling, could be implemented by limiting the family-based analyses on the feature groups. For example, the analysis could skip situations which require more than two features to occur. The described analysis is represented by a point in the middle of the cube. If we choose to cover only a small fraction of the variant space, select small feature groups and use a family-based analysis, the analysis would be represented by a point near to **G**. If we choose to cover nearly all variants, with feature groups that are nearly as large as variants and analyze them variant-based, then the point would be near to **A**.

The remaining points of the cube (**F**, **C**, and **H**) also represent strategies which use the dimensions to smaller or larger extent. We do not claim that all corners of the cube are useful in practice. For example using a family-based analysis on single features, point **H**, is probably not sensible. However, there are useful strategies on the ranges between these points.

Some of the described analyses do not consider interactions between features. For example, feature-based type checking provides type errors, type definitions and type requirements for each feature. Each reported type error is based on the code of one feature (e.g., a feature declares a variable as Integer and uses it as an array). The definitions and requirements are lists of types, which are provided by the feature, and which must be provided by other features, respectively. Whether all requirements are actually provided in all variants cannot be determined by a feature-based analysis. To detect feature interactions, a subsequent analysis (e.g., family-based) is required to compare the requirements gained by the first analysis. This second analysis would be represented by a second point in the cube.

3.2 Formal Definition of the PLA Model

Next, we develop a formal definition of the PLA model. In our model, variability-aware analyses consist of various steps that either combine features (*integration step*), or process features or feature combinations to produce a result (*processing step*). The model defines an algebra of expressions that represent *compound analyses* (i.e., combinations of integration and processing steps). We define four operators that can be used recursively to build complex expressions. The operators are based on the dimensions of the PLA cube and denote integration and processing steps. We introduce the operators for two purposes: first, to give abstract descriptions of existing complex analyses and, second, to develop and describe new analyses. Furthermore, we describe briefly our experience with the tradeoffs of each operator.

The operators are general in the sense that they can be used to describe any configurable-system analysis. However, they are not complete; to describe complex analyses, the operators are extended by textual descriptions. The model helps to describe and illustrate the core concepts of the analyses.

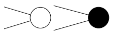
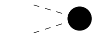
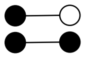
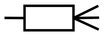
In the description, we use the concepts of a *partial configuration* and a *partial variant*. Partial configurations are selections of features that do not yield a valid configuration, but which can be extended to a valid configuration by *enabling* or *disabling* additional features. For example, in the printing-device system (Section 2.1.1), the partial configuration with features *Copy* and *Base* enabled can be extended to a valid configuration by enabling feature *Scan*. A partial variant is a system generated based on the choices of a partial configuration. The system may not be compilable (e.g., it might miss references to code implemented in other features), and it might still contain variability (e.g., ifdefs on which no choice has been made).

Operators Overall, we define four operators that work either on basic features or on the results of other operators, as illustrated in Table 3.1. We call the input and output items of operators *objects*. For example, an object could be a system variant, a feature, a program with ifdef variability (some of the variability may be resolved), or a variant simulator.

The **fixed combinator** (ρ) takes two objects and combines them to a (partial) variant, in which both objects are fixed. In the resulting variant, both objects are always activated and cannot be deactivated any more. If the objects contain variability (e.g., two existing variant simulators), the variability is preserved in the result (e.g., a composed variant simulator). An example of the fixed combinator is the composition of two feature modules using a code generator, such as FeatureHouse [AKL09]. The result contains the artifacts from both operands, as shown in Figure 2.4, page 19. Normally, features cannot be extracted from the result of a fixed combination. Our experience is that, implementing an analysis that uses only the fixed combinator is relatively easy, because the analyzed object contains little or no variability. This allows reusing existing tools which are not variability aware. The downside is that one needs to build and analyze many variants (exponential explosion) to cover the configuration space. We use solid lines leading to a circle as visual notation for the fixed combinator. The circle is filled iff the resulting object contains variability.

The **variability combinator** (ν) takes two objects and a feature model, and combines the objects into a single one. The combinator retains variability information according to the given feature model. The resulting object is a simulator in which each operand can be switched on and off at a later point in time (e.g., as in Figure 2.9, page 36 at run time). An analysis that uses only the variability combinator with the variability model of the system produces

Table 3.1: Operators of the PLA model. Filled circles denote artifacts which contain variability. $\mathbb{C}$ and $\mathbb{S}$ denote the set of partial variants and simulators, respectively. $\mathbb{F}$ is the set of features, and M is the set of propositional logic formulas on $\mathbb{F}$. S is the set of (partial) feature selections. For a detailed discussion of the signatures, we refer to our original publication of the PLA model [vRAK+13].

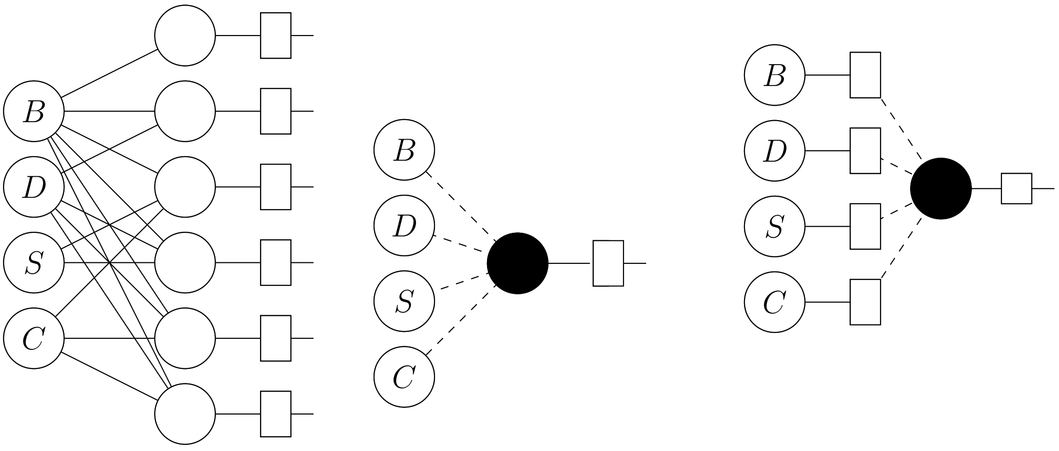
Operator	**Visual**	**Definition**	**Signatures**
Fixed combinator ρ		$\rho_{\mathbb{C}}(A_1, A_2, \ldots)$ $\rho_{\mathbb{S}}(A_1, A_2, \ldots)$	$\rho_{\mathbb{C}} : \mathbb{C}^n \to \mathbb{C}$ $\rho_{\mathbb{S}} : \mathbb{S}^n \to \mathbb{S}$
Variability combinator ν		$\nu(A_1, \ldots, A_n, m)$	$\nu : ((\mathbb{C} \cup \mathbb{S})^n \times M) \to \mathbb{S}$
Variability restriction †		$\dagger_{\mathbb{C}}(A, sel)$ $\dagger_{\mathbb{S}}(A, sel)$	$\dagger_{\mathbb{C}} : (\mathbb{S} \times S) \to \mathbb{C}$ $\dagger_{\mathbb{S}} : (\mathbb{S} \times S) \to \mathbb{S}$
Processing step τ		$\tau(A)$	$\tau_{\mathbb{C}} : \mathbb{C} \to \mathbb{C}^n$ $\tau_{\mathbb{S}} : \mathbb{S} \to \mathbb{S}^n$

Operator	**Description**
Fixed combinator	Combines all operands into a new object, where the operands are fixed
Variability combinator	Generates a variable object from operands (variable or fixed) with variability model $m \in M$
Variability restriction	Restricts the variablity according to a feature selection $sel \in S$ The sink circle is unfilled iff no variability remains
Processing step	Applies a processing step to a fixed or variable object; the results are fixed or variable depending on the input object

one simulator that includes the variability of the entire configurable system. However, one could also use the variability combinator with a limited variability model, generating a simulator for a part of the system. For example, instead of using the variability model $\hat{\Phi}$ of the printing-device system, we could use the limited variability model $\hat{\Phi} \wedge \mathit{Duplex}$ to simulate only variants with the feature *Duplex*. Our experience is that the analysis of simulators is efficient, because the analysis examines shared parts only once [AvRW+13; LvRK+13]. However, the analysis of a simulator is more expensive than the analysis of a single variant, which can cause problems with limited system resources such as main memory [AvRW+13]. We use dashed lines as visual notation for the variability combinator. The lines lead to a filled circle representing an object with variability information.

The operator **variability restriction** (†) is used when existing variability needs to be fully or partially eliminated. The operator takes one argument that contains variability (e.g., a variable piece of code or a variant simulator) and restricts it according to a given feature selection. A well-known example of this operator is the C preprocessor CPP. If one applies CPP to a configurable system that contains features implemented with ifdef directives, CPP eliminates all directives and leaves only the code that correspond to a given feature selection. Another application of variability restriction is to limit model checking of a configurable system to a subset of its configurations (e.g., [tBFGM15]). The variability-restriction operator has essentially the same tradeoffs as the fixed combinator: relatively simple analysis of each resulting object, but to achieve full coverage, potentially many objects have to be analyzed. In contrast to the fixed combinator, which combines multiple objects, variability restriction takes one object with variability and restricts variability. In the visual notation the variability-restriction operator is denoted by a solid line leading from a filled to a filled or unfilled circle, depending on whether the result still contains variability information.

The operator **processing step** (τ) represents an analysis or a pre-processing step that is performed on objects (features or combined structures). The processing step produces results that can, again, be aggregated with other transformation operators. An example processing step takes a feature's source code and filters the code for some properties. For example, we might verify that a LINUX driver correctly uses the spinlock protocol, which ensures mutual exclusive access to resources. A spinlock is a data structure that can be passed to function spin_lock and spin_unlock to obtain or release the lock. When analyzing whether the locking infrastructure is used correctly (e.g., no consecutive locking calls), we are interested only in the calls to the spin_lock and spin_unlock functions. So, we can write a processing step that filters these calls from the source code. The filtered features are then processed with other operators. The main analysis can be executed more efficiently, when uninteresting information has been filtered out as early as possible. Our experience is that the efficiency of the processing step depends on whether it is applied to an object with or without variability and whether the analysis tool is variability-aware (e.g., see Section 6.5). That is, the tool can recognize parts of the analyzed object that are not influenced by variability and analyze these parts only once. We use a box as visual notation for a processing step.



(a) Variant-based (b) Family-based (c) Feature-based and Family-based

Figure 3.2: Three possible operator combinations for the printer system (features are abbreviated with first letter). The operators are always applied from left to right.

Combination of operators The four operators can be combined to form complex and powerful analyses that leverage the advantages of several operators while reducing the disadvantages. Next, we illustrate three example analysis patterns (Figure 3.2), which could be used in the analysis of the printer system (Section 2.1.1).

- The pattern in Figure 3.2a creates all variants by applying the fixed combinator in a brute-force fashion and analyzes the resulting variants individually. This variant-based approach leads to many analysis runs. It corresponds to point **A** in the PLA cube.

- The pattern in Figure 3.2b uses the variability combinator to generate a variant simulator incorporating all features. The analysis of the simulator requires special tools that can cope with the variability information. The variant simulator captures the behavior of all variants and therefore it might get very large and costly to analyze. This family-based approach corresponds to point **D** in the PLA cube.

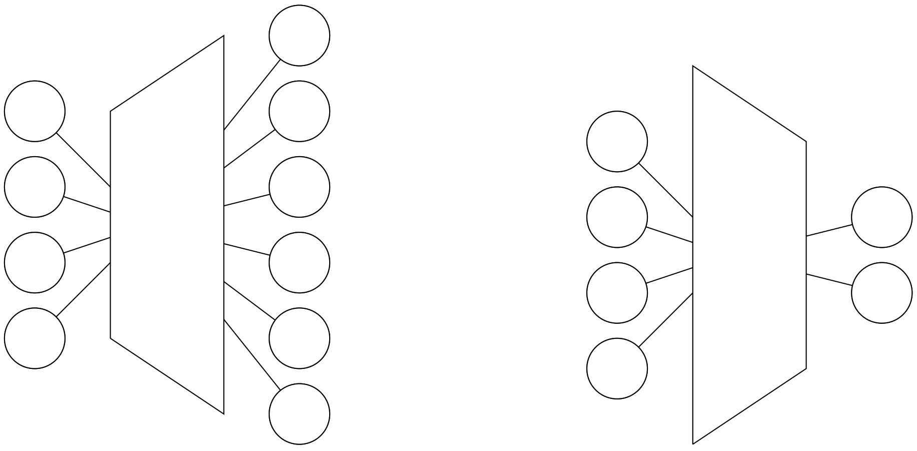
- The pattern in Figure 3.2c applies a processing step on each feature before combining the results with the variability combinator. This can lead to a small simulator that can be analyzed more efficiently. For example, we can use the processing step to filter interesting functions (e.g., locking and unlocking). The simulator would be much smaller and contain only the locking behavior that we are interested in. However, implementing this analysis pattern requires three tools: an abstraction mechanism for filtering, a tool to combine the abstraction results into a simulator, and a tool that can analyze the abstract simulator. Few off-the-shelf tools provide these functionalities, so this analysis requires special tools. The implementation effort causes a higher upfront investment for the right analysis pattern than for the left or the middle one. This approach combines a feature-based step with family-based analysis and, therefore, it is represented by points **E** and **D** in the PLA cube.[1]

Sampling by combining operators Using the operators, sampling can be accomplished in several ways:

- When using the fixed combinator ρ, a sample set can be build by constructing only some valid variants. For example, $\tau(\rho(\mathit{BasicPrinter}, \mathit{Scan}))$ is a sampling analysis of the printer system based only on the variant in which *BasicPrinter* and *Scan* are enabled.
- The variable combinator ν has a feature selection that determines the variability model of the resulting object. This feature selection is used to restrict the analyzed variants to a sample set of interesting configurations. To this end, a restricted variability model that allows only the interesting configurations is used as operand. If only processing steps on features are used, then sampling can be accomplished by analyzing only a subset of features.

Notational sugar We noticed two patterns that occur in many analysis descriptions. Given that they are quite cumbersome to draw, we introduce notational sugar that can be used as a shortcut. The first pattern is the generation of all valid variants from a given set of features. In our model, this pattern is expressed by multiple applications of the fixed operator (shown in Figure 3.2a). As shortcut, we denote this pattern by a trapezoid, as shown in Figure 3.3a. The second pattern is the application of a sampling heuristic where only a subset of all valid variants are generated. We denote this pattern by a reversed trapezoid shape that reflects that only few variants are generated, as shown in Figure 3.3b.

[1] Point **H** represents an analysis that uses the family-based strategy in each feature(-group).



(a) Generate all variants (b) Generate a sample set of variants

Figure 3.3: Simplified notation for recurring analysis patterns for configurable systems. The notation in Figure 3.3a represents the generation of all valid variants and the notation in Figure 3.3b represents the generation of a subset of all valid variants (sampling).

3.3 Categorizing Existing Analyses

In this section, we categorize existing analyses by means of our formal model, to demonstrate the capabilities of our model for systematic description and comparison of analyses for configurable systems. Our experience is that it was relatively easy to express these high-level descriptions of complex variability-aware analyses using our model.

Configurable-system verification with variability encoding In the past, the verification of functional properties of configurable systems was often limited to the verification of abstract models or to the verification of a sample set of variants. In recent years, several researchers [ASW$^+$11; CHSL11; CHS$^+$10; KATS12; TSAH12] have developed analysis techniques that can be summarized as variability-encoding techniques (Chapter 5). They aim at encoding the functional behavior of all valid system variants in one variant simulator and use model checking or theorem proving to verify the simulator correctness. If the simulator can be proved correct, then all variants satisfy the functional properties. Features that are unknown when the simulator is generated cannot be included subsequently, so this is a closed-world approach.

In terms of our model, the techniques use a variability combinator to build the simulator and then use a processing step for model checking (Figure 3.2b). In the work of Classen et al. [CHSL11; CHS$^+$10] and Thüm et al. [TSAH12], the

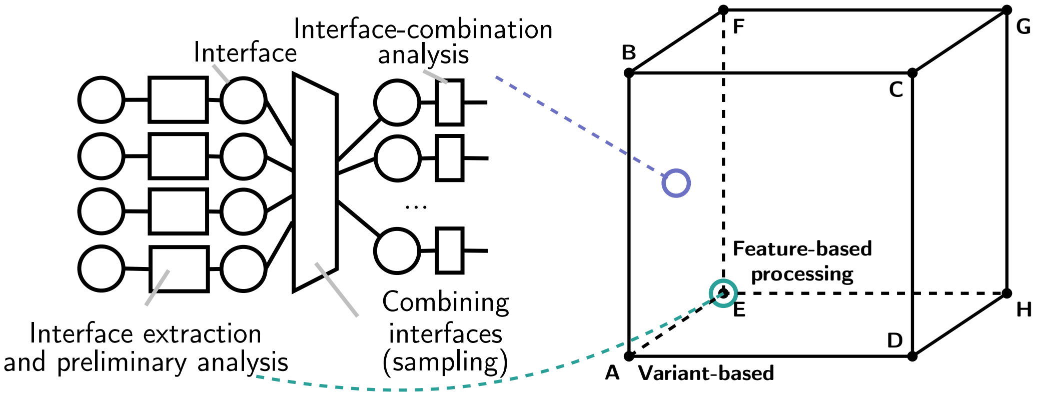

Interface-combination analysis
Interface
Interface extraction and preliminary analysis
Combining interfaces (sampling)
...
B
F
G
C
Feature-based processing
E
H
D
A
Variant-based

Figure 3.4: Pattern of the analysis of Li et al.[LKF05] and the corresponding points in the cube. In the first step, all features are analyzed and interfaces of the features are extracted (feature-based, point **E**). Then, selected combinations of these interfaces are analyzed to find critical feature interactions (point on plane **A**–**B**–**F**–**E**). The dashed lines between the figures indicate which point corresponds to which set of processing steps.

first step was done manually, whereas we used an automatic combination tool (Chapter 5 and Chapter 6). In terms of our model, both analysis approaches are expressed by the schema shown in Figure 3.2b, and they are located on point **D** in the PLA cube (Figure 3.1).

Modular verification of configurable systems Li et al. [LKF05] used modular verification to check the correctness of features in an open-world setting. In an open world, features are developed in isolation, possibly by different teams, which are not aware of each other. For example, when extending a framework, plug-in developers may not know about all other plug-ins in the system [ABKS13]. Yet, the features (and plug-ins) have to satisfy specifications when working together. To prove feature compatibility, Li et al. have used a feature-based processing step, in which they analyze individual features and extract interfaces. During this processing step, they already search for specification violations in the features. Then, the feature-based interfaces are aggregated in a combination step (fixed combination). In most cases, only two feature-interfaces are combined, so the results are not necessarily variants. The resulting abstractions of feature combinations are analyzed to detect feature interactions that cause violations of the the configurable-system specifications. This approach incurs less effort than analyzing all concrete variants or all concrete combinations of features. This analysis approach can be expressed in our model as shown in Figure 3.4. The analysis uses two sub analyses. The first

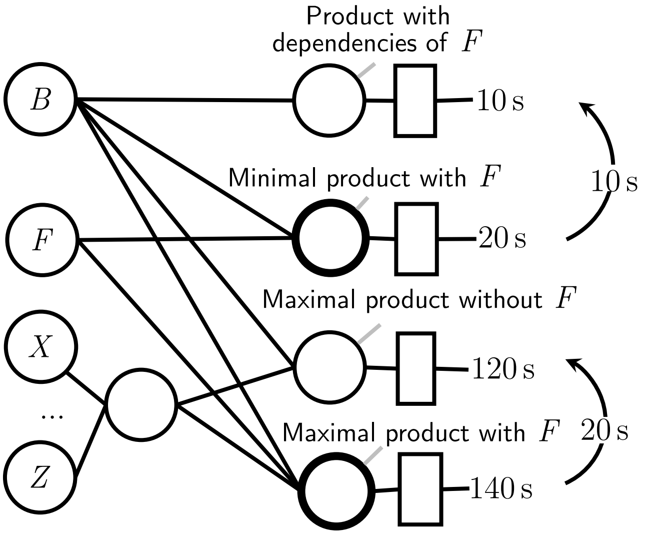


Figure 3.5: Pattern of the analysis of Siegmund et al. [SKK+12]. F denotes the feature under analysis, B is the base variant F depends on, and X to Z are all other features. The circles with bold frame denote variants that contain feature F.

sub analysis is feature-based (point **E** in the cube). The second sub analysis is analyzing a sample set of groups of feature interfaces. It is represented by a point on the plane **A**–**B**–**F**–**E**.

Detection of non-functional feature interactions Siegmund et al. [SKK+12] developed an analysis technique to automatically detect feature interactions based on performance measurements. In this context, a feature interaction between two features is a change in application performance that is only measured when both features are present. For example, a feature F contributes a minimum of 10 seconds to the total run time of a system. Another feature X has no influence on the run time when used without F. However, in each variant which contains both, *F and X*, they interact and consume 20 seconds. For their analysis, Siegmund et al. used a combination of feature-based and variant-based sampling strategies, which is shown in Figure 3.5. In the PLA cube (Figure 3.1), it is located on the plane between points **A**, **B**, **F** and **E**, because grouping and sampling are combined. The analysis determines whether there is an interaction between feature F and any other feature in the configurable system, and it calculates the set B of features that F depends on. Then, it measures the performance of, (1) the variant that contains only the features B, and (2) the variant that contains the features F and B. The difference between these measurements is used as prediction for the performance

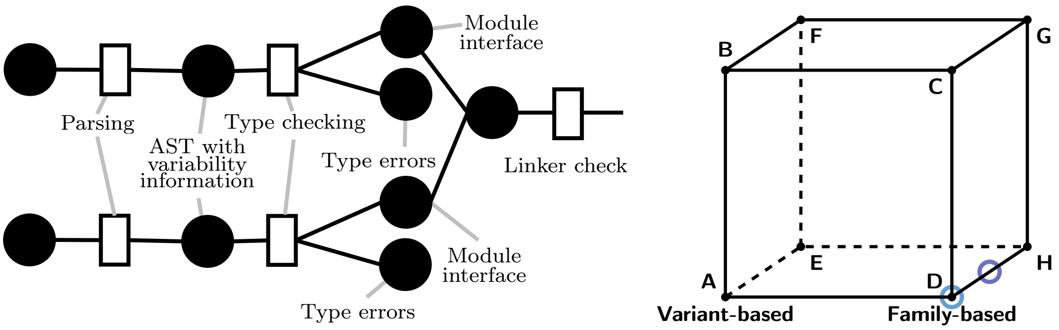


Figure 3.6: Visualization of an analysis by Kästner et al. [KOE12] and the corresponding point in the cube. X and Y denote the files under analysis. Both files contain variability information (in ifdef annotations). The analysis executes parsing and type checking on the files and then checks the compatibility of the inferred type interfaces.

of F. In this example, the influence which the feature F has on performance is estimated to be 10 seconds. To detect interactions, the analysis builds another pair of variants that contain more features (again the only difference between the variants is F). If the difference between the measurements of the second pair (20 seconds in this example) is different from the first pair, then there exists an interaction between F and some other feature. The analysis is sample-based, because it uses a systematically chosen set of four variants. To determine which features do interact, one can systematically evaluate more pairs of variants [SKK$^+$12].

Variability-aware type checking Kästner et al. [KOE12] developed an analysis for type checking of large annotation-based systems in C. The authors used the analysis to determine type- and linker errors in Busybox. The analysis can be expressed in terms of our formal model as shown in Figure 3.6. Each file contains C code with variability annotated in cpp directives. As first step, Kästner et al. parse and type-check each file separately using the family-based strategy. For each file, this yields an abstract syntax tree (AST), an intermediary representation of the hierarchical syntactic structure of the code file [ALSU06]. The ASTs are then type checked separately. The type checking step reports possible type errors and generates a type interface (imported and exported symbols) of the file. As each file of the Busybox system uses only part of the configuration options, this step is based on feature groups (a group of features per file). In the second step, the type interface is checked for compatibility with type interfaces of other files (linker check). The analysis uses a fixed combinator (interface combination) on objects containing variability.

In terms of the cube, the parsing and type-checking step uses a family-based analysis on feature-groups and it is represented by a point between **D** and **H**. The second step (dependency checking between files) is a family-based and represented by point **D**.

3.4 Related Work

The PLA model is based on three basic analysis strategies (feature-based, variant-based, and family-based), which have been documented in a recent literature survey [TAK+14]. This survey also discusses simple combinations of the strategies, but it does not take the step to formalize them in a coherent model, such as the PLA model. Furthermore, the survey discusses which types of configurable-system analyses have been published and, more importantly, how they are compared to other strategies. In our research, we showed that the family-based strategy often outperforms variant-based and sample-based strategies. We assume that this efficiency is the reason why most publications that were considered in the survey publish family-based strategies. However, almost a third of all considered approaches rely on generating *all variants*, which is infeasible on large configurable systems. This suggests that our model might be useful to raise the awareness of complexity problems and possible solutions.

The survey of Thüm et al. [TAK+14] also contains an overview of publications that compare different strategies. In many studies, a particular (optimized) strategy is compared against an unoptimized variant-based strategy (e.g., [CEW14; CHS+10; CCP+12]) or a sample-based strategy (e.g., [AvRW+13; LvRK+13]). The main conclusion of the survey is that the field of configurable-system analysis is broad and diverse. Some strategies have been used very often (most notably the family-based strategy) and other strategies are under-represented (e.g., the feature-based strategy).

Interestingly, there is evidence that even sampling does not scale to very large configurable systems. In a study, Liebig et al. generated sample variants of very large configurable systems (e.g., Linux) to compare sample-based analyses against family-based analyses [LvRK+13]. They used different coverage criteria, such as code coverage or pair-wise coverage, to generate the sample sets. It turned out that the generation of the sample sets required significant effort and time compared to the actual analysis. In many cases the family-based analysis was faster than the generation of the sample variants. Furthermore, the sample sets, by design, only covered a very small part of the configuration space of the configurable systems.

3.5 Summary and Outlook

The PLA model is a guide line for researchers investigating configurable-system analysis. It is build on reoccurring patterns that we identified in analyses for configurable systems. We demonstrated the usefulness of the model for the comparison of analyses by means of existing configurable-system analyses (Section 3.3). Our initial experience with the model suggests that it is helpful for the description of complex analyses. Of course the model is rather abstract as we do not focus concrete sampling heuristics or implementation mechanisms (e.g., module-based or annotation-based variability). Despite the numerous points in the cube which we illustrate with examples, most of the cube is unexplored.

Consequently, one interesting next step is to investigate different planes on (and in) the cube to establish how combinations of the dimensions influence analysis performance (e.g., run time, efficiency or memory consumption). Existing studies have mainly compared two or three strategies against each other. However, to isolate the effects of different combinations of strategies, one would choose a representative set of strategies along the investigated dimensions. Comparison of these strategies (in the same environments such as case studies, implementation frameworks, etc.) would yield insights how analysis performance evolves when dimensions are used and combined to more or less extent. For example, one could develop an analysis that combines either all pairs, or all triples of features to feature groups and runs type checking on these groups. The type checking could be done variant-based, which would place the analysis on plane **A**–**E**–**F**–**B**. Alternatively, the analysis could be run family-based (plane **D**–**H**–**G**–**C**).

In this thesis, we explore some parts of the cube, however it is beyond the scope of this thesis to provide complete experimental coverage of the cube. Instead we focus on interesting and promising points which we summarize in the following paragraphs (illustrated by Figures 3.7, 3.8, and 3.9).

Variant-based, family-based, and sample-based model checking We developed an approach for family-based model checking, which we describe in Chapter 6. We evaluated our approach against variant-based and sample-based model checking (Section 6.5). In this evaluation, we explore several dimensions of the cube as illustrated in Figure 3.7: Our family-based model-checking approach is represented by point **D** and we compare it to the variant-based strategy (point **A**) and to three different sample-based approaches (between points **A** and **B**). Our evaluation shows that family-based model checking is more efficient than variant-based and sample-based model checking.

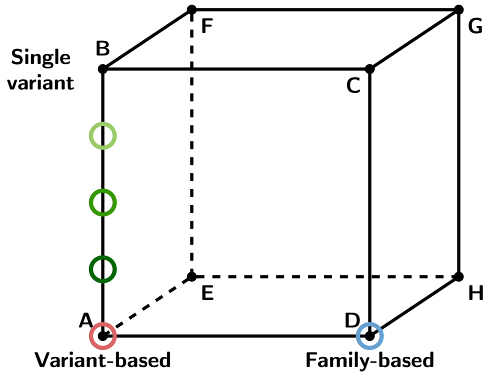


Figure 3.7: Family-based, variant-based, and sample-based model checking (described in Section 6.5)

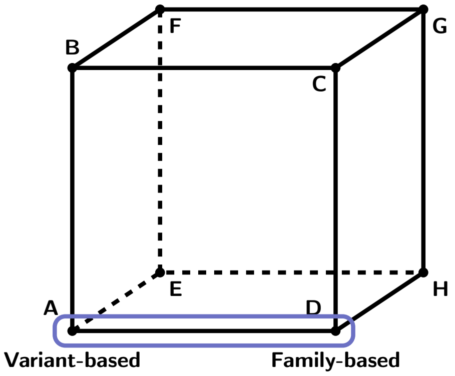


Figure 3.8: Model checking with a variant-based *and* family-based strategy (described in Section 6.6)

Combining family-based and variant-based model checking To explore range **A**–**D** of the cube, we implemented a model checking approach that is based on a partitioned family of system variants (Section 6.6). Each partition of the family represents a set of variants which is encoded in a variant simulator. Each simulator is then verified and, together, the partition-based results yield correctness information for the entire system.

As each system family can be partitioned in many possible ways, we evaluated all possible partitioning strategies for a set of example systems. Therefore, our evaluation represents the whole range of strategies from point **A** to point **D** (illustrated in Figure 3.8). Our evaluation (Section 6.6.3) shows that combined strategies lead to lower memory consumption (compared to the family-based strategy) and faster verification (compared to the variant-based strategy).

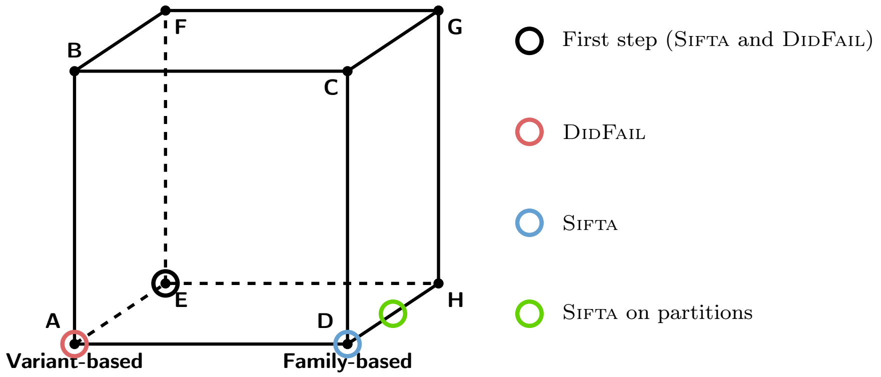


Figure 3.9: Different strategies for analysis of large sets of Android apps (described in Section 7.4)

Analyzing large sets of Android apps In Section 7.4, we describe an analysis that applies feature-based processing to Android apps and analyses the results in a family-based fashion (Figure 3.9, points **E** and **D**). We implemented this analysis is implemented in our tool Sifta [vRBS+15]. We compared this feature-based and family-based analysis with a related analysis that uses the same feature-based step, but then uses a variant-based strategy (DidFail [KFB+14], points **E** and **A**). Furthermore, we improved our analysis by applying it's second, family-based step on partitions of the app set. Each partition represents a groups of apps and therefore, this strategy is represented by the green point in Figure 3.9. This improvement reduces the main memory needed during the analysis and makes it applicable to very large sets of apps (51 935 apps in our experiment, Section 7.5.2).

CHAPTER 4

Presence-Condition Simplification

This chapter shares material with the publication "Presence-Condition Simplification in Highly Configurable Systems" in *ICSE'2015* [vRGA+15].

In the studies that lead to this thesis, we designed, implemented and evaluated several analyses for configurable systems (e.g., Chapters 6 and 7). In variability-aware analyses, presence conditions are usually of central importance because they denote in which configurations interesting situations (e.g., a defect) occur. However, we experienced that presence conditions are often very complex. In fact, they are often *more* complex than necessary because they contain redundant or uninteresting information. In this chapter, we describe this problem formally and present a solution called *presence-condition simplification*. We also present the results of our evaluation of three algorithms for presence-condition simplification on real-world use cases and subject systems. All three algorithms have been introduced by other researchers in different fields of computer science.

In our work on variability-aware verification, we found many presence conditions denoting conditions for detected defects in analyzed systems (described in Section 6.2). For example, our tool SPLverifier (described in Chapter 6) reports the following defect in the E-Mail system (cf. Section 2.1.4):

```
Specification 11 violated on condition
  encrypt && decrypt && keys &&
  ((sign && verify && base && autoresponder) ||
  (!sign && !verify && base && autoresponder))
```

When receiving this verdict, a user has to understand how the defect occurs and which configuration options contribute to the defect before working on fixing it. Simplifying the presence condition and identifying the responsible options help in fixing the defect [AvRW^{+}13]. A closer inspection of this example reveals that the defect is caused by an interaction of only two options, *encrypt* and *autoresponder*. All other parts of the presence condition (e.g., *encrypt* requires *keys*) are redundant because they are already implied by the variability model. The model-checking process introduces such redundant parts because, as an optimization, it analyzes only configurations that are valid with respect to the variability model (cf. Section 6.2.4). A desirable simplification of the presence condition is to identify *encrypt* and *autoresponder* as the sole cause of the defect and to "hide" the redundant parts. In our example, the simplified report is as follows:

```
Specification 11 violated on condition
  VariabilityModel && (encrypt && autoresponder)
```

A straightforward way to simplify a presence condition is to find the smallest, but equivalent expression. This problem is known as the *minimum-equivalent-expression* problem [BU11; HS11]. Although finding a minimal equivalent expression might reduce the size of the presence condition, the minimal equivalent expression is still larger than necessary. One reason for unnecessarily large expressions is that variability-aware analyses, such as our model-checking approach (Section 6.2), typically consider only configurations satisfying the variability model. Thus, the variability model is often an integral part of every reported presence condition, even though the condition describes only a local situation or fact. Since the variability model must be satisfied globally, this information obfuscates the presence condition.

Our goal is to simplify a given presence condition such that it becomes smaller and can be used instead of the original presence condition. We explicitly do *not* aim to preserve equivalence of the simplified and the original presence condition. In the example above, we want to remove the constraints already enforced by the variability model from the presence condition and show only the rest to the user. This rest must be satisfied *in addition to* the variability model to reach the situation of interest (e.g., it identifies the source of the defect).

problem definition

To this end, we introduce the *presence-condition-simplification problem* and present a formal definition of the problem. We are interested in a function simp(p, m) such that the expression $p' = \text{simp}(p,\ m)$ is equivalent to the presence condition p under all assignments that satisfy the *context* m of the presence condition: $m \Rightarrow (p' \Leftrightarrow p)$. In addition to this invariant, the size of p' should be as small as possible (we define a size measure in Section 4.2). The presence condition p and the context m are Boolean expressions, and we require that p is embedded in m, which means that p is evaluated only if m is satisfied. A simplification function 'simp' receives two Boolean expressions p and m and returns a Boolean expression.

Besides simplification of presence conditions in tool reports, there are many other application scenarios such as the simplification of ifdef preprocessor directives and the simplification of cross-tree-constraints in feature models. We included these scenarios in our evaluation (Section 4.4). Based on our work, presence-condition simplification has been integrated in the variability-aware analysis tool TypeChef [KGR+11].

solution algorithms

We developed a brute-force algorithm as solution for the presence-condition-simplification problem (simp_{BF} in Section 4.3). However, this algorithm has an exponential complexity, so we cannot use it in realistic application scenarios. Instead, we identified and adopted three heuristic algorithms from the area of circuit optimization. We apply these algorithms to solve the presence-condition-simplification problem (although the solutions are not guaranteed to be optimal). To the best of our knowledge, the algorithms have not yet been applied to the simplification of presence conditions before. The first algorithm, Restrict [CBM90], is based on BDDs [Bry92]. The second and third algorithms are solutions for two-level logic minimization: the Quine-McCluskey algorithm [McC56; Qui52] and the Espresso algorithm [BSMH84]. We discuss the algorithms and how they are adopted in Section 4.3.

To compare the three algorithms and to explore their feasibility and effectiveness for presence-condition simplification, we ran a series of experiments on three application scenarios and 29 subject systems (Section 4.4). We evaluated processing time and size reduction of presence conditions for the three algorithms. Our results show that presence-condition simplification can achieve substantial improvements in reasonable time for various realistic application scenarios. For example, in an experiment where we simplified analysis results of family-based verification (E1, Section 4.4.1), one simplification algorithm (simp_{BDD}) reduced the size of presence conditions by 59%, on average. Furthermore, we analyzed how the simplification algorithms scale with the increasing complexity of the input expressions. We provide a replication package for our experiments and further detailed results on the supplementary website.

4.1 Application Scenarios

Although the application domain is much broader, we are particularly interested in presence-condition simplification in the context of developing and analyzing highly configurable systems. Next, we illustrate three application scenarios.

4.1.1 Reporting Analysis Results

Variability-aware analyses often report the condition under which certain events or states occur as presence conditions. Recall the example presence condition from the chapter introduction. The presence condition is reported by SPLverifier (Chapter 6,[AvRW$^+$13]) for a violation of a specification in the E-Mail system. Such violations indicate either an incomplete variability model, which should be fixed to prevent defective configurations, or a defect in the system.

Since SPLverifier verifies only configurations that satisfy the variability model, which is standard in variability-aware analyses [TAK$^+$14], the reported presence conditions may contain facts that are already implied by the variability model. This mix of defect condition with variability-model constraints hinders understanding and pinning down the source of a defect. Even though the E-Mail system has only nine configuration options, the reported defect conditions are often unnecessarily complicated. The defect conditions we encountered for the E-Mail system have between 5 and 17 literals. As shown in the introduction, we yield a presence condition containing only 2 instead of 11 literals[1] for the presence condition shown on page 57:

```
Specification 11 violated on condition
  VariabilityModel && (encrypt && autoresponder)
```

4.1.2 Simplification of Variability Annotations

Variability annotations are directives in a system's source code that conditionally include or exclude parts of the code (cf. Section 2.1.3). We focus on two implementation mechanisms for variability annotations: conditional inclusion of files in build scripts and #if preprocessor directives (described in Section 2.1.3). Figure 4.1 shows an example of both mechanisms used together, taken from the Linux kernel v3.4. Figure 4.1a shows an excerpt of the Makefile in the kernel directory. It states that the object file of lockdep.c is included if option LOCKDEP is enabled. Figure 4.1b shows an excerpt of file lockdep.c, which contains several #if directives.

nested variability

Observe that, in the example, the innermost #if directive (Figure 4.1b, Line 1301) is enclosed by two conditions: the #if condition in Line 826 and the condition from the Makefile. The conjunction of both enclosing conditions is the context of the condition in Line 1301: $m = (\mathsf{LOCKDEP} \wedge \mathsf{PROVE\_LOCKING})$ and

[1] We do not count the variability model as a literal because users know that it is enforced. We added it to the simplified presence condition for the sake of completeness.

```
32 ...
33 obj−$(LOCKDEP) \
34  += lockdep.o
35 ...
```

(a) Excerpt from kernel/Makefile

```
826  #if defined(PROVE_LOCKING)
827  ...
1301 #if defined(TRACE_IRQFLAGS)
1302  && defined(PROVE_LOCKING)
1303 ...
1674 #else
1675 ...
1688 #endif
1689 ...
2146 #endif
```

(b) Excerpt from kernel/lockdep.c

Figure 4.1: Nested variability annotations with redundancy

$p = (\mathsf{TRACE\_IRQFLAGS} \land \mathsf{PROVE\_LOCKING})$. Using presence-condition simplification, we can remove the redundant term PROVE_LOCKING from the condition of the inner preprocessor directive without changing the behavior of any variant of the LINUX kernel: $\text{simp}(p,\ m) = \mathsf{TRACE\_IRQFLAGS}$.

Admittedly, the expressions involved in this example are relatively simple, so a developer might be aware of the redundancy and leave it for documentation. Still, in more complex cases, simplification can be more effectful, especially because it is also beneficial for tools working on the code to ease automatic reasoning.

Automatic code analysis of systems with ifdef variability is difficult because the preprocessor directives can be interleaved with normal C code in complicated ways. The tool TYPECHEF [KGR+11] solves this problem by providing a variability-aware parser for C code with ifdef directives. It is used in many research projects [KvRE+12; LvRK+13], which would benefit from presence condition simplification. TYPECHEF resolves preprocessor directives and macros, and it generates an abstract syntax tree (AST), preserving the variability induced by ifdef directives. Technically, nodes in the AST are annotated with the presence conditions that correspond to the ifdef directives. Due to approximation in the parsing process (e.g., macro expansion) [KGR+11], these presence conditions are often an overapproximation of the actual presence conditions and contain redundancy. We can make the AST generated by TYPECHEF more concise by simplifying the presence conditions with their context (presence conditions of ancestors in the AST conjoined with the presence condition of the file), which improves the performance of subsequent analyses, such as type checking or data-flow analysis [LvRK+13].

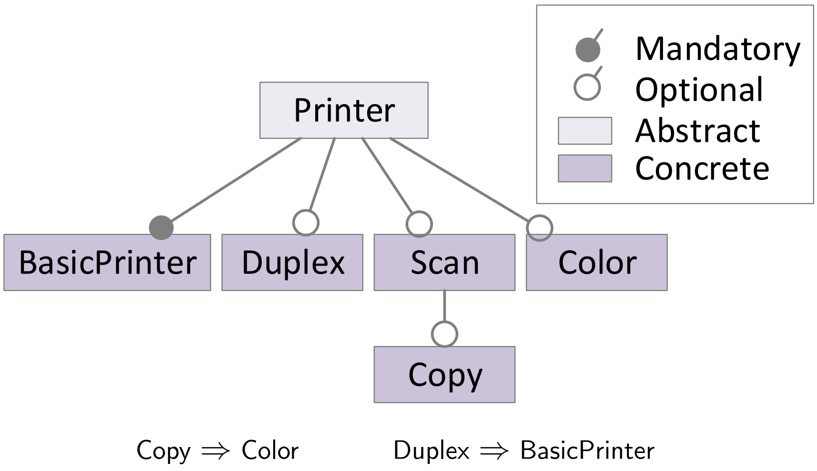


Figure 4.2: Extended feature model of the printing-device system

4.1.3 Variability-Model Generation

A variability model can be expressed in different encodings. In this work, we use Boolean expressions, but other scenarios require richer representations, such as feature models by Kang et al. [KCH^{+}90]; Figure 4.2 shows a representation of the feature model of an extended version of the printing-device system.

Such a model contains a hierarchy that shows dependencies between the configuration options (child–parent implication). For example, selecting option Copy implies selecting its parent Scan. Constraints that cannot be encoded in the hierarchy are written as separate *cross-tree constraints*. For example, the dependency from Copy to Color is expressed as cross-tree constraints (we introduced this dependency as illustration).

If a variability model is given as a Boolean expression (for instance, when extracted from source code [SLB^{+}11]), it is sometimes desirable to transform it into a visual model for presentation. There are a number of approaches (e.g., [SLB^{+}11]) that synthesize a hierarchy (shown as tree in Figure 4.2) and constraints between siblings in the hierarchy. All constraints that cannot be encoded in the hierarchy or as sibling constraints are added as cross-tree constraints.

If the cross-tree constraints are still complex, it is advisable to simplify them using the hierarchy and the sibling constraints as context. That is, the cross-tree constraints should not restate the dependencies covered by the hierarchy or the sibling constraints. For example, the cross-tree constraint Duplex ⇒ BasicPrinter in Figure 4.2 is redundant as this dependency is already documented in the tree (BasicPrinter is mandatory). Given the hierarchy constraints h, the sibling constraints s, and the cross-tree constraints ctc, we can simplify the cross-tree constraints with simp(ctc, $h \wedge s$). The result of simplification can replace the original cross-tree constraints, because the context of hierarchy and sibling constraints always hold.

4.2 Problem Formalization

As said previously, the function simp(p, m) takes two arguments: a presence condition p and a context m. The goal is to represent the *relevant* information in p as concise as possible. Information is relevant if it cannot be derived from the context of p. The input parameters and the result of the simplification are Boolean expressions. We explore the problem and the described algorithms (Section 4.3) only in the configurable-systems context, even though they are not limited to this area (the algorithms are applied on general Boolean expressions which are used in many areas of computer science).

non-tautology context

We assume that the context m is available and holds significant information on the situations in which p can be evaluated. If it is not available, or if it represents a tautology, then we have to assume that all information in p is relevant to identify the situation or fact that p represents. In this case, the only possibility to improve the presentation of p is to generate a *minimum equivalent expression* for p [BU11; HS11]. However, in the scenarios that we focus on, usually a substantial, non-tautology context is available (e.g., a variability model).

Presence conditions are meant to be evaluated only if the context m holds. The information encoded in a presence condition p is essentially the set of implicates of p.[2] Elements of this set can be categorized as follows: An implicate of p is either (1) also an implicate of m or (2) not an implicate of m. Implicates in group 1 are redundant and can be dropped because they are already implied by the context. Some implicates in group 2 are implied by elements of group 2 conjoined with m and are therefore also redundant. If we can extract the essential, non-redundant elements of group 2 and present them as a replacement for p, this would be sufficient, because the context m guarantees that the implicates in group 1 are satisfied. Hence, we do not search for an equivalence-preserving function, but we aim at removing implicates from p if they are redundant with respect to m and if they increase the size of p.

The set diagram in Figure 4.3 illustrates the relationship between p, m, and simp(p, m) in terms of the configuration space of a configurable system. The white rectangle m represents the set of all valid configurations. The rectangle p represents the set of configurations denoted by the presence condition. p

[2] An implicate of an expression p is an expression that is implied by p. For example, $A \vee B$ is an implicate of $A \wedge B$ because $(A \wedge B) \Rightarrow (A \vee B)$.

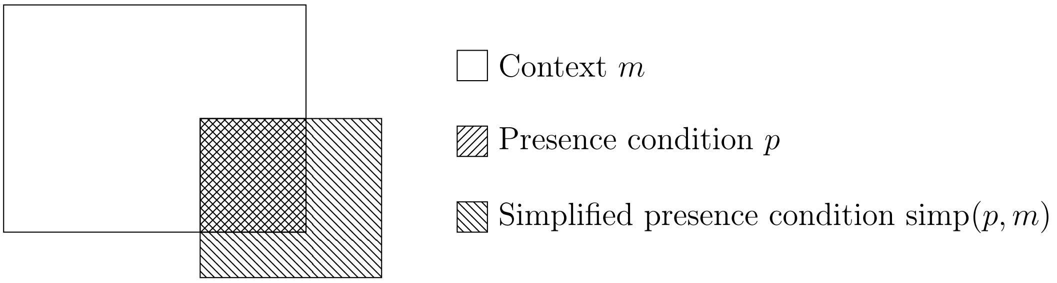


Figure 4.3: Illustration of presence-condition simplification. Each point on the plane represents a configuration. The crosshatched area ▩ denotes the overlapping of the area of p ▨ and simp($p,\ m$) ▧. The simplified presence condition can include configurations that are not included in the context if it helps reducing the size of the condition.[3]

encloses only configurations that are in m. Also, p is (often) smaller than m because it specifies a certain local condition within the global space of configurations. The rectangle simp($p,\ m$) represents the simplified presence condition. It encloses all configurations of p, but also configurations outside of m, if it helps to remove implicates from the expression (i.e., if it reduces the size of p). The objective is that the area of simp($p,\ m$) represents a more concise expression than p.

Formally, the invariant of simp($p,\ m$) is:

$$m \Rightarrow (\mathrm{simp}(p,\ m) \Leftrightarrow p) \tag{4.1}$$

This invariant states that, in the context of m, the expressions p and simp($p,\ m$) are logically equivalent. Therefore, we can use simp($p,\ m$) as replacement for p, provided that m holds.

Equation 4.1 is a sufficient condition for the correctness of replacing all occurrences of p in the context m by simp($p,\ m$). In the simplest case, simp($p,\ m$) $= p$ would be a valid solution. However, our goal is to *simplify p*. So, we define an objective function stating that simp($p,\ m$) must be minimal according to a given measure size:

$$\forall x : \big(m \Rightarrow (x \Leftrightarrow p)\big) \Rightarrow \big(\mathrm{size}(\mathrm{simp}(p,\ m)) \leq \mathrm{size}(x)\big) \tag{4.2}$$

Defining a general measure for the size of Boolean expressions is not reasonable as it depends on the application scenario. In the application scenarios we

[3] A condition that includes more configurations can be smaller in size than a condition with more implicates.

are interested in (cf. Section 4.1), conciseness of expressions is most important, because they are usually presented to the user. In other cases, expressions are used to generate hardware circuits, for which other optimization goals are needed (driven by hardware cost or minimization of signal run times).

In practice, we have to compare formulas given in notations with different constraints (e.g., CNF, DNF, or BDD) because different simplification algorithms (described in Section 4.3) have different input and output formats. To avoid bias of different notations, we focus on the complexity of the encoded formula. To this end, we convert all expressions to a canonical normal form before comparison. As canonical form, we choose a reduced if-then-else normal form (derived from BDDs) that contains only $\land$, $\lor$, and $\neg$ as operators.

After the expressions are converted to the same notation, there are several possible size measures for comparison. We choose the number of occurrences of literals as size measure because it represents the total expression length and is not influenced, for instance, by lengths of variable names. So, for the remaining sections of this chapter, we define the measure size(y) as the number of occurrences of literals in the string representation of an expression y in canonical form. For expression $y = (A \land B) \lor (\neg A \land C)$, size($y$) $= 4$ (B,C, and twice A).

We also evaluated the number of operators and the number of nodes in a BDD representation as alternative size measures, but observed no major deviations in our experiments (cf. Section 4.4). In their work on the minimum-equivalent-expression problem, Hemaspaandra and Schnoor [HS11] have also used the number of occurrences of literals and the number of operators. We decided against measures such as the depth of an AST of the formula, because a CNF/DNF representation would always have depth 2, which renders the measure useless for our purpose.

4.3 Algorithms

In this section, we introduce four algorithms solving the presence-condition-simplification problem: Brute-Force (simp$_{BF}$), Restrict (simp$_{BDD}$), Espresso (simp$_{E}$), and Quine-McCluskey (simp$_{QC}$). simp$_{BF}$ finds an optimal solution, but it iterates over all possible solutions, which is inefficient. The other three algorithms employ heuristics to reduce computational effort while still satisfying the invariant of Equation 4.1.

Except for the very simple simp$_{BF}$ algorithm, we did not develop the solution algorithms ourselves and treat them as black boxes. Therefore we limit our description of the algorithms to basic facts and refer to detailed descriptions in textbooks for further details.

Algorithm 4.1: BRUTE-FORCE (simp_{BF})

```
Data: Expr $p$, Expr $m$
Result: Expr $min$
$min = p$;
for $clause\_set \in \mathcal{P}(\text{clauses}(\text{CCNF}(p)))$ do
    $s = \bigwedge(clause - Set)$;
    if $\big(m \implies (s \Leftrightarrow p)\big) \wedge \big(\text{size}(s) < \text{size}(min)\big)$ then $min = s$;
end
return $min$
```

Naive solution The BRUTE-FORCE (simp_{BF}) algorithm enumerates all implicates of p as shown in Algorithm 4.1. Technically, it uses the clauses of the canonical conjunctive normal form (CCNF) of p. Then, it builds the powerset of these clauses. For each element of the powerset, the algorithm tests whether it satisfies Equation 4.1 and therefore qualifies as a solution. From all possible solutions, the algorithm selects an optimal solution according to the size measure.

The CCNF has 2^n clauses for n configuration options. Therefore, the size of the powerset of the set of clauses is 2^{2^n}, and we have to iterate through the entire set. Due to its computational complexity, we cannot use simp_{BF} in our experiments.[4] However, we use it as theoretical baseline for the complexity of the optimal solution of presence-condition simplification.

BDD simplification The second algorithm was first described by Coudert and Madre [CBM90] in 1989 as the RESTRICT algorithm (simp_{BDD}). The RESTRICT algorithm takes two expressions p and m represented as BDDs and generates a third BDD $c = \text{simp}_{BDD}(p, m)$ that satisfies the invariant of Equation 4.1 [CBM90]. Basically, RESTRICT compares branches of the BDDs p and m recursively and projects p to BDD nodes that occur only in p and not in m. The algorithm is intended to minimize the number of nodes in the BDD representation of $\text{simp}_{BDD}(p, m)$. This is in line with our optimization goal, but as the algorithm uses heuristics, it does not always generate optimal results. In the original publication of the algorithm [CBM90], it is described only in prose. Therefore, we illustrate the algorithm with a pseudocode implementation that we provide in the Appendix (page 224), due to its size. For further details, we refer to the original publication [CBM90] and to the supplementary website.

Like many other BDD algorithms, simp_{BDD} is a polynomial-time graph-manipulation algorithm (if caching is used). In the worst case, the size of the graph may be exponential in the number of the variables, which renders the

[4] Our implementation works for up to four configuration options.

algorithm also exponential in the number of variables. However, in practice, many useful Boolean functions have compact BDD representations [GPFW97].

Two-level logic minimization The third solution is to transform the problem of presence-condition simplification into a *two-level-logic-minimization* problem [CS02], which can be solved with the Quine-McCluskey and Espresso algorithms (simp_{QC} and simp_{E}). The attribute "two-level" arises from the fact that input expressions are expected in DNF, and a DNF has two levels (the global level with $\vee$ operations and the clause level with $\wedge$ operations). Two-level logic minimization receives an expression f and a second expression dc, which represents a *don't-care set*. The expressions divide the entire space of option assignments into three partitions: (1) the set of assignments for which $f \wedge \neg dc$ is satisfied, called the *ON set*, (2) the set of assignments for which $\neg f \wedge \neg dc$ is satisfied, called the *OFF set*, and (3) the *DC set* for which dc is satisfied. The result of two-level logic minimization is a simplified version of f.

Mapped to our problem, expression f represents the presence condition p. DC describes variable assignments for which the result $\text{simp}_{E}(p,\ m)$ need not be equivalent to p. In our case, these are all variable assignments that are not valid in the context ($\neg m$). That is, DC is the piece of information needed for minimization. So, the setup $f \equiv p$ and $dc \equiv \neg m$ satisfies Equation 4.1.

Quine-McCluskey

Two-level logic minimization can be exact or heuristics-based. An exact algorithm determines the minimal set of prime implicants needed to represent f without respecting dc. It can be solved with the Quine-McCluskey algorithm, which has exponential time complexity (the problem it solves is NP-hard). In a nutshell, the algorithm starts with computing all prime implicants for the union of the ON and DC sets. Finding the smallest set of these prime implicants that still cover f is basically a set-covering problem, which is NP-hard.[5] The algorithm uses reduction techniques and a branch-and-bound strategy to solve this problem [CS02].

Espresso

For performance, several heuristics have been developed. The most prominent heuristic-based algorithm is the Espresso algorithm [CS02], which utilizes a local search without generating all prime implicants. It is composed of three main operations: *expand*, *reduce*, and *irredundant*. The operations *expand* and *reduce* are applied to improve the current term during optimization, and the operation *irredundant* is used to get out of a local minimum. In our experiments, we evaluated the Espresso algorithm, denoted with simp_{E}, and the Quine-McCluskey algorithm, denoted with simp_{QC}. For further details on the algorithms, we refer to an overview paper [CS02].

[5] The decision version of set covering is NP-complete and the optimization version is NP-hard [CSRL01; KV07].

4.4 Evaluation

We evaluated simp_{BDD}, simp_E, and simp_{QC} guided by two research questions:

RQ1 Is presence-condition simplification able to reduce the size of presence conditions with a known context substantially?

RQ2 How does the processing time of the algorithms simp_{BDD}, simp_E, and simp_{QC} scale to complex simplification tasks?

We evaluated these research questions on the application scenarios described in Section 4.1 on, overall, 29 example configurable systems. As a measure of simplification (RQ1), we compared the number of occurrences of literals in the expression before and after simplification (see Section 4.2). To ensure a fair comparison, we transformed the results generated by the simp_E and simp_{QC} algorithms to BDDs after the algorithms have terminated. This step ensures that the compared result strings are compact and have the same variable order. Such comparison would not be required in a practical application and therefore we do not include the time needed for this transformation in our measurement.

The processing time (RQ2) is measured per simplification task. Preliminary experiments showed that in simp_{BDD}, most time is consumed while building the BDDs representing the expressions (presence condition and context); the actual simplification operation is very fast (cf. Appendix, page 226). For simp_E and simp_{QC}, we could not measure loading and simplification of the expressions separately. To ensure fairness, we did not call simp_{BDD} on in-memory BDDs, but wrote the expressions to a file, invoked simp_{BDD} in a new process, and measured the time for that process to terminate. This time includes parsing the presence condition and context, and writing the result expression.

In total, we designed five experiments, E1 through E5. To evaluate research question RQ1, we needed sets of Boolean presence conditions and contexts from different application scenarios and configurable systems. We obtained these sets from different research projects which we discuss in Section 4.4.1. E1 and E2 represent variations of the "Reporting Analysis Results" application scenario. E3 and E4 apply the "Simplification of Variability Annotations" scenario to source code and to the internal code representation in TypeChef, respectively.

To evaluate research question RQ2, we needed a setting where we can flexibly control the size of the problem. We chose to evaluate this question with the "Variability-Model Generation" scenario and used the SPLOT[6] variability-model generator [MBC09] to create simplification tasks. In the generator, we can increase the number of generated variables and therefore generate harder problems. We used these generated tasks in E5 to evaluate the processing-time performance of the algorithms.

[6] `http://www.splot-research.org/`

4.4.1 Subject Systems and Experiments

We used a diverse set of subject systems from various sources to evaluate the different applications of presence-condition simplification. For systems that we use in the application scenario "Reporting Analysis Results" (in E1 and E2), we ensured that a variability model is available as context. For the other application scenarios, no variability model is necessary. Table 4.1 gives an overview of the systems and in which experiments they are used. The table also shows the maximum number of configuration options in presence conditions or contexts that occurred in the experiments on these systems. The number differs between application scenarios (e.g., SQLite in E1 and E3) and it can influence the difficulty of simplification tasks (e.g., simp_E and simp_{QC} have timeouts on some tasks of SQLite in E1). The experiments 1–5 are described next.

Classification of variants (E1) The first application scenario is based on an approach that estimates non-functional properties (footprint, response time, etc.) of the variants of a configurable system [SRK+13]. Experiments evaluating the approach typically generated huge datasets containing presence conditions. For E1, we used the following systems from previous work [SKK+12; SRK+13; SvRA13]: Apache, E-mail, h264, and LLVM (prediction of response time per variant), and Linked List, PKJab, SNW, SQLite, and ZipMe (prediction of binary footprint per variant). Presence conditions in this scenario identify system configurations for which the prediction accuracy is low, possibly due to unknown interactions among configuration options. Presence-condition simplification is useful for pinpointing such cases of low accuracy to a smaller number of options such that further investigation is possible. We used presence conditions for seven different levels of prediction accuracy and simplified all of them separately using the corresponding variability model as context.

Reporting defect locations (E2) For our second experiment, we used data from a study in which we evaluated the performance of variability-aware model checking (see Section 6.5, [AvRW+13]). During experiments, we found many defects in the subject systems that occur only under certain presence conditions. We used the E-mail and Elevator systems [AvRW+13; CHSL11; Hal05]; standard benchmarks which we also used in other chapters of this thesis.[7] The presence conditions of defects and the variability model are given as textual Boolean expressions. An example for the defect location scenario is

[7] Comparable systems from the variability-aware model-checking evaluation (Section 6.5) cannot be used because they have no defects (AJStats and ZipMe), because all variants are defect (GPL), or because the variability model is too simple (Mine Pump).

Table 4.1: Subject systems. The column "Exp." denotes in which experiments we used the systems. The columns "Max. Options" and "Max. size" denote the maximum number of options in and the maximum size of presence conditions and contexts in the experiments. These statistic depends on what presence conditions and contexts represent (e.g., tool reports in E1 versus ifdef annotations in E3).

System	Version	Domain	Exp.	Max. Options	Max. size
Apache	2.2	Web Server	E1	10	227
	2.4.6	Web Server	E3	5	5
Berkeley DB	6.0.20	Database	E3	6	21
Busybox	1.22.1	Utilities	E3	28	28
Cherokee	1.2.101	Web Server	E3	3	3
Elevator	1.0	Simulation	E2	5	10
E-mail	1.0	Simulation	E1	10	88
	1.0	Simulation	E2	9	22
Freebsd	9.1.0	Operating System	E3	14	96
Gimp	2.8.6	Image Manipulation	E3	4	9
Gnumeric	1.10.15	Spreadsheet	E3	3	5
Gnuplot	4.6.3	Graph Generation	E3	6	12
H264	0.85.1448	Video Encoding	E1	17	970
Libxml2	2.9.0	XML Toolkit	E3	8	17
Linked list	1.0	Datastructure	E1	19	1512
Linux	2.6.33.3	Operating System	E3	24	44
	2.6.33.3	Operating System	E4	14	468
Linux	3.4	Operating System	E3	24	206
LLVM	2.7	Compiler	E1	12	1067
Openvpn	2.3.2	Networks	E3	11	27
Parrot	5.0.0	Virtual Machines	E3	2	2
Postgresql	9.3.0	Database	E3	5	13
Qemu	1.6.1	Virtual Machines	E3	15	54
Sendmail	8.14.7	Email Routing	E3	6	6
SNW	1.0	Simulation	E1	27	4531
SPLOT models	(generated)	Variability Models	E5	60	9931305
SQLite	3.7.0.4	Database	E1	87	93106
	3.8.0.2	Database	E3	6	21
Subversion	1.8.1	Version Control	E3	5	11
Vim73	73	Editor	E3	10	16
Xfig	3.2.5b	Image Manipulation	E3	6	9
Xterm	296	Terminal	E3	12	23
ZipMe	1.0	Data Compression	E1	9	43

the output of the SPLverifier tool given in Section 4.1.1. We simplified the defect presence conditions and evaluated the performance of the simplification algorithms.

Annotation Simplification (E3) To evaluate the simplification potential for variability annotations, we used several configurable software systems with #if directives and applied simplification to the #if conditions (scenario described in Section 4.1.2). For our experiments (see Section 4.4.3), we used 21 configurable systems, including the Linux kernel.

The context of #if conditions in these projects has two components: the conditions of enclosing #if directives and the condition under which the respective file will be included in the project, as described in Section 4.1. In projects that use KConfig, we used the tool KBuildMiner [BSCW10; BSL+10] to extract the conditions in which source files are used. For the others, we assumed that each file is used in all configurations. We extracted #if conditions in source files with the Predator tool [TDS+14]. Predator also provides the hierarchy of #if conditions, such that we can generate for each #if condition a context consisting of the conjunction of the enclosing #if conditions and the file condition. Given these pairs of #if conditions and contexts, we apply the simp_{BDD}, simp_{E} and simp_{QC} algorithms and measure how often the conditions could be improved to evaluate the potential for presence-condition simplification. We skipped pairs for which #if conditions or contexts are tautologies or contradictions because then simplification is impossible.

AST-annotation simplification (E4) In this experiment, we analyzed the variability-aware ASTs generated by TypeChef [KGR+11]. Each generated AST node has a presence condition. Due to difficulties in parsing C code with #if directives (e.g., undisciplined annotations and macro expansion), the resulting presence conditions are often larger than the conditions written in the source code [KGR+11]. For simplification, we generated a context for each presence condition p by building the conjunction m of all presence conditions on the path from p to the root node of the AST. Then, we applied simplification of p in the context m and evaluated the reduction in the size of the presence conditions. Again, we did not simplify if p or m is a tautology or a contradiction. Even though simplification optimizes only an internal representation here, it can affect processing time. Furthermore, presence conditions are visible to users (1) as part of reports, (2) as debugging info, and (3) if the AST is printed again after some modification (e.g., automatic, variability-aware code refactoring [LJG+15]).

Cross-tree-constraint simplification (E5) To evaluate the scalability potential of presence-condition simplification, we used the variability-model generator from the SPLOT repository [MBC09] for generating test variability models. Each model comprises hierarchy, grouping, and cross-tree constraints given in CNF. This is the same setup as in the final step of the variability-model generation scenario (Section 4.1).

As scaling factor, we used the number of configuration options of the generated models. We generated sets of 10 variability models with 20/30/40/50/60 configuration options (50 models in total). For each model, we simplified the cross-tree constraints using the hierarchy and sibling constraints as context. All constraints are given in CNF, so we apply the FORCE algorithm [AMS03] to optimize the BDD variable ordering. The more compact representation is beneficial for simp_{BDD}, but also for simp_E and simp_{QC}.

4.4.2 Experiment Setup

For our experiments, we used existing implementations of the algorithms.[8] We provide links to the tools on the supplementary website. We also tried to use SCHERZO, a newer tool for two-level logic minimization, however, we were not able to apply it to presence-condition simplification because of technical problems and missing documentation.

All experiments were executed on an Intel Xeon machine (8 cores with 2.93 GHz) with Ubuntu 12.04. Regarding parallelization, we have not observed that more than one core was used in the experiments. In all experiments, simplification has been executed in a JVM with 4 GB of RAM. We set the timeout for the simplification algorithms in all experiments to 60 seconds (the usual response time was much lower). In the experiments E1–E4, we encountered only 12 timeouts, (7 with simp_E and 5 with simp_{QC}). All timeouts occurred while simplifying presence conditions of SQLITE (E1). For variability-annotation simplification (E3), we used scripts to call the external analysis tools (e.g., TYPECHEF). The tool output was aggregated and later simplified.

During the experiments, we measured the processing time of the algorithms and the number of literals (size) in the expressions before and after simplification. For each simplification, we compared the size of the original presence condition p and the simplified presence condition $\mathrm{simp}(p,\ m)$. For these comparisons, we defined the *reduction factor* as

$$\mathrm{size}\big(\mathrm{simp}(p,\ m)\big)/\mathrm{size}(p)$$

[8] simp_{BDD} is available as function net.sf.javabdd.BDD.simplify(BDD) in the JAVABDD library, simp_E is available in the ESPRESSO tool, and simp_{QC} is also implemented in ESPRESSO as a revised version of the original QUINE-MCCLUSKEY algorithm.

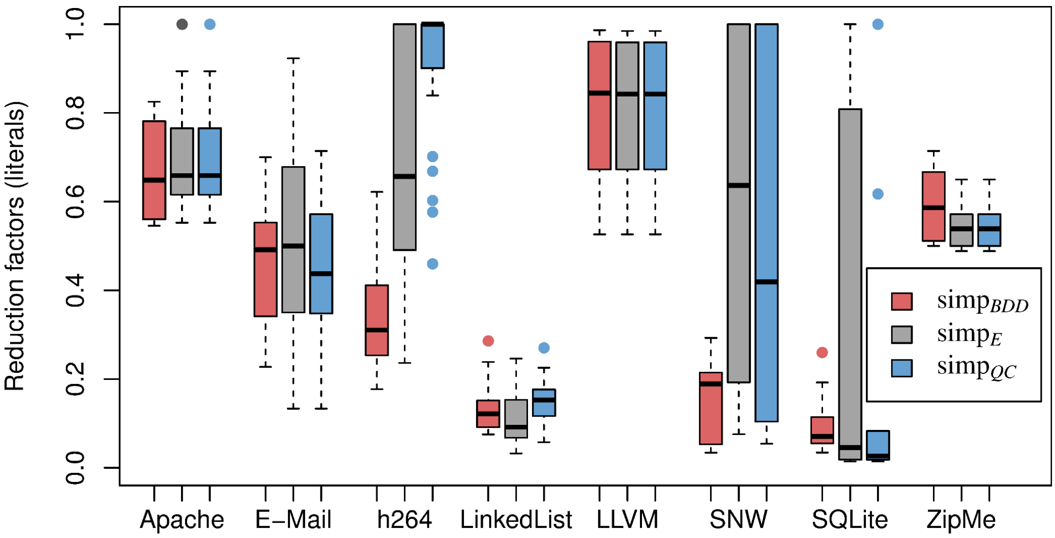


Figure 4.4: Reduction factors for the classification of variants (E1)

In some cases, the size of the supposedly simplified expression was larger than the size of the original expression. This can happen because some of the algorithms rely on heuristics. Such cases are easy to detect and we just used the original expression as result instead of the generated expression. In such cases, we logged that simplification did not improve the expression size (the reduction factor is 1). For example, in the 175 simplification tasks of Experiment 1, simp$_E$ and simp$_{QC}$ failed to reduce the expression size in 29 and 38 cases, respectively. simp$_{BDD}$ improved all 175 expressions. To provide a ground truth, we would need to iterate over all solutions (i.e., apply the Brute-Force algorithm). However, due to the complexity of the problem, Brute-Force does not scale for any of our experiments.

4.4.3 Results

Classification of variants (E1) Figure 4.4 shows the reduction factors we observed for presence conditions for inaccurate performance predictions (cf. Section 4.4.1) per subject system. A lower reduction factor indicates a better simplification result. Each boxplot covers all experiments per algorithm and subject system. Figure 4.4 shows that (1) the number of literals is generally much lower after simplification and that (2) simp$_{BDD}$ generates slightly better results, on average, than simp$_E$ and simp$_{QC}$, as confirmed by paired Mann-Whitney tests (p-values below 0.001 for both tests). These results confirm RQ1: For this application scenario and the considered systems, there is significant simplification potential, and the algorithms are able to simplify the presence

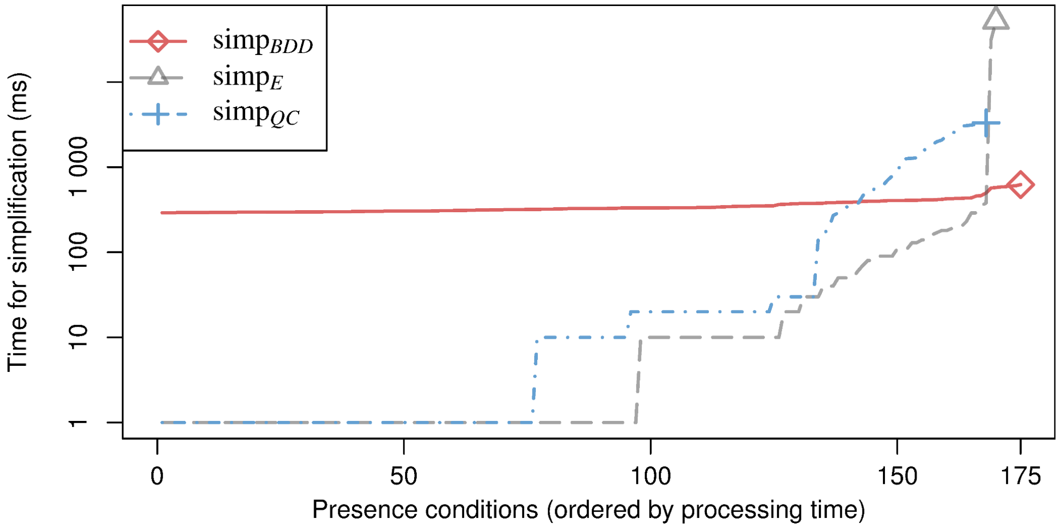


Figure 4.5: Time for simplification in a quantile plot (E1); a point (x, y) in the plot states that the x-th fastest simplification with the respective algorithm took y milliseconds; the right-most x value indicates the number of solved tasks; the y axis has a logarithmic scale

conditions substantially. Furthermore, we observe that for some systems, the reduction factors are very similar across all algorithms (Apache, Linked list, LLVM, and ZipMe). For the other systems, we observe diverse reduction factors. We could not establish a reason for this effect and attribute it to unknown differences between the subject systems.

Figure 4.5 shows the time needed for simplification in a quantile plot. A quantile plot shows the number of presence conditions (x-axis) that can be solved with runtimes below a given value (y-axis) per algorithm. For example, the point $(150, 110)$ in graph simp$_E$ in the plot states that the 150-th fastest simplification with simp$_E$ took 110 ms and there were 149 simplification tasks that took 110 ms or less with simp$_E$. This means that for each algorithm, the "easy" tasks are shown on the left of the plot. If a plot does not extend to the maximum value of the x axis, this means that the remaining tasks were timeouts (cf. simp$_{QC}$ and simp$_E$).

The plot shows how the algorithms scale when tasks get harder to solve using the same simplification tasks as in Figure 4.4; the time for simp$_{QC}$ and simp$_E$ is negligible for easy tasks but increases with harder tasks; simp$_{BDD}$ needs between 350 ms and 500 ms in most cases, however, it can solve more problems than simp$_{QC}$ and simp$_E$. Note that in our setup simp$_{BDD}$ requires startup time for setup of the BDD, which dominates the processing time. A closer investigation showed that most of the startup time of simp$_{BDD}$ is spent

Table 4.2: Reduction factors for defect location reporting (E2)

	simp_{BDD}	simp_{E}	simp_{QC}
ELEVATOR	0.39	0.37	0.37
E-MAIL	0.22	0.14	0.14

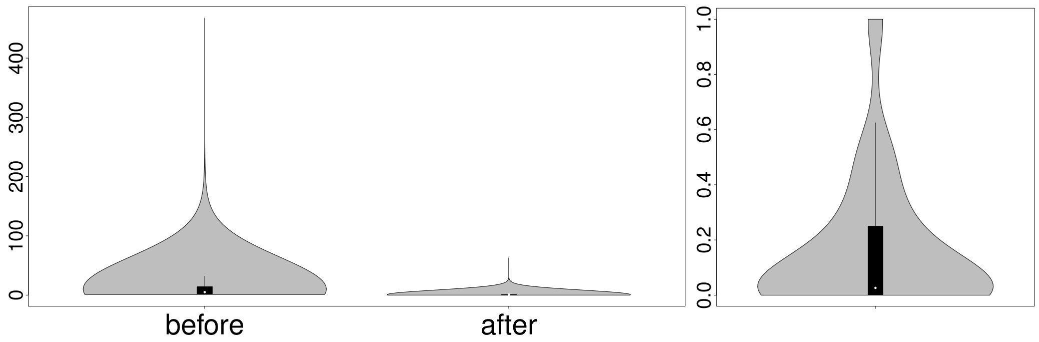
for loading the presence condition and context as BDDs (at least 320 ms). The actual simplification took less than 2 ms in each case. The startup time can be influenced by reducing the initial number of nodes in the BDD. We evaluate and discuss this further in the Appendix (page 226).

Reporting defect locations (E2) In E2, we evaluated simplification of presence conditions reported during verification of E-MAIL and ELEVATOR [AvRW+13] (cf. Section 6.5). Note that E-MAIL and ELEVATOR are the same systems as in E1, but the considered presence conditions represent very different facts: In E1, the presence conditions represent the prediction accuracy of the non-functional property prediction approach [SKK+12; SvRA13]. In E2, the considered presence conditions point to configurations in which specifications of the configurable systems are violated, as identified by SPLVERIFIER (Chapter 6).

Table 4.2 shows the average reduction factor per case study and algorithm. All three algorithms achieved significant improvements of the simplified expressions in terms of the reduction factors, providing further evidence for RQ1. Overall, the reduction factors are very similar for all algorithms; the results for the E-MAIL system are better than for the ELEVATOR. This is probably due to the internal structure of the case studies and the nature of the defects. The maximum time measured for simplification was 320 ms.

Variability-annotation simplification (E3) In E3, we evaluated the potential for simplification of ifdef conditions in source code. Overall, we found only few situations where our approach could improve the presence conditions. For detailed results of the experiment, we refer to the supplementary website. With all three algorithms and in all systems except gnuplot (3.7%) and libxml2 (3.2%), we could improve only less than 2% of the parsable, non-trivial presence conditions.[9] If we could not parse the conditions, this was usually due to non-Boolean configuration options. Most situations for which we could improve the conditions are rather simple, similar to the example shown in Figure 4.1 on page 61.

[9] Of 301 520 presence conditions, 277 297 could be parsed and of these 128 869 were non-trivial.



(a) Presence-condition sizes before/after simplification (b) Reduction factor

Figure 4.6: Experiment results for the TypeChef AST simplification on Linux (E4)

The Experiment 3 shows that our approach can be applied to variability annotations, but there is only little potential for simplification in the considered systems. The ifdef conditions in the analyzed systems do not contain much redundancy, which indicates a good code quality. So the expectation stated in RQ1 is not confirmed by E3.

AST-annotation simplification (E4) To evaluate the simplification potential in ASTs as generated by TypeChef, we modified TypeChef such that it applies simplification to all presence conditions generated as AST annotations. In particular, we analyzed the AST presence conditions generated for Linux 2.6.33.3 (the actual subject of E4 is TypeChef, not Linux, so one version is sufficient). Figure 4.6 shows the results by means of violin plots. A violin plot is a boxplot with a rotated kernel density plot on each side. In our case, the width of the density plots show the relative number of presence conditions of a certain size (Figure 4.6a) and the relative number of simplification tasks with a certain reduction factor (Figure 4.6b).

In total, we found 25 580 099 non-trivial presence conditions in the AST of Linux's code base. Figure 4.6a shows that most of these have less than 100 literals. However, there is a substantial number of presence conditions that have an extremely large number of literals. After simplification (shown data generated with simp_{BDD}), the conditions have less than 50 literals. Figure 4.6b shows the reduction factors observed with simp_{BDD} (the results are similar for simp_{E} and simp_{QC}). For most presence conditions, we achieved extreme improvements, which leads us to two conclusions confirming RQ1: (1) TypeChef introduces many redundancies during parsing, because we did

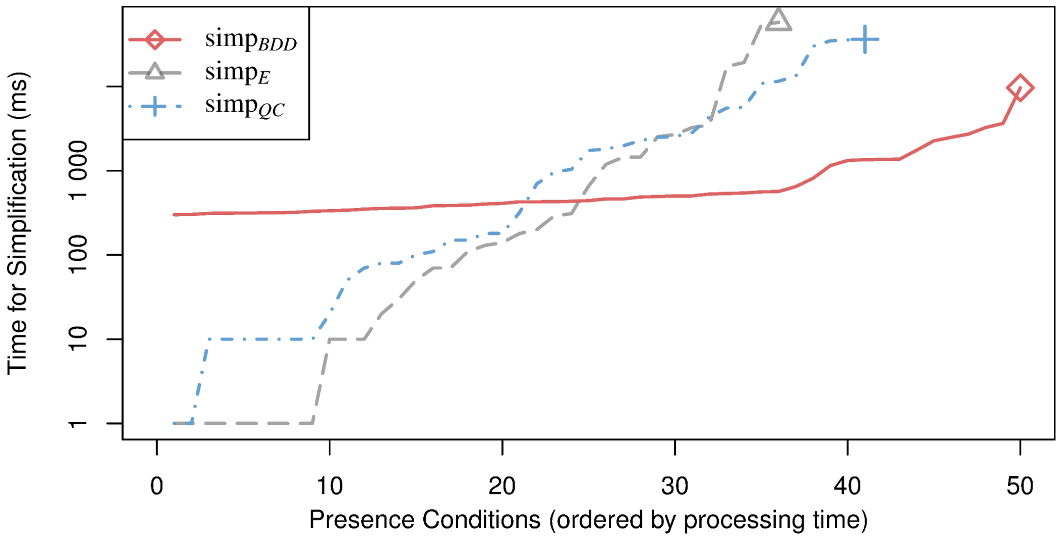


Figure 4.7: Scalability of simplification algorithms shown in a quantile plot (E5), similar to the quantile plot in Figure 4.5

not observe similar sizes for Linux in E3, and (2) simplification can remove those redundancies from the AST.

Cross-tree-constraint simplification (E5) To evaluate the scalability of simp$_{BDD}$, simp$_E$, and simp$_{QC}$ (RQ2), we ran experiments with synthetic feature models of different sizes (see scenario "Cross-Tree-Constraint Simplification"). Figure 4.7 shows the time needed for simplification of the cross-tree constraints in a quantile plot. It supports the general result of E1: simp$_{BDD}$ has a higher processing time for simple tasks (again, consider BDD setup time), but it can solve more tasks and is faster than simp$_E$ and simp$_{QC}$ when it comes to harder tasks. simp$_{QC}$ performs better than simp$_E$, which was not to be expected, because it is an earlier algorithm solving the same problem. We were not able to run a complete evaluation for larger problem sizes, because the computation of the input files for simp$_E$ and simp$_{QC}$ is very expensive for harder problems. When evaluating presence-condition simplification on 10 problem instances with 150 options, simp$_{BDD}$ still needs only 9 ms, on average (when the BDD is already loaded in memory). In summary, the answer to RQ2 is that simp$_{BDD}$ has a high minimum processing time (500 ms) but scales better than the other algorithms.

Discussion As a summary of our evaluation, we answer the research questions based on the results of our experiments. RQ1 asks whether presence-conditions is able to reduce the size of presence conditions significantly. Our experiments

(E1, E2, and E4) showed that there is significant simplification potential in presence conditions in practical application scenarios. Furthermore, the experiments showed that the three algorithms achieve significant reduction in the size of the presence conditions. The result of E4 is an exception to this conclusion. We found only little simplification potential in the ifdef annotations in source code, which indicates a good code quality.

Our second research question, RQ2, asks how the processing time of simp_{BDD}, simp_{E} and simp_{QC} scales to complex simplification tasks. E5 showed that simp_{BDD} scales better than simp_{E} and simp_{QC}. For loading and simplification of presence conditions with up to 60 configuration options, simp_{BDD} needs less than 10 seconds. Our experiments (E1 and E5) also showed that simp_{BDD} has a high minimum processing time of about 500 ms. A closer investigation (Appendix, page 226) showed that most of the startup time is spent for loading the presence condition and context as BDDs (at least 320 ms). Once the presence condition and the context are loaded as BDDs in memory, the actual simplification takes less than 2 ms in each simplification task of E5.

4.4.4 Threats to Validity

A threat to construct validity is that there is no generally accepted measure for the simplicity of Boolean expressions. This is a problem that is not specific to our work; in general, it is difficult to define such a measure. We have tried a number of different measures and observed similar results, so we expect our observations to hold with other sensible measures. In addition, we tried to use a ground truth for the minimal size measure in our experiments. However, deriving the ground truth, even on small problems, requires an infeasible amount of computation. Hence, we focus on the comparison between the algorithms.

Another threat to internal validity is that we used existing tools to compare the algorithms, so we rely on that the tools actually implement the algorithms correctly. Still, we verified that each simplification result satisfies the invariant of Equation 4.1.

A threat to the external validity—as always—is the problem of selection of a representative set of subject systems and application scenarios. To mitigate this threat, we selected a diverse set of application scenarios and subject systems. Our assumption that simplification can significantly improve presence-condition size holds in all these scenarios, except for variability-annotation simplification.

A threat to the external validity is that we can only handle Boolean options. However, it has been shown that the majority of presence conditions in large configurable systems (e.g., kernels of Linux and FreeBSD) can be expressed using only propositional expressions [BSL+10], and that the large majority of options in configurable systems software is Boolean [BSL+13]. Even if parts of the context cannot be expressed with Boolean options, a partial context can be used for simplification.

Even though the majority of our subject systems and application scenarios are real, our approach is certainly limited with respect to very large presence conditions. We tried larger problem instances in Experiment 5. However, for one variability model with 100 options, we had to generate an input file for simp_E with 637 GB (condition encoded as DNF). For very large presence conditions, the simplified presence condition will probably still be quite large, so it is questionable whether simplification is even useful in such cases. We argue that even if the result expression is still large, every bit of size reduction helps if the result is used as input to an analysis tool. If a user interprets the (still large) result, the user might use tools such as dependency graphs. Such graphs are simpler once redundant information is removed by simplification.

4.5 Related Work

Simplification of presence conditions in the context of highly configurable systems has not been investigated before. However, work on variability-model analysis and the extraction of presence conditions is related.

Variability-model analysis aims at analyzing properties of variability models (e.g., consistency) or of sets of models (e.g., relationships between models)—to assure correctness and to support evolution and configuration of systems. A common operation is to calculate differences between two variability models. This problem has been explored before [Ach11; CW07; TBK09] and it is closely related to ours: Simplification of a presence condition p with a context m is similar to removing m's information from p and returning the remainder. In general, computing differences (diffs) between two entities a and b involves two tasks: stripping the information of a from b and vice versa. Thus, a diff is a pair of sub-comparisons. The main difference to our simplification problem is that a diff has to be exact. That is, *all* the information of b is removed from a and *only* the remainder is presented to the user. Therefore, applying model-differencing methods on presence conditions has fewer means to influence the size of the presence condition. Our problem formulation gives us more leverage: Information that is contained in p and m can either remain in $\mathrm{simp}(p,\ m)$ or be removed. We use this leverage to make $\mathrm{simp}(p,\ m)$ smaller and more readable.

Off-the-shelf reasoners, such as BDD libraries or SAT solvers, are used for reasoning about Boolean expressions. Scalability experiments [Men09; MWC09; MWCC08] show that SAT solvers are more scalable than BDDs for most analyses on feature models. However, BDDs are efficient for analyses that rely on enumerating configurations; they are known to scale up to models with 2000 features [MWC09]. In our experiments, simp$_{BDD}$ exhibited a better scalability than simp$_E$ and simp$_{QC}$, but for smaller models and presence conditions. We are not aware of any SAT-based algorithm applicable to our problem (besides simp$_{BF}$ and similar brute-force approaches). However, investigating the feasibility of using a SAT solver would be valuable future work.

Various researchers have extracted and analyzed presence conditions in the context of highly configurable systems. Presence conditions have been extracted using static analysis from build systems [BSCW10; NH13] and using dynamic analysis by compiling individual system variants [DTSL12]. All these pieces of work illustrate the importance of complex presence conditions to realize the mapping between the variability model and implementation. In fact, presence conditions in source code and other artifacts are means to maintain the variability model by establishing a balance between constraints residing in the model and in other artifacts. Our experiments (in particular E3) have shown that the size of presence conditions in real systems is moderate, suggesting that such systems are relatively well maintained.

The Explain algorithm is a BDD-based algorithm that is used in interactive configuration of configurable-systems [Sub05]. Given a variability model and a (partial) configuration p of the system, it allows users to query why a certain configuration option x is implied or hidden from the selection (e.g., because it is implied by previous choices). Explain generates a minimal set of option choices in p that imply a fixed value for x (or that x is still variable). In difference to presence-condition simplification, Explain reports only one set of variable assignments (it boils down to a shortest-path problem). So, even though Explain and simp$_{BDD}$ are used in the context of configurable-systems and both use BDDs, they are quite different algorithms used for different problems.

We found two publications in which approaches similar to presence-condition simplification are discussed (although not in the configurable-systems domain): The first publication (from 1979) introduces "simplification by context" with a problem description that is equivalent to presence-condition simplification [vLS79]. As solution to the problem, they propose syntactic substitution rules that are applied on the condition. For example, a condition $A \wedge B$,[10] would be simplified to A iff B is implied by the context. In our experiments, such situations occurred only in the simplest presence conditions. More often simplification potential is caused by dependencies or redundancies between the operands, which would be ignored by such simple substitution rules. The second publication describes *Simplify*(g, C) similar to presence-condition sim-

[10] A and B are placeholders for more complex expressions that may share variables.

plification, and states that it "is implemented using standard propositional operators available in Binary Decision Diagram libraries" [DLvL15]. Despite a personal communication with the authors, we were not able to determine whether they used the RESTRICT algorithm.

In our search for simplification algorithms, we have also looked at several research areas related to BDDs and two-level-logic minimization. In particular decomposable negation normal forms [DM02; HD05], semantic tableaux [DAg99], and minimization of propositional formulae [Lib05] appeared promising at first sight, but in the end, we have not found algorithms applicable to our problem (except for simp_{BDD}, simp_{E}, and simp_{QC}).

CHAPTER 5

Variability Encoding

This chapter shares material with the publication "Variability Encoding: From Compile-Time to Load-Time Variability" in *JLAMP'2015* [vRTS+16].

Before we evaluate different strategies for the analysis of configurable system, we need to develop means to implement the strategies. For the variant-based strategy, almost no effort is required, because the analysis subjects (variants) are normal, non-configurable systems. Such systems can usually be analyzed with off-the-shelf tools. For the sample-based strategy, sample sets have to be generated. This topic has already been covered extensively by other researchers (e.g., [NL11; OMR10]). The remaining task, which we address here, is to implement the family-based strategy.

In this chapter, we introduce *variability encoding*, an automatic process that generates a *variant simulator* from a configurable system. The process encodes the compile-time variability (static) of the configurable system as load-time variability (dynamic) in the variant simulator. One important property of variability encoding is that the resulting simulator can simulate the behavior of any variant (on an abstract level), called *behavior preservation*. Behavior preservation allows us to use simulators in family-based analyses of configurable systems and, based on the analysis results, make statements about all variants (Chapter 6).

We first describe variability encoding informally and describe a family-based analysis as example use case (Section 5.1). Then, we define variability encoding based on the simple, formal programming language FEATHERWEIGHT JAVA (FJ) [Pie02] (Section 5.2). Based on the formal definitions, we prove that variability encoding preserves the behavior of all variants (Section 5.3). As our formal proof is necessarily limited to FJ, we also discuss how elements of mainstream languages, such as Java and C, interfere with variability encoding. This discussion is based on example code snippets (Section 5.4) and on our experience implementing variability encoding for Java and C (Section 5.5). Finally, we give an overview of related work (Section 5.6).

5.1 Description and Use Case

Variability encoding is a process that encodes compile-time variability of a configurable system in load-time variability. Compile-time configuration options are encoded with global program variables, and static configuration choices (e.g., ifdefs) are encoded with conditional statements in the target language (if statements). Classen et al. [CHSL11] and Post and Sinz [PS08] have used approaches that are similar to variability encoding. However, their encoding was done manually, whereas we developed a tool for *automatic* variability encoding that can be applied to large code bases.

A variant simulator that has been generated with variability encoding can be used in different variability-aware analyses. For example, we used simulators for verification (see Chapter 6), test-case generation [BLB$^+$15], and prediction of non-functional properties of variants [SvRA13].

To illustrate the use of variability encoding, we present a practical application scenario in Section 5.1.1. Then, we explain why a formal proof of behavior preservation in variability encoding is critical for our application scenarios (Section 5.1.2).

5.1.1 A Practical Application Scenario

In this section, we describe an application of variability encoding using sample code from the Linux kernel. In particular, we use code from the Variability Bug Database [ABW14], a collection of real bugs that were found in the Linux kernel and that occur only in certain configurations. For each bug, the database contains an executable program slice that represents the core of the bug in a comprehensive, self-contained way. The programs in the database have been simplified to provide a minimal, comprehensive, and self-contained scenario for the corresponding bug.

For the purpose of this example, we used model checking to verify programs from the bug database. We show that the model checker automatically identified the bug in exactly those configurations that are documented in the database.

We selected bugs from the Variability Bug Database, based on three criteria:

- The corresponding program must be compilable (e.g., no variability-dependent type errors). Otherwise, the program has no defined behavior and model checking is not possible.
- Our model checker must be able to find the bugs. For example, our model checker sometimes cannot track function pointers when they are passed as parameters between functions. We excluded bugs that use functionality which is not covered by our model checker.
- We focus on code that includes at least two different ifdef options (reflected by a bug presence condition with at least two options). Most challenges in variability encoding are caused by interactions between configuration options, which can only occur if multiple options are used.

We inspected all 43 Linux kernel bugs in the database and found three bugs matching our criteria. All other bugs in the database have either presence conditions with only one configuration option, require pointer tracking, or have type errors. We provide a list with all bugs and our analysis verdict in the appendix (page 227).

The first selected bug (Bug1[1]) causes a call the Linux BUG() macro, if configuration option VLAN_8021Q is disabled and option IPV6 is enabled. Therefore, the presence condition for the bug is IPV6 ∧ ¬VLAN_8021Q. We include the code of Bug1 for illustration (Figure 5.1a).

The second bug (Bug2[2]) causes an uninitialized return value. The bug's presence condition is NETPOLL ∧ ¬IPV6. The third bug (Bug3[3]) causes a call of an uninitialized function pointer. Its presence condition is ARCH_OPAM3 ∧ ¬PM. We don't show the code of the second and third selected bug because they are quite similar to Bug1 in length and complexity.

We used our tool Hercules (Section 5.5.2) to transform the ifdef variability in the bug programs to load-time variability (shown in Figure 5.1b). Configuration options are represented with global variables (*feature variables*). The feature variables are initialized with return values of the special function __VERIFIER_nondet_int(). This function is assumed to return an arbitrary value of type int [Bey15]. In our case, this assumption implies that all four combinations of id2i_config_ipv6==0 or id2i_config_ipv6!=0 and id2i_config_vlan_8021q==0 or id2i_config_vlan_8021q!=0 are reachable in the program. In variability-encoded configurable systems, this corresponds to covering all valid configurations.

The syntactic structure of the code is largely preserved by variability encod-

[1] http://vbdb.itu.dk/#bug/linux/d549f55

[2] http://vbdb.itu.dk/#bug/linux/e39363a

[3] http://vbdb.itu.dk/#bug/linux/63878ac

```
#include <assert.h>
#include <stdbool.h>
#include <stdlib.h>
#ifdef CONFIG_VLAN_8021Q
void* vlan_dev_real_dev() {
  return NULL;
}
#else
void* vlan_dev_real_dev() {
    assert(false); // (3) ERROR
    return NULL;
}
#endif
#if defined(CONFIG_IPV6) ||
    defined(CONFIG_VLAN_8021Q)
static int ocrdma_inet6addr_event() {
    vlan_dev_real_dev(); // (2)
    return 0;
}
#endif /* IPV6 and VLAN */
int main(int argc, char** argv) {
#if defined(CONFIG_IPV6) ||
    defined(CONFIG_VLAN_8021Q)
  ocrdma_inet6addr_event(); // (1)
#endif
  return 0;
}
```

(a) Original source code of Bug1

```
...
int id2i_config_ipv6;
int id2i_config_vlan_8021q;
extern int __VERIFIER_nondet_int();
void id2i_init() {
  id2i_config_vlan_8021q =
    __VERIFIER_nondet_int();
  id2i_config_ipv6 =
    __VERIFIER_nondet_int();
}
void * _5_vlan_dev_real_dev() {
  return ((void *) 0);
}
void * _6_vlan_dev_real_dev() {
  (0 ? ((void ) 0) : __assert_fail(...));
  return ((void *) 0);
}
static int ocrdma_inet6addr_event() {
  if (((id2i_config_vlan_8021q ) )) {
    _5_vlan_dev_real_dev();
  }
  if ((((! id2i_config_vlan_8021q) ) )) {
    _6_vlan_dev_real_dev();
  }
  return 0;
}
int main(int argc , char **argv ) {
  id2i_init();
  if ((((! id2i_config_vlan_8021q) &&
      id2i_config_ipv6) ||
      id2i_config_vlan_8021q)) {
    ocrdma_inet6addr_event();
  }
  return 0;
}
```

(b) Source code of a variant simulator for Bug1

Figure 5.1: Original and variability encoded source code of Bug1 from the Variability Bug Database. The program path causing the bug is annotated with comments in the original code. The simulator was generated with our tool Hercules (Section 5.5.2). Vertical bars show which functions and statements in the original code (left) correspond to which functions and statements in the simulator (right).

ing. The largest structural difference is caused by function vlan_dev_real_dev. In the original code (Figure 5.1a), it is defined in two variants. One variant of the function contains the assertion that causes the program to fail. During variability encoding, Hercules duplicates this function and issues new names for both duplicates (_5_vlan_... and _6_vlan_...). The prefixes _5_ and _6_ encode the presence conditions VLAN_8021Q and ¬VLAN_8021Q for the function variants. Other changes compared to the original code are caused by macro expansions (NULL, assert). These are done automatically by the parser on which Hercules is based.

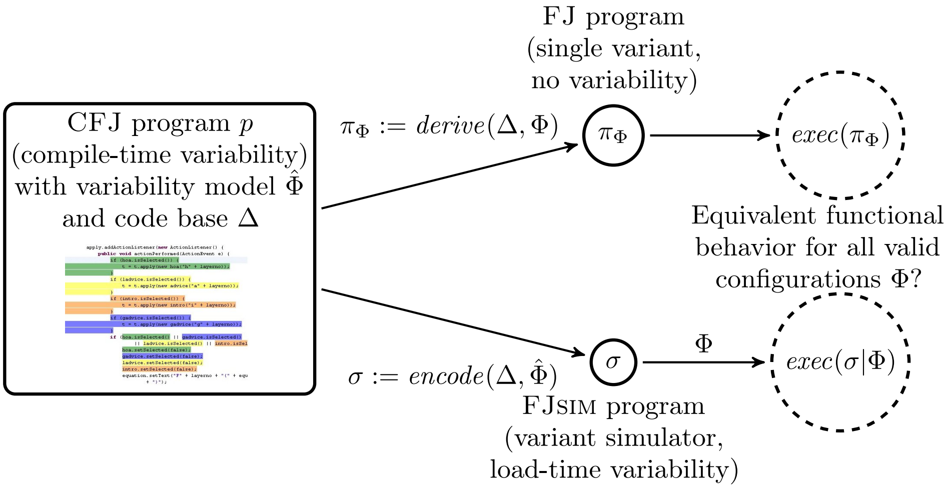
For verification of the generated programs, we used the model checker CPAchecker with extensions that allow special handling of the feature variables. These extensions are described in more detail in Chapter 6. Essentially, the extensions enable CPAchecker to explore the program execution in all valid configurations and issue a summarized result.

After verification of Bug1, CPAchecker reported ErrorSummary: !id2i_config_vlan_8021q@0 & id2i_config_ipv6@0, which exactly corresponds to the bug presence condition IPV6 ∧ ¬VLAN_8021Q reported in the bug database. Similarly, for Bug2 and Bug3, CPAchecker reported error summaries that exactly correspond to the presence conditions stated in the bug database.

Even though the programs are simple, this experiment indicates that the combination of variability encoding and model checking can be effectively used to identify defects in real programs and report precise defect presence conditions. Furthermore, it shows in a limited setting that variability encoding, as implemented in Hercules, can correctly encode the ifdef variability of real-world programs (see Section 5.5.2 for a larger evaluation).

5.1.2 The Need for a Formal Correctness Proof

Variability encoding has been used in several research projects and enabled considerable analysis speedups compared to variant-based analysis [ASW$^+$11; AvRW$^+$13; SvRA13]. In these projects, and in the previous example, the subject systems used simple language features, so we could safely assume that the generated code preserved the behavior of the original code and that the subsequent analyses were valid. However, to this end, it is not obvious how to implement variability encoding correctly in the presence of other language features, such as inheritance, overriding, or switch statements. Variability encoding is a complex program transformation and in practice even relatively simple automatic refactoring engines cannot preserve behavior in the presence of variability [LJG$^+$15]. To improve our understanding of variability encoding, we decided to approach behavior preservation in variability encoding formally. We define variability encoding based on a small, formal language that already



```
apply.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (hoa.isSelected()) {
            t = t.apply(new hoa("h" + layerno));
        }
        if (ladvice.isSelected()) {
            t = t.apply(new advice("a" + layerno));
        }
        if (intro.isSelected()) {
            t = t.apply(new intro("i" + layerno));
        }
        if (gadvice.isSelected()) {
            t = t.apply(new gadvice("g" + layerno));
        }
        if (hoa.isSelected() || gadvice.isSelected()
                || ladvice.isSelected() || intro.isSel
            hoa.setSelected(false);
            gadvice.setSelected(false);
            ladvice.setSelected(false);
            intro.setSelected(false);
            equation.setText("F" + layerno + "(" + equ
                    + ")");
```

Figure 5.2: Variability encoding and behavior preservation

contains many language features that we deem problematic (e.g., method overriding), and we prove behavior preservation in this setting. Then, we discuss how other language features can be dealt with (e.g., switch-case statements).

5.2 A Formal Model of Variability Encoding

In this section, we develop a formal model of variability encoding. We illustrate the process of variability encoding and the property of behavior preservation between variants and the corresponding variant simulator in Figure 5.2. We use $\hat{\Phi}$ to denote a variability model, Φ to denote a configuration and ϕ to denote a presence condition. Given a configurable program p with variability model $\hat{\Phi}$ and code base Δ, we use weak bisimulation to prove that the execution of any variant π_Φ (generated with configuration Φ) and the variant simulator σ, limited to Φ[4], yields the same observable behavior (Section 5.3). This proof shows the soundness and completeness of variability encoding with respect to the configuration space (i.e., all the behavior of all variants is subsumed in the simulator). In particular, we derive the variant π_Φ with $derive(\Delta, \Phi)$ and encode the variant simulator σ with $encode(\Delta, \hat{\Phi})$ (both functions are defined in Section 5.2.4). With $exec(\pi_\Phi)$ and $exec(\sigma|\Phi)$ we denote the execution of variant and variant simulator assuming configuration Φ, respectively. In terms of the PLA model (Chapter 3), variability encoding is an implementation of the variability combinator (ν) and a variant simulator σ is an element of the set $\mathbb{S}$.

[4] We set the simulator's feature variables to the option values in configuration Φ.

Our model of variability encoding supports all language constructs of FJ, including classes, methods, fields, inheritance, dynamic type casts, and method overriding. Note that FJ does not support method overloading. If a method m has a certain signature, all other methods named m in the inheritance hierarchy must have exactly the same signature [Pie02, p. 257, *Valid method overriding*]. In Section 5.4, we discuss how allowing overloading as in JAVA would affect variability encoding. Essentially, variability in our model is implemented by optional methods (either included or excluded from the code) and variable method bodies (alternatives for the default method body).

In Section 5.2.1, we give an overview of FJ, which is the base language for our formal model. Then, we describe COLORED FEATHERWEIGHT JAVA (CFJ) [KATS12], which we use to represent compile-time configurable programs (Section 5.2.2). Then, we introduce the language FJSIM, which we use to represent variant simulators (Section 5.2.3). Finally, we define how variants and variant simulators are derived from CFJ programs (Section 5.2.4).

5.2.1 Featherweight Java (FJ)

FJ is a functional subset of JAVA with a precise syntax definition, a sound type system, and evaluation rules [Pie02]. We focus on the definitions relevant to our model. In the following description, we use a short notation for lists: $\overline{\mathsf{a}}$ denotes a list of syntax elements. Sequences of field declarations, parameter names, and method declarations are assumed to contain no duplicates. For example, the parameter list of a method definition is denoted as $(\overline{\mathsf{C}\ \mathsf{x}})$, which expands to $(\mathsf{C}_1\,\mathsf{x}_1, \ldots, \mathsf{C}_n\,\mathsf{x}_n)$.

Figure 5.3 shows the syntax rules of FJ including rules for class and method declarations and terms, such as field accesses and method invocations. The syntax does not include an assignment operator; fields are only assigned once in the constructor. It also does not contain conditional (e.g., if) or loop constructs (e.g., while). A method body consists of only a single return statement with a term that may contain other nested terms. FJ supports the keyword super only as first statement in constructor bodies. Despite its limitations, FJ is Turing complete, as one can encode the Lambda calculus in FJ [Pie02].

An FJ program consists of a class table *CT* and a *start term* init. The evaluation of a program begins with the start term, which is similar to the main method in JAVA. We assume that there is a special variable this, but that this is never used as the name of a formal parameter of a method declaration. It is considered to be implicitly bound in every method declaration. During evaluation, this is substituted with a representation of the object that would be referenced by this in JAVA (Figure 5.4, E-INVKNEW).

Figure 5.4 gives the small-step operational semantics of FJ. The evaluation

Syntax

$$
\begin{array}{lcll}
\mathsf{P} & ::= & (\overline{\mathsf{L}}, \mathsf{t}) & \textit{program} \\
\mathsf{L} & ::= & \mathsf{class\ C\ extends\ C}\ \{\ \overline{\mathsf{C\ f;}}\ \mathsf{K}\ \overline{\mathsf{M}}\ \} & \textit{class declaration} \\
\mathsf{K} & ::= & \mathsf{C}\,(\overline{\mathsf{C\ x}})\ \{\ \mathsf{super}(\overline{\mathsf{x}});\ \overline{\mathsf{this.f=f;}}\ \} & \textit{constructor declaration} \\
\mathsf{M} & ::= & \mathsf{C\ m}\,(\overline{\mathsf{C\ x}})\ \{\ \mathsf{return\ t;}\ \} & \textit{method declaration} \\
\mathsf{v} & ::= & & \textit{values:} \\
 & & \mathsf{new\ C}(\overline{\mathsf{v}}) & \textit{object creation} \\
\mathsf{t} & ::= & & \textit{terms:} \\
 & & \mathsf{x} & \textit{variable} \\
 & & \mathsf{t.f} & \textit{field access} \\
 & & \mathsf{t.m}(\overline{\mathsf{t}}) & \textit{method invocation} \\
 & & \mathsf{new\ C}(\overline{\mathsf{t}}) & \textit{object creation} \\
 & & (\mathsf{C})\mathsf{t} & \textit{cast}
\end{array}
$$

Figure 5.3: The syntax of FJ [Pie02]

$$
\frac{\textit{fields}(\mathsf{C}) = \overline{\mathsf{C\ f}}}{(\mathsf{new\ C}(\overline{\mathsf{v}})).\mathsf{f}_i \to \mathsf{v}_i}\ \text{(E-ProjNew)}
\qquad
\frac{\textit{mbody}(\mathsf{m}, \mathsf{C}) = (\overline{\mathsf{x}}, \mathsf{t}_0)}{(\mathsf{new\ C}(\overline{\mathsf{v}})).\mathsf{m}(\overline{\mathsf{u}}) \to [\overline{\mathsf{x} \mapsto \mathsf{u}}, \mathsf{this} \mapsto \mathsf{new\ C}(\overline{\mathsf{v}})]\, \mathsf{t}_0}\ \text{(E-InvkNew)}
$$

$$
\frac{\mathsf{C} <: \mathsf{D}}{(\mathsf{D})(\mathsf{new\ C}(\overline{\mathsf{v}})) \to \mathsf{new\ C}(\overline{\mathsf{v}})}\ \text{(E-CastNew)}
\qquad
\textit{congruence rules are omitted}
$$

Figure 5.4: Evaluation rules of FJ [Pie02]

rules are designed to be conform as much as possible with Java. For example, the rule E-InvkNew defines method-call resolution. During evaluation, a start term $(\mathsf{new\ C}(\overline{\mathsf{v}})).\mathsf{m}(\overline{\mathsf{u}})$ is evaluated using E-InvkNew. This rule applies alpha-equivalent substitution ($[\mathsf{x} \mapsto \mathsf{y}]\, \mathsf{t}_0$) replacing occurrences of x in t_0 with y. The term $(\mathsf{new\ C}(\overline{\mathsf{v}})).\mathsf{m}(\overline{\mathsf{u}})$ is evaluated to the body t_0 of method m with substitutions of the keyword this and of uses of formal parameters. The result of this evaluation step is a new term. The term is a fully evaluated value ($\mathsf{new\ C}(\overline{\mathsf{v}})$), or further evaluation steps can be applied to the term. A class inheritance relation, in which class C extends class D, induces a subtype relation $\mathsf{C} <: \mathsf{D}$.

5.2.2 Colored Featherweight Java (CFJ)

As the source language for variability encoding, we use CFJ [KATS12], which extends FJ with support for compile-time variability using presence conditions attached to program elements. A CFJ program has an initial term and a set of classes that are, in fact, nested term structures with presence conditions on some terms. These structures are isomorphic to abstract syntax trees with presence conditions on nodes, which is a canonical representation of compile-time variability [KAK09; KGR+11]. Given a configuration Φ, we can derive a variant (an FJ program) from a CFJ program. This derivation is, basically, done by checking for all program elements[5] e whether their presence conditions ϕ_{e} are satisfied ($sat(\phi_{\mathsf{e}} \wedge \Phi)$) and removing all elements with unsatisfied conditions. The function *derive* is formally defined in Section 5.2.4.

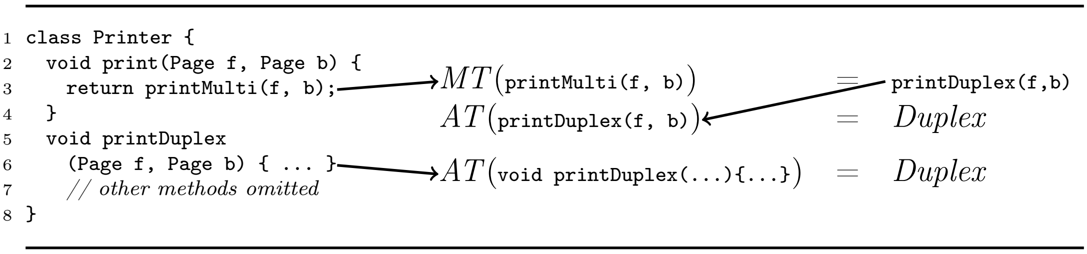
variability restrictions

We restrict variability in CFJ such that only complete methods and method bodies can be variable. Also, a method name can be used only once per class. This restriction significantly improves the readability of the definitions and the behavior-preservation proof. The described restrictions are no severe limitations to the applicability of our approach as one can always transform more fine-grained variability, such as optional parameters, to our restricted version of CFJ [KGR+11; LKA11]. For example, alternative method implementations can be expressed as alternative expressions in the `return` statement (e.g., `return (!feature_A ? "default": "alternative");`). To prove behavior correctness for a language with more fine-grained variability, one would need to prove either that the translation from this language to our model preserves behavior or that variability encoding works correctly on the language with fine-grained variability. Both is well beyond the scope of this thesis, however, we discuss informally how to handle fine-grained variability in some program elements (e.g., optional program variables) in Section 5.4.

A CFJ program consists of a representation of the code base ($CT, AT, MT, \mathsf{init}$) and a variability model $\hat{\Phi}$. The code base consists of a class table CT, an annotation table AT, a metaexpression table MT, and a start term init. The class table and start term are structures as in FJ (Section 5.2.1).

The variability information of a CFJ program is defined in terms of an annotation table AT and a metaexpression table MT. The annotation table AT contains a presence condition for each program element defining in which configurations (i.e., system variants) the program element is present. The metaexpression table MT contains for each variable program element a either an alternative program element a_1 or the empty program element $\bullet$ (denoting that there is no alternative). Correspondingly, AT contains a presence condition for each alternative program element. Alternatives such as $\mathsf{a}_1 = MT(\mathsf{a})$ can again have alternatives $\mathsf{a}_2 = MT(\mathsf{a}_1)$. During variant derivation, for each program element a, MT is used to recursively search an alternative a_i for

[5] Program elements are, for example, class, field, and method declarations, method calls, and field references.



```
1 class Printer {
2   void print(Page f, Page b) {
3     return printMulti(f, b);
4   }
5   void printDuplex
6     (Page f, Page b) { ... }
7     // other methods omitted
8 }
```

$$
\begin{aligned}
MT(\texttt{printMulti(f, b)}) &= \texttt{printDuplex(f,b)}\\
AT(\texttt{printDuplex(f, b)}) &= \mathit{Duplex}\\
AT(\texttt{void printDuplex(...)\{...\}}) &= \mathit{Duplex}
\end{aligned}
$$

Figure 5.5: CFJ program for method print(f,b) of the printer driver; AT entries with condition *true* are omitted; arrows illustrate the relation between the code and corresponding entries in AT and MT.

which $AT(\mathsf{a}_i)$ is satisfied by the configuration of the derived variant. If such an alternative is found, it substitutes a. We formally define how variants are generated from CFJ programs in Section 5.2.4. We assume that references in AT and MT to program elements are always unambiguous. Figure 5.5 illustrates how the printer driver can be represented in CFJ. References are illustrated with arrows. Expression printMulti(f,b) in Line 3 is replaced by printDuplex(f, b) iff feature *Duplex* is selected. In this case, the declaration of method printDuplex is included during program generation, too (its AT entry is *Duplex*). For more information on CFJ, we refer to Kästner et al. [KATS12].

5.2.3 Featherweight Simulation Java (FJsim)

We introduce FJsim as the target language of our model of variability encoding (CFJ is the source language). FJsim does not support compile-time variability, but load-time variability. FJsim also supports access of superclass methods with the super keyword as in Java. The keyword super is necessary to correctly implement method-call resolution in the presence of optional methods. The keyword super and the capability for load-time variability is only used in variant simulators (not in configurable programs or in program variants).

An FJsim program consists of a representation of the code base (CT, init) and a configuration Φ. The configuration is set at load time to simulate only a certain variant. Figure 5.6 shows the syntax of FJsim. Similar to keyword this in FJ, we assume that there is a special variable named super that it is never used as the name of a formal parameter of a method declaration. It is considered to be implicitly bound in every method declaration. During evaluation, super is substituted with a *method-lookup-annotation* term that states in which class the lookup for the super method should start.

Finally, FJsim provides a conditional-execution construct called *feature choice*. The feature-choice construct uses a presence condition over feature

Syntax

t	::=	. . .	*terms:*
		(ϕ ? t : t)	*feature choice*
		t.@C.m($\bar{t}$)	*method-lookup annotation*

Figure 5.6: The syntax FJSim adds to FJ. ϕ denotes a presence condition. @C.m() denotes that the lookup for method m is started in class C.

variables to select one of two alternative terms at run time. The selection is made depending on configuration Φ, which has been fixed at load time, though.

Syntax At the syntax level, our extensions require two additional terms: First, we introduce the ternary operator presence condition ? then : else with a similar semantics as in Java. During evaluation of a ternary operator, its presence condition ϕ is evaluated with respect to the given configuration Φ (i.e., $sat(\phi \wedge \Phi)$ means that the presence condition is satisfiable in configuration Φ). Second, we introduce the syntax construct t.@C.m($\bar{t}$) which denotes that the lookup for method m is started in class C (and may continue in superclasses of C). When the method is executed, t is used as this. We use the syntax extension to model the method-lookup strategy as known from super in Java. In the variant simulator, super provides support for accessing overridden methods in subject programs. The syntax extension t.@C.m($\bar{t}$) is necessary as a term starting with super does not contain information on which class it is embedded in (and where method lookup should start) [Pie02].

Typing Figure 5.7 shows the typing rules, auxiliary functions and evaluation rules we introduce for FJSim. The upper part of the figure shows typing rules for typing feature choices, as well as rules for typing the superclass lookup. The expression $\Gamma \vdash$ t : C denotes that the term t is of type C in context Γ, which maps bound variables to types. T-VarEnc enforces that the terms in the then and else branches of a feature choice have the same type. T-SuperRef enforces that one of the superclasses actually implements the method referred to in a call with lookup annotation. All other typing rules are identical to the rules of FJ [Pie02] and omitted for brevity.

Evaluation The lower part of Figure 5.7 shows the evaluation rules that FJSim adds to FJ. The evaluation rule E-InvkNewSuper resolves references to superclasses ((new C(...)).@D) and searches for a method implementation in the superclass D. The keyword super itself is already handled earlier, during the

Typing

$$\frac{\Gamma \vdash \mathsf{t}_0 : \mathsf{C} \quad \Gamma \vdash \mathsf{t}_1 : \mathsf{C}}{\Gamma \vdash (\phi \mathrel{?} \mathsf{t}_0 : \mathsf{t}_1) : \mathsf{C}} \qquad (\text{T-VarEnc})$$

$$\frac{\begin{array}{c} CT(\mathsf{E}) = \mathsf{class\,E} \ldots \{\ldots \mathsf{F\,m}(\overline{\mathsf{G\,f}})\{\ldots\}\} \\ \mathsf{D} <: \mathsf{E} \quad \Gamma \vdash \mathsf{t} : \mathsf{C} \quad \mathsf{C} <: \mathsf{D} \quad \Gamma \vdash \overline{\mathsf{a} : \mathsf{H}} \quad \overline{\mathsf{H} <: \mathsf{G}} \end{array}}{\Gamma \vdash \mathsf{t.@D.m}(\overline{\mathsf{a}}) : \mathsf{F}} \qquad (\text{T-SuperRef})$$

$$\frac{\begin{array}{c} \overline{\mathsf{x} : \mathsf{C}}, \mathsf{this} : \mathsf{C}_0, \mathsf{super} : \mathsf{D} \vdash \mathsf{t}_0 : \mathsf{E}_1 \quad \mathsf{E}_1 <: \mathsf{C}_1 \\ CT(\mathsf{C}_0) = \mathsf{class\,C\,extends\,D}\{\ldots\} \\ override(\mathsf{m}, \mathsf{D}, \overline{\mathsf{C}} \to \mathsf{C}_1) \end{array}}{\Gamma \vdash \mathsf{C}_1\ \mathsf{m}(\overline{\mathsf{C\,x}})\{\mathsf{return}\ \mathsf{t}_0;\}\ \mathsf{OK\ in}\ \mathsf{C}_0} \qquad (\text{Method Typing, replaces rule from [Pie02]})$$

Auxiliary functions

$$\frac{\begin{array}{c} CT(\mathsf{C}_0) = \mathsf{class\,C_0\,extends\,D}\{\overline{\mathsf{C\,f};}\ \mathsf{K}\,\overline{\mathsf{M}}\} \\ \mathsf{B}\,m\,(\overline{\mathsf{B\,x}})\{\mathsf{return\,t};\} \in \mathsf{M} \end{array}}{mbody(\mathsf{m}, \mathsf{C}_0) = (\overline{\mathsf{x}}, [\mathsf{super} \mapsto \mathsf{this.@D}]\,\mathsf{t})} \qquad \frac{\begin{array}{c} CT(\mathsf{D}) = \mathsf{class\,D\,extends\,E}\{\overline{\mathsf{C\,f};}\ \mathsf{K}\,\overline{\mathsf{M}}\} \\ \mathsf{C} <: \mathsf{D} \quad \mathsf{C} \neq \mathsf{D} \quad \mathsf{m} \in \overline{\mathsf{M}} \end{array}}{hasSuperImpl(\mathsf{C}, \mathsf{m})}$$

Evaluation

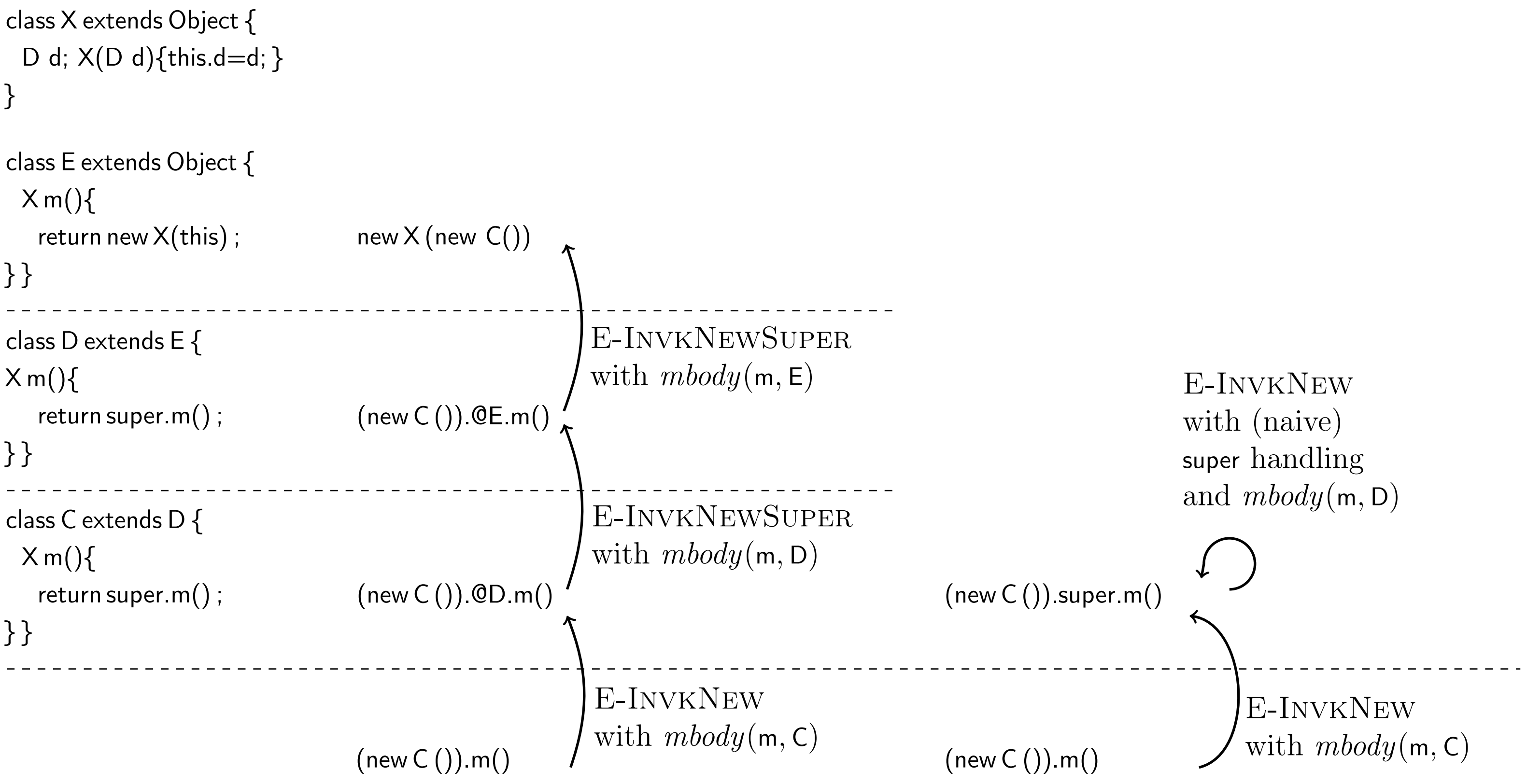
$$\frac{sat(\phi \wedge \Phi)}{(\phi \mathrel{?} \mathsf{t}_0 : \mathsf{t}_1) \to \mathsf{t}_0} \quad (\text{E-VarEnc-En}) \qquad \frac{mbody(\mathsf{m}, \mathsf{D}) = (\overline{\mathsf{x}}, \mathsf{t}_0)}{\begin{array}{c}(\mathsf{new\,C}\,(\overline{\mathsf{v}})).\mathsf{@D.m}(\overline{\mathsf{u}}) \to \\ [\overline{\mathsf{x} \mapsto \mathsf{u}}, \mathsf{this} \mapsto \mathsf{new\,C}\,(\overline{\mathsf{v}})]\,\mathsf{t}_0\end{array}} \quad (\text{E-InvkNewSuper})$$

$$\frac{sat(\neg\phi \wedge \Phi)}{(\phi \mathrel{?} \mathsf{t}_0 : \mathsf{t}_1) \to \mathsf{t}_1} \quad (\text{E-VarEnc-Dis})$$

Figure 5.7: The typing rules, auxiliary functions, and evaluation rules FJSim adds to FJ; ϕ denotes a presence condition, Φ a configuration.

call to the auxiliary function *mbody*, which we redefined in Figure 5.7. The redefined *mbody* function replaces super used in a class C with this.@D where D is the superclass of C. A method call (new C(...)).@D.m(...) executes the method m from class D on the object new C(...). Figure 5.8 shows an example evaluation of super and motivates why we need the @D notation to correctly evaluate super.

In Figure 5.8b, the initial term (new C ()).m() is evaluated using the standard rule E-InvkNew [Pie02] and *mbody*(m,C) = this.@D.m(). It substitutes this with new C() resulting in the term (new C ()).@D.m(), which is then evaluated with



```
class X extends Object {
  D d; X(D d){this.d=d; }
}

class E extends Object {
  X m(){
    return new X(this) ;
}}

class D extends E {
X m(){
    return super.m() ;
}}

class C extends D {
  X m(){
    return super.m() ;
}}
```

new X (new C())
E-InvkNewSuper with $mbody$(m, E)
(new C ()).@E.m()
E-InvkNewSuper with $mbody$(m, D)
(new C ()).@D.m()
E-InvkNew with $mbody$(m, C)
(new C ()).m()
E-InvkNew with (naive) super handling and $mbody$(m, D)
(new C ()).super.m()
E-InvkNew with $mbody$(m, C)
(new C ()).m()

(a) Program

(b) Evaluation with method-lookup annotation

(c) Incorrect evaluation without method-lookup annotation

Figure 5.8: An example of correct evaluation of super references in FJsim (b) and incorrect evaluation in FJsim without method-lookup annotations (c)

E-InvkNewSuper; this in new X(this) is again substituted with new C(). The resulting term (new C()).@E.m() is then evaluated to (new X((new C()))). If we would not insert the @D annotation, we could not know in which superclass to start searching for implementations of method m. In this case, we might select the implementation from D (the superclass of the this object) again and generate an (incorrect) endless loop (cf. Figure 5.8c).

The evaluation rules E-VarEnc-En and E-VarEnc-Dis define how feature choices are evaluated. The evaluation of an FJsim program is deterministic, as the configuration Φ used for evaluation of the feature choices has only one satisfying assignment. This way, in each step evaluating a feature choice, either E-VarEnc-En or E-VarEnc-Dis is applied. As stated earlier, our model of load-time variability guarantees that each configuration has only one satisfying assignment, so no variability remains during the actual execution of an FJsim program (Section 2.1.1). If we would not enforce this property, both evaluation rules might be applicable to the same term (the presence condition ϕ *and* its negation $\neg\phi$ would be satisfiable) and the program behavior would be non-deterministic. When variant simulators are analyzed, for example with a model checker, this non-determinism helps to explore identical execution paths from many variants simultaneously as we discuss in Section 5.5 and in Chapter 6.

5.2.4 Generation of Variants and Variant Simulators

Generation of variants Variant generation derives an FJ program from an CFJ program based on a given configuration Φ. Kästner et al. formalized the generation of variants [KATS12]. In Figure 5.9 we show the relevant subset of these rules.

The function *derive* takes a CFJ program (CT, AT, MT, init, $\hat{\Phi}$) and a valid configuration Φ. It returns a corresponding variant implemented in FJ (the program is also an FJsim program without feature choices and super calls). It uses the auxiliary functions $\curlyeqprec \cdot \curlyeqsucc$, $\ll \cdot \gg$, and $[\![\cdot]\!]$. These functions transform a single CFJ syntax element (class, method, term) into an FJ syntax element. Function $\curlyeqprec \cdot \curlyeqsucc$ traverses the elements of the program recursively and invokes the derivation of term variants. Function $\ll \cdot \gg$ chooses between alternative program elements according to the given configuration Φ by iterating alternatives defined in MT. For example, $\ll MT$ (printMulti(f,b)), printDuplex(f,b) $\gg$ selects between printMulti(f,b) and its alternative printDuplex(f,b) in Figure 5.5. Function $[\![\cdot]\!]$ removes program elements if their presence condition is not satisfied. For example, the method definition of printDuplex is eliminated in $[\![$void printDuplex(Page f, Page b) {...}$]\!]$ iff its presence condition in AT is not satisfied.

$$\begin{array}{rcl} \mathit{derive} : (\text{CFJ program}, \text{Configuration}) & \rightarrow & \text{FJ program} \\ \mathit{derive}((CT, AT, MT, \mathsf{init}, \hat{\Phi}), \Phi) & = & (\preccurlyeq(\mathit{range}(CT), \mathsf{init})\succcurlyeq) \end{array}$$

$$\begin{array}{rcll} \preccurlyeq\succcurlyeq : \text{CFJ term} & \rightarrow & \text{FJ}\textsc{sim}\text{ term} & \\ \preccurlyeq\mathsf{v}\succcurlyeq & = & \mathsf{v} & \text{(G.1)} \\ \preccurlyeq\mathsf{t.f}\succcurlyeq & = & \preccurlyeq\mathsf{t}\succcurlyeq\mathsf{.f} & \text{(G.2)} \\ \preccurlyeq\mathsf{t.m}(\bar{\mathsf{t}})\succcurlyeq & = & \preccurlyeq\mathsf{t}\succcurlyeq\mathsf{.m}(\preccurlyeq\bar{\mathsf{t}}\succcurlyeq) & \text{(G.3)} \\ \preccurlyeq\mathsf{new\ C}(\bar{\mathsf{t}})\succcurlyeq & = & \mathsf{new\ C}(\preccurlyeq\bar{\mathsf{t}}\succcurlyeq) & \text{(G.4)} \\ \preccurlyeq\mathsf{C\ m}(\overline{\mathsf{C\ x}})\{\,\mathsf{return\ t};\}\succcurlyeq & = & \mathsf{C\ m}(\overline{\mathsf{C\ x}})\,\{\,\mathsf{return}\ \ll MT(\mathsf{t}), \mathsf{t} \gg ;\} & \text{(G.5)} \\ \preccurlyeq\mathsf{class\ C\ extends\ D}\ \{\,\overline{\mathsf{C\ f}};\ \mathsf{K}\ \overline{\mathsf{M}}\}\succcurlyeq & = & \mathsf{class\ C\ extends\ D}\ \{\,\overline{\mathsf{C\ f}};\ \mathsf{K}\preccurlyeq[\![\overline{\mathsf{M}}]\!]\succcurlyeq\} & \text{(G.6)} \\ \preccurlyeq(\bar{\mathsf{L}}, \mathsf{t})\succcurlyeq & = & (\preccurlyeq\bar{\mathsf{L}}\succcurlyeq, \mathsf{t}) & \text{(G.7)} \end{array}$$

$$\begin{array}{rcl} \ll\gg & : & \text{CFJ term} \times \text{CFJ term} \rightarrow \text{FJ}\textsc{sim}\text{ term} \\ \ll \mathsf{t}_1, \mathsf{t}_2 \gg & = & \begin{cases} \preccurlyeq\mathsf{t}_1\succcurlyeq & \mathsf{t}_1 \neq \bullet, \quad \mathit{sat}(\Phi \wedge AT(\mathsf{t}_1)) \\ \ll MT(\mathsf{t}_1), \mathsf{t}_2 \gg & \mathsf{t}_1 \neq \bullet, \quad \neg\mathit{sat}(\Phi \wedge AT(\mathsf{t}_1)) \\ \preccurlyeq\mathsf{t}_2\succcurlyeq & \mathsf{t}_1 = \bullet\,(\mathit{otherwise}) \end{cases} \end{array}$$

$$\begin{array}{rcl} [\![]\!] & : & \text{CFJ term} \rightarrow \text{FJ}\textsc{sim}\text{ term} \\ [\![\mathsf{a}]\!] & = & \begin{cases} \mathsf{a} & \mathit{sat}(\Phi \wedge AT(\mathsf{a})) \\ \bullet & \mathit{otherwise} \end{cases} \end{array}$$

Figure 5.9: Variant generation rules, adopted from Kästner et al. [KATS12]; lists are processed element-wise, e.g., $[\![\mathsf{t}_1, \mathsf{t}_2, \ldots, \mathsf{t}_n]\!] = [\![\mathsf{t}_1]\!], [\![\mathsf{t}_2]\!], \ldots, [\![\mathsf{t}_n]\!]$; $\bullet$ denotes the empty program element and $\preccurlyeq \bullet \succcurlyeq\ =\ \bullet$; $\mathit{range}(CT)$ denotes all class definitions in class table CT.

Variant-simulator generation The generation of variant simulators is similar to the generation of variants. The main differences are that, the target language is FJSIM and that instead of removing optional program elements, we encode this variability by means of feature choices. For example, the expression printMulti(f,b) and its alternative printDuplex(f,b) in Figure 5.5 are encoded as $(\mathit{Duplex}\,?\,\mathsf{printDuplex(f,b)}\!:\mathsf{printMulti(f,b)})$ in FJSIM. Figure 5.10 shows the definition of function *encode*. Function *encode* generates a variant simulator in FJSIM for a given CFJ program $(CT, AT, MT, \mathsf{init}, \hat{\Phi})$. Function *encode* uses the auxiliary functions $\preccurlyeq \cdot \succcurlyeq$ and $\ll \cdot \gg$, which we redefine in Figure 5.10. The figure also defines function $[\![\mathsf{t}]\!]_{\mathsf{C}}$, where C denotes the class containing the term t. The omitted cases of $\preccurlyeq \cdot \succcurlyeq$ are the same as in Figure 5.9. Function $\ll \mathsf{t}_1, \mathsf{t}_2 \gg$ iterates through all alternatives of term t_1, introducing feature choices. It uses

$$\begin{array}{rcl} \mathit{encode} : \text{CFJ program} & \rightarrow & \text{FJ}\textsc{sim}\text{ program} \\ \mathit{encode}(CT, AT, MT, \mathsf{init}, \hat{\Phi}) & = & (\curlyeqprec(\mathit{range}(CT), \mathsf{init})\curlyeqsucc, \hat{\Phi}) \end{array}$$

$$\begin{array}{rcl} \curlyeqprec\curlyeqsucc : \text{CFJ term} & \rightarrow & \text{FJ}\textsc{sim}\text{ term} \\ & \ldots & \\ \curlyeqprec\mathsf{class\ C\ extends\ D\ \{\ \overline{C\ f};\ K\ \overline{M}\}}\curlyeqsucc & = & \mathsf{class\ C\ extends\ D\ \{\ \overline{C\ f};\ K}\curlyeqprec[\![\overline{\mathsf{M}}]\!]_{\mathsf{C}}\curlyeqsucc\} \quad \text{(G.6)} \\ & \ldots & \end{array}$$

$$\begin{array}{rcl} \ll\gg & : & \text{CFJ term} \times \text{CFJ term} \rightarrow \text{FJ}\textsc{sim}\text{ term} \\ \ll \mathsf{t}_1, \mathsf{t}_2 \gg & = & \begin{cases} AT(\mathsf{t}_1)\,?\curlyeqprec\mathsf{t}_1\curlyeqsucc : \ll MT(\mathsf{t}_1), \mathsf{t}_2 \gg & \mathsf{t}_1 \neq \bullet \wedge \mathit{sat}(\hat{\Phi} \wedge AT(\mathsf{t}_1)) \\ \ll MT(\mathsf{t}_1), \mathsf{t}_2 \gg & \mathsf{t}_1 \neq \bullet \wedge \neg\mathit{sat}(\hat{\Phi} \wedge AT(\mathsf{t}_1)) \\ \curlyeqprec\mathsf{t}_2\curlyeqsucc & \mathsf{t}_1 = \bullet\,(\mathit{otherwise}) \end{cases} \end{array}$$

$$\begin{array}{rcl} [\![]\!]_{\mathsf{C}} & : & \text{CFJ method definition} \rightarrow \text{FJ}\textsc{sim}\text{ method definition} \\ [\![\mathsf{a}]\!]_{\mathsf{C}} & = & \begin{cases} \mathsf{D\ m(\overline{C\ x})\,\{\,return} & \mathsf{a = D\ m(\overline{C\ x})\,\{\,return\,t;\,\}} \\ \qquad (AT(\mathsf{a})\,?\curlyeqprec\mathsf{t}\curlyeqsucc\ :\ \mathsf{super.m(\overline{C\ x})); \}} & \mathit{sat}(\hat{\Phi} \wedge AT(\mathsf{a})) \quad \mathit{hasSuperImpl}(\mathsf{C}, \mathsf{m}) \\ \mathsf{D\ m(\overline{C\ x})\,\{\,return\,super.m(\overline{C\ x});\,\}} & \mathsf{a = D\ m(\overline{C\ x})\,\{\,return\,t;\,\}} \\ & \neg\mathit{sat}(\hat{\Phi} \wedge AT(\mathsf{a})) \quad \mathit{hasSuperImpl}(\mathsf{C}, \mathsf{m}) \\ \mathsf{D\ m(\overline{C\ x})\,\{\,return}\curlyeqprec\mathsf{t}\curlyeqsucc;\,\} & \mathsf{a = D\ m(\overline{C\ x})\,\{\,return\,t;\,\}} \\ & \mathit{sat}(\hat{\Phi} \wedge AT(\mathsf{a})) \quad \neg\mathit{hasSuperImpl}(\mathsf{C}, \mathsf{m}) \\ \mathsf{a} & \mathit{otherwise} \end{cases} \end{array}$$

Figure 5.10: Variant-simulator generation rules. $\ll \cdot \gg$ introduces *feature choices*, if multiple terms are feasible. For brevity, we omit the propagation of AT, MT, and $\hat{\Phi}$.

the default term t_2 as innermost else case. All functions employ the variability model $\hat{\Phi}$ instead of a configuration Φ, so that variable parts are only dropped if their presence condition is not satisfiable in $\hat{\Phi}$. Function $[\![\cdot]\!]_{\mathsf{C}}$ handles optional methods. The function introduces a call to the same method in a superclass of C if there exists a variant in which the currently generated method is not present. In this case, a call to the current method will execute the superclass method in the variant. We model this behavior in the variant simulator using the keyword super.

5.3 Behavior Preservation

Many applications that use variability encoding depend on the fact that variability encoding preserves the behavior of the simulated variants. This includes, in particular, all control-flow sensitive applications, such as verification [ASW+11; AvRW+13; TSAH12] or testing [KvRE+12]. In this section, we prove the behavior-preservation property for variability encoding based on our model of Section 5.2. As the key result of this chapter, our proof guarantees that a variant simulator can be used for behavioral analysis of the variants it simulates. It shows that the execution of a variant simulator and the corresponding variants exhibit the same observable behavior, as illustrated in Figure 5.2.

In particular, we prove that the behavior of each variant of a configurable system is *weakly bisimilar* to the variant simulator, if the variant simulator is executed with the variant's configuration. We use weak bisimulation [Mil99] as proof technique to show that each execution trace in a variant is represented by a trace in the variant simulator with the configuration that corresponds to the variant. We use the *weak* form of bisimulation to allow the occurrence of additional feature-choice transitions in the simulator. Next, we define a trace semantics for FJSIM programs in Section 5.3.1, which we use to prove behavior preservation in Section 5.3.2.

5.3.1 A Trace Semantics for FJSIM Programs

We introduce a trace semantics for FJSIM that encodes the run-time semantics defined by the evaluation rules (Figures 5.4 and 5.7). This way, the behavior of a program is represented as a transition system, which is better suited for our bisimulation proof than the source code representation. In particular, we model the run-time behavior of a variant π_Φ and a variant simulator σ. Using function $genTS(CT, \mathsf{t})$ of Figure 5.11, we define the transition system of a program $p = (\mathsf{CT}, \mathsf{init}, \Phi)$ as $genTS(\mathsf{CT}, \mathsf{init})$. The generated system is a labeled transition system (S, T, PC), where S is the set of states, T is the set of transitions $(T \subseteq S \times PC \times S)$, and PC is the set of presence conditions. States in the transition system represent FJSIM terms, and transitions represent evaluation steps that rewrite one term into another. Presence conditions are propositional formulas over variables in the set of configuration options. Transitions are labeled with presence conditions that have to hold during evaluation in order to proceed with the respective evaluation step. A trace is a sequence of states of the transition system starting in the initial state (term init). Each trace represents an execution path in the corresponding FJSIM program. Function *genTS* recursively processes all terms of an FJSIM program, and adds states and transitions for each term to the transition system. The generation for transition systems for FJ programs (system variants) is defined analogous.

5.3.2 Proof of Behavior Preservation

We assume that all valid variants of a given CFJ program with code base Δ are well typed with respect to the variability model $\hat{\Phi}$ [KATS12], as the behavior of

$genTS$: (FJsim CT, FJsim term) → Transition system ($States$, $Transitions$, $Presence\ conditions$)

$$
genTS(CT,\mathsf{t}) = \begin{cases}
\begin{array}{ll}
\textsc{Termination case}\quad (\text{if } \mathsf{t} \text{ is a value}) & \\
(\{\mathsf{v}\},\emptyset,\emptyset) & \mathsf{t}=\mathsf{v} \\ \hline
\text{TS-}\textsc{ProjNew} & \mathsf{t} = ((\mathsf{new\ C}(\bar{\mathsf{v}})).\mathsf{f}_j) \text{ and} \\
(\{\mathsf{t},\mathsf{v}_j\},\{(\mathsf{t},\mathit{true},\mathsf{v}_j)\},\{\mathit{true}\}) & \mathsf{f}_j \in \mathit{fields}(\mathsf{C}) \text{ and } v_j \in \bar{\mathsf{v}} \\ \hline
\text{TS-}\textsc{CastNew} & \\
(\{\mathsf{t},\mathsf{t}'\},\{(\mathsf{t},\mathit{true},\mathsf{t}')\},\{\mathit{true}\}) \Cup genTS(CT,\mathsf{t}') & \mathsf{t} = (\mathsf{D})(\mathsf{new\ C}(\bar{\mathsf{v}})) \text{ and} \\
\text{ with } \mathsf{t}' = \mathsf{new\ C}(\bar{\mathsf{v}}) & \mathsf{C} <: \mathsf{D} \\ \hline
\text{TS-}\textsc{InvkNewSuper} & \\
(\{\mathsf{t},\mathsf{t}'\},\{(\mathsf{t},\mathit{true},\mathsf{t}')\},\{\mathit{true}\}) \Cup genTS(CT,\mathsf{t}') & \mathsf{t} = (\mathsf{new\ C}\ (\bar{\mathsf{v}})).@\mathsf{D}.m(\bar{\mathsf{u}}) \\
\text{ with } \mathsf{t}' = [\overline{\mathsf{x} \mapsto \mathsf{u}}, \mathsf{this} \mapsto \mathsf{new\ C}(\bar{\mathsf{v}})]\,\mathsf{t}_0 & \text{ and} \\
 & \mathit{mbody}(\mathsf{m},\mathsf{D}) = (\bar{\mathsf{x}},\mathsf{t}_0) \\ \hline
\text{TS-}\textsc{InvkNew} & \\
(\{\mathsf{t},\mathsf{t}'\},\{(\mathsf{t},\mathit{true},\mathsf{t}')\},\{\mathit{true}\}) \Cup genTS(CT,\mathsf{t}') & \mathsf{t} = ((\mathsf{new\ C}(\bar{\mathsf{v}})).\mathsf{m}(\bar{\mathsf{u}})) \text{ and} \\
\text{ with } \mathsf{t}' = [\overline{\mathsf{x} \mapsto \mathsf{u}}, \mathsf{this} \mapsto \mathsf{new\ C}(\bar{\mathsf{v}})]\,\mathsf{t}_0 & \mathit{mbody}(\mathsf{m},\mathsf{C}) = (\bar{\mathsf{x}},\mathsf{t}_0) \\ \hline
\text{TS-}\textsc{VarEnc-En} \text{ and TS-}\textsc{VarEnc-Dis} & \\
(\{\mathsf{t},\mathsf{t}_0\},\{(\mathsf{t},\phi,\mathsf{t}_0)\},\{\phi\}) \Cup & \mathsf{t} = (\phi\,?\,\mathsf{t}_0 : \mathsf{t}_1) \\
(\{\mathsf{t},\mathsf{t}_1\},\{(\mathsf{t},\neg\phi,\mathsf{t}_1)\},\{\neg\phi\}) \Cup & \\
genTS(CT,\mathsf{t}_0) \Cup genTS(CT,\mathsf{t}_1) & \\ \hline
\textit{Cases for congruence rules are omitted} &
\end{array}
\end{cases}
$$

Figure 5.11: Definition of function $genTS$ for the generation of a transition system ($States$, $Transitions$, $Presence\ Conditions$) from an FJsim program. For legibility, we define the join operation $\Cup$ on transition systems (S,T,PC) and (S',T',PC') as follows: $(S,T,PC) \Cup (S',T',PC') = (S \cup S', T \cup T', PC \cup PC')$.

ill-typed variants is undefined [Pie02]. To prove that the behavior of all variants is preserved by the corresponding variant simulators (i.e., that variability encoding is sound), we create a variant π_Φ with function $derive(\Delta,\Phi)$ for every configuration Φ. The generated variant $\pi_\Phi = (CT_\Phi, \mathsf{init}, true)$ has a class table CT_Φ, which is constructed with the function $encode$ of Figure 5.9 and a start term init. The corresponding variant simulator $\sigma = (CT_\sigma, \mathsf{init}, \hat{\Phi})$ is generated using function $encode(\Delta,\hat{\Phi})$ of Figure 5.9 and Figure 5.10. Hence, the variant simulator σ and the variant π_Φ are constructed from the same configurable code base Δ, and the execution of both programs, σ and π_Φ, starts with the term init.

We generate the transition systems for the variant π_Φ and the variant

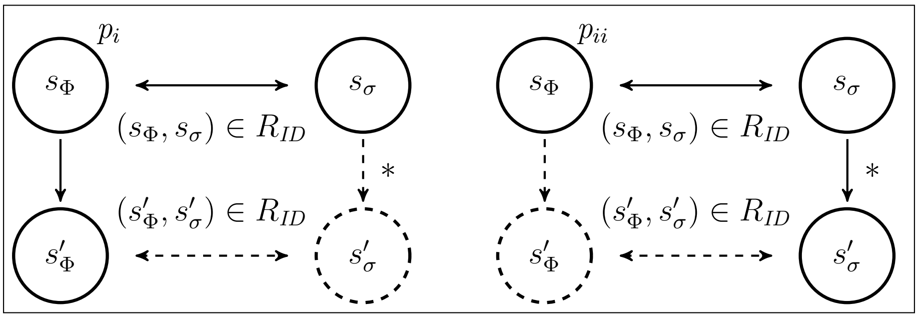

p_i
s_Φ
s_σ
$(s_\Phi, s_\sigma) \in R_{ID}$
$*$
$(s'_\Phi, s'_\sigma) \in R_{ID}$
s'_Φ
s'_σ
p_{ii}
s_Φ
s_σ
$(s_\Phi, s_\sigma) \in R_{ID}$
$*$
$(s'_\Phi, s'_\sigma) \in R_{ID}$
s'_Φ
s'_σ

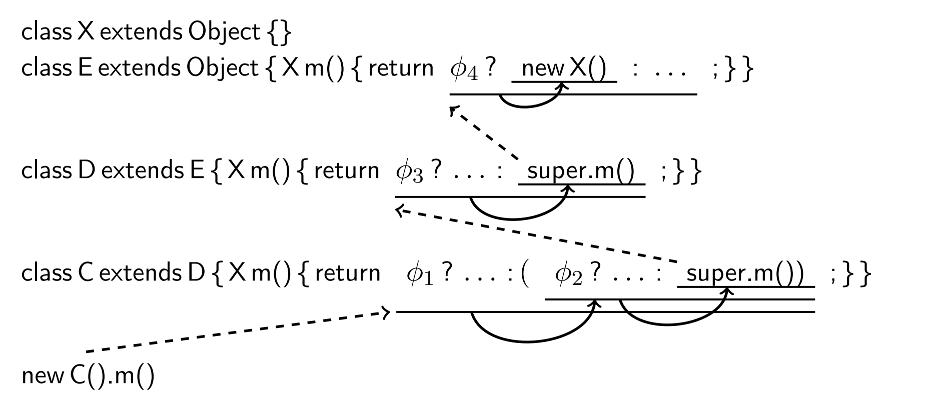
Figure 5.12: The weak bisimulation property has two sub-properties p_i and p_{ii}. The sub-properties prove the existence of dashed relations and states assuming solid ones.

simulator σ as defined in Section 5.3.1. The transition system for variant π_Φ is denoted with $(S_\Phi, \rightarrow_\Phi, PC)$ and derived by $genTS(CT_\Phi, \mathsf{init})$, where S_Φ is the set of states of the system, PC is the set of labels (i.e. presence conditions) on the transitions, and $\rightarrow_\Phi$ is the set of transitions ($\rightarrow_\Phi \subseteq S_\Phi \times PC \times S_\Phi$).

The transition system for variant simulator σ is denoted with $(S_\sigma, \rightarrow_\sigma, PC)$ and derived by $genTS(CT_\sigma, \mathsf{init})$. When we execute the simulator σ with configuration Φ, we use the corresponding projection of the transitions from $\rightarrow_\sigma$: $\rightarrow_{\sigma|\Phi} = \{(s, pc, s') \in \rightarrow_\sigma | sat(\Phi \wedge AT(pc))\}$. This corresponds to execution of a simulator in a real execution environment that evaluates the presence conditions of feature choices with respect to a configuration. For clarity, we denote states from the variant transition system with s_Φ and s'_Φ, and states from the variant-simulator transition system with s_σ and s'_σ. If it is clear from the context, we omit the subscripts in the transition relations $\rightarrow_\Phi$ and $\rightarrow_{\sigma|\Phi}$.

Based on these definitions, Figure 5.12 illustrates the weak bisimulation property we want to prove.[6] R_{ID} is the *simulation relation*, which relates states of the variant transition system to states of the variant-simulator transition system. In our proof, we use the syntactic equality of terms in FJSIM as simulation relation. For two states s_Φ and s_σ, $(s_\Phi, s_\sigma) \in R_{ID}$ holds, iff the terms represented by s_Φ and s_σ are equal. This is possible because each FJ term is by definition also an FJSIM term. The weak bisimulation property has two sub-properties (p_i and p_{ii}), which must both be proved. Property p_i states that, for each direct successor state s'_Φ of s_Φ, there exists a state s'_σ in the variant-simulator transition system that is related to the state s'_Φ of the variant

[6] We choose the *weak bisimulation* property over the often weaker *weak-trace-equivalence* property because both properties are equal in our case. Bisimulation and trace equivalence are different iff a state can have two equally-labeled outgoing edges leading to different states. Such behavior often occurs in the context of concurrent execution. In our transition system, an FJSIM term is always evaluated with a specific evaluation rule yielding exactly one term (assuming a valid configuration Φ).



```
class X extends Object {}
class E extends Object { X m() { return  φ4 ?  new X()  :  ...  ; } }

class D extends E { X m() { return  φ3 ? ... :  super.m()  ; } }

class C extends D { X m() { return  φ1 ? ... : (  φ2 ? ... :  super.m())  ; } }

new C().m()
```

Figure 5.13: Proof concept in presence of overriding methods. The initial term (bottom) is evaluated as shown by the arrows. We prove existence of the dashed arrows in Case 3 of Theorem 1 and existence of the solid arrows in Lemma 1.

transition system with $(s'_\Phi, s'_\sigma) \in R_{ID}$, and that s'_σ is a successor state of s_σ. Property p_{ii} states that, for each successor state s'_σ of s_σ, there exists a state s'_Φ in the variant-simulator transition system that is related to the state s'_σ of the variant transition system, with $(s'_\Phi, s'_\sigma) \in R_{ID}$, and that s'_Φ is a successor state of s_Φ. In *weak* bisimulation, s'_σ does not need to be a *direct* successor to s_σ; in our case there may be a number of auxiliary states in between that evaluate feature choices. We denote such a sequence of states linked by consecutive transitions with $s_\sigma \xrightarrow{*} s'_\sigma$. A $s_\sigma \xrightarrow{*} s'_\sigma$ sequence starts with an evaluation rule from normal FJ and continues with zero or more applications of the new FJSIM evaluation rules (E-VARENC-EN, E-VARENC-DIS, or E-INVKNEWSUPER).

A particularly interesting part of the proof is how we prove correctness in the presence of overriding methods. We moved a corresponding part of the proof to Lemma 1, to simplify understanding. Figure 5.13 shows an example for how the lemma is used. It shows four classes of a variant simulator and a term (new C()).m() that is evaluated. The classes C, D, and E implement method m. The term (new C()).m() is evaluated as shown in the figure if the presence conditions ϕ_1, ϕ_2, and ϕ_3 are not satisfiable, and ϕ_4 is satisfiable in a configuration Φ. In this case, term (new C()).m() evaluates to term new X(), defined in E. As a consequence, the transition system of the variant simulator must contain a path from (new C()).m() to new X(), which may contain auxiliary states. We prove the existence of this path in two steps. First, Lemma 1 proves that the intra-method dispatch among alternative implementations is resolved correctly (solid arrows in Figure 5.13). Second, Case 3 in the proof of Theorem 1 shows that the steps from overriding to overridden methods are evaluated correctly (dashed arrows in Figure 5.13).

Lemma 1. *Given (1) a configuration Φ, (2) the method body* $\mathsf{return\,t;}$ *of a method* $\mathsf{m(\overline{x})}$ *in class* C*, and (3) that a call* $\mathsf{(new\,C(\overline{v})).m(\overline{u})}$ *is type correct in Φ, there exists a chain of consecutive states* $\mathsf{t} \xrightarrow{*} \ldots$ *that either evaluate* t (i) *to a term* t_Φ *that has a presence condition satisfied by Φ and is one of* m*'s alternative implementations in* C*, or* (ii) *to* $\mathsf{(new\,C(\overline{v}))}.@D.m(\mathsf{\overline{u}})$ *if no such term* t_Φ *exists and a superclass of* C *implements* m*. This evaluation sequence uses only the rules* TS-VarEnc-En *and* TS-VarEnc-Dis*.*

Proof. We use induction over the number of feature choices n in term t to prove Lemma 1. The induction hypothesis is that each subterm of t evaluates to t_Φ (Case i) or to $\mathsf{(new\,C(\overline{v}))}@D.m(\mathsf{\overline{u}})$ (Case ii) if it is annotated with a presence condition satisfied in Φ and has n or less feature choices . In both cases, the resulting term does not contain the keyword super. In Case (i), the term is a part of the CFJ program and as such cannot use super. In Case (ii), super has been substituted with a super reference ($\mathsf{this.@..}$) when the method body was loaded with the function *mbody*.

There are two base cases ($n = 0$) which correspond to cases (i) and (ii) in Lemma 1. In Base Case (i), t is one of the alternative implementations of m. In this case, the presence condition $AT(\mathsf{t})$ must be satisfied by Φ, otherwise a call to m is not well typed. In Base Case (ii), t is an invocation of m in the direct superclass of C. Therefore, t equals $\mathsf{(new\,C(\overline{v}))@D.m(\overline{u})}$, and there exists an implementation of m in some superclass of C, because the call is well typed. The base cases are exclusive; a given term can satisfy either (i) or (ii).

In the inductive step $n \to n + 1$, t'_1 is a feature choice $\mathsf{t}'_1 = (\phi\,?\,\mathsf{t}_\phi : \mathsf{t}_{\neg\phi})$. The subterms t_ϕ and $\mathsf{t}_{\neg\phi}$ have n or less feature-choice terms. From the definition of *genTS* (Figure 5.11, TS-VarEnc-En and TS-VarEnc-Dis), we know that the variant-simulator transition system contains the transitions *tr* from t'_1 to t_ϕ and tr' from t'_1 to $\mathsf{t}_{\neg\phi}$ (with the presence conditions ϕ and $\neg\phi$, respectively). Either ϕ is satisfied by configuration Φ (i.e., $sat(\phi \wedge \Phi)$) or the negation $\neg\phi$ is satisfied by Φ. Exactly one of ϕ or $\neg\phi$ is satisfied, because Φ is a configuration, and as such has a fixed value for each configuration option. The code that is invoked if ϕ is satisfied is encoded in t_ϕ and the code that is invoked if $\neg\phi$ is satisfied is encoded in $\mathsf{t}_{\neg\phi}$ (function $\ll \cdot \gg$, Figure 5.10). First, we consider the case where ϕ is satisfied under configuration Φ. Our version of CFJ enforces that feature choices can only occur as the outermost terms in return statements (alternatives are only allowed for method bodies, cf. Section 5.2.2). Thus, if the result of this evaluation step t_ϕ is not a feature choice, it does not contain any further variability and one of the base cases applies. Otherwise, the resulting term t_ϕ is a feature choice. This means, t_ϕ has the same syntactic form as t'_1 in the beginning of the induction step and it is shorter than t'_1 (has one feature choice less). Therefore, we can apply the induction hypothesis. If $\neg\phi$ is satisfied under configuration Φ the proof is analogous. In each step of the evaluation E-VarEnc-En or E-VarEnc-Dis is applied.

The induction shows that there is a sequence of consecutive states $(\mathsf{t}, \mathsf{t}_2, \ldots, \mathsf{t}_n)$ in the variant-simulator transition system that evaluate t to a non-feature-choice term t_n with t_n either being a term with a satisfied presence condition (Case (i)) or a call to an implementation of m in a superclass (Case (ii)). The chain is finite because the term becomes smaller in each iteration as long as the evaluated term is a feature choice. Therefore, the induction hypothesis and the lemma holds. □

Theorem 1. *Given a* CFJ *code base Δ with a variability model $\hat{\Phi}$ and a start term* init*, a configuration Φ, the simulation relation R_{ID} (term equality), a variant $\pi_\Phi = (CT_\Phi, \mathsf{init}, true) = derive(\Delta, \Phi)$, a variant simulator $\sigma = (CT_\sigma, \mathsf{init}, \hat{\Phi}) = encode(\Delta, \hat{\Phi})$, which is executed with Φ, and the corresponding transition systems $(S_\Phi, \rightarrow, PC_\Phi) = genTS(CT_\Phi, \mathsf{init})$ and $(S_\sigma, \rightarrow, PC_\sigma) = genTS(CT_\sigma, \mathsf{init})$, then the weak bisimulation property holds:*

$$\forall s_\Phi \in S_\Phi, \forall s_\sigma \in S_\sigma \text{ with } (s_\Phi, s_\sigma) \in R_{ID} :$$

(p_i) $\forall s'_\Phi$ with $s_\Phi \rightarrow s'_\Phi$: $\exists s'_\sigma \in S_\sigma$ such that $(s_\sigma \overset{*}{\rightarrow} s'_\sigma$ and $(s'_\Phi, s'_\sigma) \in R_{ID})$ and

(p_{ii}) $\forall s'_\sigma$ with $s_\sigma \overset{*}{\rightarrow} s'_\sigma$: $\exists s'_\Phi \in S_\Phi$ such that $(s_\Phi \rightarrow s'_\Phi$ and $(s'_\Phi, s'_\sigma) \in R_{ID})$

Proof. We prove the two properties of bisimulation (p_i and p_{ii}) separately.

Property $\mathbf{p}_i$ We prove p_i with a case distinction over the construction rules used to generate the transition $(s_\Phi \rightarrow s'_\Phi)$, according to the definition of the transition systems (Figure 5.11). Overall, there are eleven cases. However, we omit cases handling congruence rules and focus on the cases of the six evaluation rules shown in Figures 5.4 and 5.7. The omitted cases are very similar to Case 1 shown below. For a complete proof, we refer to the supplementary material attached to our *JLAMP* paper [vRTS+16].

Case 1 (TS-ProjNew)*:* As the transition $(s_\Phi, true, s'_\Phi)$ has been generated with TS-ProjNew (Figure 5.11), f_j is a field in class C of variant π_Φ ($\mathsf{f}_j \in \mathit{fields}(\pi_\Phi, \mathsf{C})$). Thus, there is a valid configuration Φ, in which the field is present in the program. The variant simulator generation rules, in particular G.2 (Figure 5.9 and Figure 5.10), ensure that the field is also present in the variant simulator: $\mathsf{f}_j \in \mathit{fields}(\sigma, \mathsf{C})$. The definition of *genTS* (Figure 5.11, TS-ProjNew) ensures that there is a transition $(s_\sigma \rightarrow s'_\sigma)$ with $s'_\sigma = \mathsf{f}_j$. Therefore, s'_Φ and s'_σ represent the same terms and $(s'_\Phi, s'_\sigma) \in R_{ID}$ holds.

Case 3 (TS-InvkNew)*:* Let $s_\Phi = (\mathsf{new}\ \mathsf{C}(\bar{\mathsf{v}})).\mathsf{m}(\bar{\mathsf{u}})$. Method m in π_Φ must have been generated with variant generation rule G.5 of Figure 5.9. It is important to note that $\bar{\mathsf{v}}$ and $\bar{\mathsf{u}}$ are lists of values (values cannot be evaluated any further). Let t_0 be the body of method m in the variant π_Φ. As the method body is included in π_Φ with Rule G.5, we can conclude that the configuration Φ implies the presence condition $AT(\mathsf{t}_0)$ and $s'_\Phi = \mathsf{t}_0$. As s_Φ is in simulation relation to s_σ, we know that $s_\sigma = (\mathsf{new\ C}(\bar{\mathsf{v}})).\mathsf{m}(\bar{\mathsf{u}})$. From the definition of *genTS* (Figure 5.11, TS-InvkNew), we infer (1) that $\rightarrow_{\sigma|\Phi}$ contains a transition $tr_1 = (s_\sigma \rightarrow ([\overline{\mathsf{x} \mapsto \mathsf{u}}, \mathsf{this} \mapsto \mathsf{new\ C}(\bar{\mathsf{v}})]\mathsf{t}'_1))$ with $mbody(\sigma, \mathsf{m}, \mathsf{C}) = \mathsf{t}'_1$ and (2) that tr_1 has the presence condition *true*.

We use induction over the number of superclasses of C in simulator σ to prove that s_Φ and s_σ evaluate to the same term. In the base case, class C is a direct subclass of Object. Object does not implement any methods [Pie02]. Because each variant is well typed, term s_Φ is also well typed, and we can apply Lemma 1 with configuration Φ. The application of Lemma 1 shows that there exists a list of consecutive states $(s_\sigma, \mathsf{t}_2, \ldots, \mathsf{t}_n)$ in the variant-simulator transition system that evaluate s_σ to either a term t_n with a presence condition satisfied by Φ or to a call of m in a superclass. As C does not have superclasses (other than Object), we know that s_σ evaluates to t_n with $\mathsf{t}_n = s'_\sigma$ and $(s'_\Phi, s'_\sigma) \in R_{ID}$. The evaluation step $s_\sigma \rightarrow \mathsf{t}_2$ applies evaluation rule E-InvkNewSuper. All subsequent evaluation steps, until t_n is reached, apply rules TS-VarEnc-En or TS-VarEnc-Dis which concludes the base case of the induction.

In the inductive step, C has $n + 1$ superclasses, including Object. The term s_Φ is an invocation of method m on class C and the invocation is well typed in variant π_Φ, because either C has an implementation of m in the variant π_Φ or the method invocation is dispatched to an implementation of m in a superclass. If C itself has an implementation, Lemma 1 shows that there is a chain of states in σ from s_σ to s'_σ. As s'_σ is the first alternative implementation of m that satisfies the configuration Φ, s'_σ is in simulation relation to s'_{π_Φ}. If C does not have an implementation of m in the variant π_Φ, Lemma 1 shows that there is a chain of states in simulator σ from s_σ to $(\mathsf{new\,C}(\bar{\mathsf{v}}))@\mathsf{D}.\mathsf{m}(\bar{\mathsf{u}})$. This chain uses only TS-VarEnc-Dis. This new expression invokes method m in the direct superclass D of C. Because $\bar{\mathsf{v}}$ and $\bar{\mathsf{u}}$ are values, the next evaluation rule must be TS-InvkNewSuper. D has n superclasses, so we can apply the induction hypothesis, which states that the call of m on D evaluates to the same term as s_{π_Φ} in variant π_Φ.

Therefore, $s'_\Phi = \mathsf{t}_0$, $s'_\sigma = \mathsf{t}_n$, and $(s'_\Phi, s'_\sigma) \in R_{ID}$. So, there exists a state $s'_\sigma \in S_\sigma$, such that $(s'_\Phi, s'_\sigma) \in R_{ID}$ and a sequence of transitions $(s_\sigma \overset{*}{\rightarrow} s'_\sigma)$. The sequence starts with TS-InvkNew followed by applications of TS-VarEnc-En, TS-VarEnc-Dis, and TS-InvkNewSuper, which concludes Case 3.

Case 4 and Case 5 (TS-VarEnc-En *and* TS-VarEnc-Dis)*:* None of the rules for variant generation (Figure 5.9, G.1–G.8) can generate a feature choice. Therefore, the variant π_Φ cannot contain terms to which rules TS-VarEnc-en and TS-VarEnc-dis may apply. Therefore, Case 4 and Case 5 cannot occur.

*Case 6 (*TS-InvkNewSuper*):* None of the rules for variant generation (Figure 5.9, G.1–G.8) can generate a super term. Therefore, the variant π_Φ cannot contain terms to which rule TS-InvkNewSuper applies and thus Case 6 cannot occur. This concludes the proof of property p_i of the bisimulation property.

Property p$_{ii}$ To prove the second sub-property of bisimulation (property p_{ii}), we have to do a similar case distinction as for the first property (p_i). Case 1 ((TS-ProjNew) can be proven analogous to p_i. We focus on Cases 4, 5, and 6. All other cases can be proven analogous to the the cases for p_i given in the supplementary material attached to our *JLAMP* paper [vRTS+16].

Case 3 (TS-InvkNew) is interesting because we have to show that the simulator cannot evaluate to any states that are not present in a variant, except for states with ternary operators or the @D annotation. Given a term s_Φ = (new C($\bar{\text{v}}$)).m($\bar{\text{u}}$) with $(s_\Phi, s_\sigma) \in R_{ID}$ and a state s'_σ with $(s_\sigma \overset{*}{\rightarrow} s'_\sigma)$, we must show that the variant contains a state s'_Φ with $(s'_\Phi, s'_\sigma) \in R_{ID}$. Interestingly, s'_σ is the *only* state in the sequence $(s_\sigma \overset{*}{\rightarrow} s'_\sigma)$ that *can* be present in S_Φ. All intermediary states between s_σ and s'_σ are evaluated with the rules TS-VarEnc-En, TS-VarEnc-Dis, or TS-InvkNewSuper and, therefore, contain ternary operators or the @D annotation, which cannot occur in a variant. As a result, evaluation in the simulator deviates exactly as far from the variant evaluation as necessary to simulate the correct variant behavior. s_σ evaluates to s'_σ assuming configuration ϕ, and the argumentation of Case 3 in the proof of property p_i shows that s'_σ is equal to the state that s_Φ evaluates to in the variant. Therefore, variant π_Φ has a state s'_Φ with $(s'_\Phi, s'_\sigma) \in R_{ID}$ and a transition $(s_\Phi \rightarrow s'_\Phi)$, which concludes Case 3.

Cases 4, 5 and 6 (TS-VarEnc-En, TS-VarEnc-Dis, and TS-InvkNewSuper) are not relevant, as the given states s_Φ and s_σ are in simulation relation R_{ID} and s_Φ cannot contain ternary operators or the @D annotation. This concludes property p_{ii} and the bisimulation proof.

□

5.4 Variability Encoding Beyond FJ

For the purpose of developing a formal proof, we limited our model of variability encoding as well as the considered languages to a small subset of Java. However, variability encoding is meant to be performed on real programs written in languages such as Java or C. Even for such complex languages it is always possible to build variant simulators trivially using duplication. One can just build every valid program variant and, at program start, dispatch between these variants. However, analysis of such variant simulators is inefficient because similarities among variants cannot be exploited (equivalent to variant-based analysis, see Section 5.5). So, the question arises how additional language concepts of Java and C affect our model (which enables sharing among variants) and our proof of behavior preservation of variant simulators.

As modelling the full set of language features of Java or C is infeasible, we discuss a selection of language constructs that are problematic for variability encoding (e.g., method overloading, optional program variables, and field shadowing). We show how to deal with these constructs on concrete examples. We express the variability in the examples using ifdef directives, because they facilitate a compact notation with fine-grained variability. In most examples, we have to resort to local code duplication as a workaround to solve the variability-encoding challenges. This results in code fragments that are not shared among variants any more. However, in our experience with real applications, the code blowup introduced by these duplications can be kept locally and the most of the code is still shared, as we discuss in Section 5.5. In some cases the variability-encoding challenges and solutions may seem trivial, but it is important to discuss these basic situations before attempting to implement variability encoding in a more complex language. The overarching goal is still that a variant simulator weakly bisimulates all variants. That is, in addition to feature switches, it executes *only* statements that are equivalent to statements of the variant that is currently simulated.

Method overloading Java allows programmers to define multiple methods with the same name in a class iff the signatures of the methods differ (different number or types of parameters). Method overloading is not supported in FJ, and thus neither in CFJ and FJsim. Adding method overloading to CFJ leads to two challenges for variability encoding:

1. In combination with inheritance, methods may overload methods defined in other classes of the inheritance hierarchy. If the overloading method is optional, a simulator generated with the rules of Section 5.2.4 could execute the wrong method.

2. Alternative methods in one class can have identical signatures in the simulator if we allow annotation of single parameters with presence conditions. If this situation is not handled, variability encoding can generate simulators that are not well typed.

overloading and inheritance

Figure 5.14a shows a situation where method overloading and inheritance is combined. Class Y implements method m(B) and inherits an implementation of m(A) from class X. This is a case of overloading because both methods are callable on objects of type Y. The Java run-time environment decides

```
class A {}
class B extends A {}
class X { A m(A a) {...} }
class Y extends X {
  #if (Opt1)
  A m(B b) {return new A();}
  #endif
}

(new Y()).m(new B());
```

(a) Problem: Optional method overloading with inheritance

```
class A {}
class B extends A {}
class X { A m(A a) {...} }
class Y extends X {
  A m(B b) {
    return Opt1 ? new A() : super.m(b);
  }
}
```

(b) Solution: Introduce super also for overloaded methods.

Figure 5.14: Optional method overloading with inheritance in JAVA

```
class A {
  int x = 0;
  #if (Opt1 && Opt2)
  int m() {return 0;}
  #endif
  int m(
    #if (Opt1)
    int x
    #endif
  ) { return x+1; }
}
```

(a) Problem: Method overloading with optional parameters

```
class A {
  int x=0;
  int m() { return Opt1 && Opt2 ? 0 :
            (Opt1 ? x+1 : intErr()); }
  int m (int x) {
    return Opt1 ? x+1 : intErr();
  }
  int intErr() { throw new Error(); }
}
```

(b) Solution: Duplication of the expression Opt1? x+1 : intErr()

Figure 5.15: Method overloading with optional parameters in JAVA

which method to choose depending on the type of the given parameter. The given situation is even more complex because the method implemented in Y is optional and B is a subclass of A. This means that (new Y()).m(new B()) resolves to the implementation of m in Y if Opt1 is satisfiable and to the implementation in X otherwise. We have to model this behavior when designing the variant simulator. A straightforward approach would be to apply method renaming and generate feature switches at all call sites. However, using super, we have a more elegant solution for this problem. Figure 5.14b shows a simulator where we alternatively execute the code from Y or use super to call the method from X. This solution can be implemented in our formalism by altering the premise m $\in \overline{\mathsf{M}}$ in the *hasSuperImpl*(C, m) rule from Figure 5.7, such that it also considers methods that overload m.

```
struct str {
  int x;
  #if (Opt1)
  int y;
  #endif
} str;
int f() {
  return sizeof (str);
}
```

(a) Problem: Run-time system

```
struct str_noOpt1 {
  int x;
} str_noOpt1;
struct str_Opt1 {
  int x;
  int y;
} str_Opt1;
int f() {
  return Opt1 ? sizeof (str_Opt1)
              : sizeof (str_noOpt1);
}
```

(b) Solution: Duplication of the struct str

Figure 5.16: Interaction of variability encoding and environment functions in C

colliding method signatures

As to the second problem, if we allow overloading and optional parameters, as shown in Figure 5.15a (Line 8), signatures are not unique any more, even within a single class declaration. The code in Figure 5.15a contains two implementations of method m. The first method implementation (Line 4) is present only if the options Opt1 and Opt2 are enabled. The second method implementation (Lines 6–10) is always present, but has two alternative variants: If Opt1 is enabled, an optional parameter is present; if Opt1 is disabled, the parameter is missing. So, in the latter case, the second method definition has the same signature as the first method definition. All configurations of the configurable program are well typed, and all variants of the method are used in some configurations, so we must include both methods in the variant simulator. However, as the methods have the same signature, we have a signature conflict if we just copy them to the simulator. To solve this signature conflict, we *duplicate* the code of the second method and insert it into the first method (Figure 5.15b). The code duplicates are guarded with corresponding presence conditions. Although this solution leads to code clones, we avoid the complex task of identifying and modifying all call sites of the methods.

Optional program variables We do not support optional fields in our model. However, program variables (fields and local variables) may be optional in real-world applications.

Figure 5.16a shows a program that contains a struct with one or two integer variables, depending on the configuration. We need to include both variables in the variant simulator, because both are used in, at least, one configuration. The program also contains a function that returns the size of this struct. If Opt1 is not enabled, the function returns the size of one integer; if Opt1 is enabled, it returns the size of two integers. However, as we have to include both integers in the variant simulator, the function always returns the size of two integers and, therefore the behavior of the variants is not preserved.

There are two possible solutions of this problem. One solution (shown in Figure 5.16b) is to rename and duplicate the struct definition. In this solution, we also have to modify references to the struct and add feature choices, such that the use always refers to the correct struct implementation. For more complex instances of the problem, this leads to an exponential code blowup and complex variant simulators. An alternative solution is to transform all expressions that are affected by system functions that cannot be changed (such as sizeof). The transformed sizeof expression for our example would be (Opt1 ? 2∗sizeof(**int**) : sizeof(**int**)). In scenarios with more variables, the transformed expression grows exponentially with the number of variables, similar to the previous solution. Both presented solutions (struct duplication and transformation of struct usages) have limitations in practice and have to be applied depending on the case at hand. The same problem occurs with direct memory access in C via pointer arithmetic and with reflection in JAVA. In both cases, either a case-specific solution has to be found or code has to be duplicated.

Alternative types We do not allow static variability of return types of functions or types of variables or parameters. However, this is possible in languages such as JAVA or C, with ifdef directives. We can, for example, declare a variable with alternative types, such as **#if** (Opt1) **int #else double #endif** x;. The variable x is either of type int or of type double. All valid variants of this configurable program are well typed [AKGL10].

When building a variant simulator for this example, we cannot statically determine the type of the variable. As the different types may not have a common supertype, we have to include both possibilities in the variant simulator. Therefore, we have to duplicate the variable declaration. Each location at which the variable is used must be modified such that the used variable variant depends on the selected configuration option. In extreme cases, if each variant has a different type for the variable, we have to introduce a variable for each configuration. However, according to our experience, this situation is very rare. A similar problem occurs if method signatures are variable (e.g., variability in method modifiers). Further similar examples are struct or enum definitions in C, generics, annotations, exceptions in method declarations, and class declarations (e.g., variable inheritance with alternative extends clauses) in JAVA.

```
int m() {
    int x = 0;
    for (int i = 0; i < 10; i++) {
      #if Opt1
        int x = 1;
      #endif
      x++;
    }
    return x;
}
```

(a) Problem: Field shadowing

```
int m() {
    int x = 0;
    for (int i = 0; i < 10; i++) {
        int y = 1;
        (Opt1 ? y++ : x++);
    }
    return x;
}
```

(b) Solution: Variable renaming

Figure 5.17: Optional field shadowing in C

Field shadowing A further problem with optional program variables is shadowing [TSAH12]. As an example, assume that we define two local variables in different scopes with the same name, but one of the variables is optional depending on configuration option Opt1 (Figure 5.17a). Consequently, Opt1 influences to which program variable an identifier in the inner scope refers to. In our example, method m either returns 0 or 11. Hence, optional shadowing needs to be handled in variability encoding; one solution is to rename one of the program variables and duplicate all statements that contain the identifier (see Figure 5.17b). Shadowing may also co-occur with other language constructs, such as inner classes. Instead of renaming, we can consider all these cases (other than variable shadowing) as code smells and suggest to forbid them, because code with optional shadowing may be hard-to-understand and may cause faults in configurable programs anyway.

Concurrency In our model of variability encoding, we ensure that each statement and therefore each access to potentially shared data of the configurable program is either guarded with presence conditions or duplicated. There are two problems that occur in variability encoding of concurrent programs: (1) code executed during class initialization can sometimes not be enclosed in if statements (e.g., field initializations) and (2) feature-choice statements might slow down threads with much variable code more than others. To handle the first problem, we move code that initializes optional data structures (e.g., fields) to constructors and guard them with presence conditions there. This way, only code of one variant (and feature choices) is executed and other variants cannot interfere. The configuration of all feature variables is selected and fixed from the beginning of the execution of the variant simulator. All threads are executed with the same configuration and execute the behavior of the variant, including possible interactions between the threads. These interactions also include synchronizations on shared variables. The second problem can be solved by using an execution engine that explores all possible thread interleavings such as a model checker for concurrent programs. If multiple variants in a variant simulator with concurrency are analyzed at the same time (e.g., with a model checker), the analysis tool has to ensure that program states from different variants do not get mixed up and produce formerly unreachable states. Other common pitfalls of concurrent programs, such as breaking atomicity, shared mutable states, transactions, livelocks, and deadlocks, can be dismissed because a thread execution path in a simulator accesses the same variables (in the same order) as the corresponding path in the variant plus immutable feature variables.

Non-functional properties Concerning non-functional properties (e.g., performance, memory consumption, or response time), a variant simulator behaves differently than a variant. The implementation of variability encoding ensures that the variant simulator executes statements that would be executed in the variant and additionally executes guard statements. Therefore, in theory, the only difference in executed statements should be the evaluation of presence conditions and the loading of classes which do not occur in the variant but are necessary in the simulator. Code for the instantiation of such classes can also be guarded. Depending on the system and its granularity, there may be many feature choices and the evaluation of presence conditions may also be expensive. The code overhead of variant simulators naturally influences non-functional properties such as binary size and time for program setup. To accurately predict non-functional properties of a variant based on the performance of the variant simulator [SvRA13], the prediction technique must take this overhead into account.

5.5 Experience with Variability Encoding

In this section, we give an overview of our work in applying variability encoding in analyses of configurable systems. We also summarize our experience with implementing variability encoding for real languages and systems. Our formal proof of behavior preservation (Section 5.3) strengthens the results of these projects. It increases confidence that the implementations of variability encoding used in these projects also preserve behavior (even though the implementations are much more complex).

5.5.1 Variability Encoding in Java

Variability encoding has been used for testing, verification, and performance modelling of configurable Java programs.

Testing For testing of configurable systems, Kästner et al. [KvRE$^+$12] built an interpreter for Java-based systems with ifdef variability on top of the tool TypeChef [KGR$^+$11]. The idea is to parse the ifdef variability as if it was run-time variability (similar to variant simulators). Then, the program is executed with a fixed test input, but without fixing the feature variables; once a feature choice is reached, the interpreter executes both branches. The interpreter aims at visiting execution paths as few times as possible, still covering the executions of all variants, which can reduce testing time substantially.

Bürdek et al. [BLB$^+$15] used variability encoding to efficiently generate test suites for configurable systems. They used a variability-aware extension of the test-case generator CPA/Tiger [BHTV13] on variant simulators that have been generated with FeatureHouse [AKL09]. Given a set of test goals that should be covered (e.g., statement coverage or branch coverage), the approach generates a *variability-aware* test suite. The test suite contains a set of test cases which each have a presence condition stating on which variants they can be run. Furthermore, for each combination of test cases and test goals, the test suite has a presence condition stating in which variants the test case covers the test goal. Based on this information, one can generate a test plan that contains a minimal set of test cases and variants on which they need to be executed to cover all test goals in all variants. The evaluation shows that variability-aware test case generation generates far less test cases than a variant-based approach.

Verification We used variability encoding for model checking configurable systems which we discuss in Chapter 6 [ASW$^+$11; AvRW$^+$13]. For the experiments, we selected three configurable Java programs that served as benchmarks for the community before and that provide functional specifications violated by some variants. To identify the violations, we constructed a variant simulator per configurable program using FeatureHouse [AKL13], and we verified the variant simulator using the software model checker Java Pathfinder [VHB$^+$03]. The experiments showed that analyzing the variant simulator is substantially faster than checking all variants individually [AvRW$^+$13]. Also, analyzing the variant simulator was as precise for correctness checking as the analysis of all variants.

In a work on deductive verification of configurable programs, Thüm et al. [TMB$^+$14; TSAH12] applied variability encoding not only to the configurable Java code, but also to the corresponding specifications, which have been written in an extension of the Java Modelling Language (JML). Much like for programs, a configurable JML specification may give rise to different variants, depending on the configuration, which needs to be taken into account during verification. Technically, this approach transforms a configurable JML specification to a corresponding *specification simulator* (a.k.a. meta-specification [TSAH12]), which is similar to creating a variant simulator for a configurable Java program. Verifying the variant simulator using the theorem prover KeY, the authors observed considerable speedups compared to the verification of all variants, and the resulting proofs for the variant simulator have a similar complexity as the proofs for variants [TSAH12].

An interesting property of variant simulators is that existing verifiers for run-time variability can be reused as-is for compile-time variability. This property was exploited by verifying compile-time configurable programs with the theorem prover KeY [TMB+14] and the software model checker Java Pathfinder [AvRW+13] (Chapter 6). The results showed that the combination improves efficiency and effectiveness at the same time [AvRW+13; TMB+14].

Performance modelling Siegmund et al. [SvRA13] used variability encoding to quantify the effects of individual features on the run time of the variants of a configurable program. Using FeatureHouse, they constructed variant simulators for a set of configurable Java programs, with (non-variable) test cases. Based on these tests, they executed the variant simulators with a normal Java run-time environment, and they measured how much time was spent in each method and in which context the method was called. Using this information, they built a performance model per configurable program that allowed them to estimate the performance contribution of individual features and feature interactions, without creating and measuring each program variant individually.

Summary In all of these approaches, variability encoding plays a central role in reducing analysis or measurement effort. As in all cases variability has been implemented with feature modules (cf. Section 2.1.3), most of the variability stems from method refinements resulting in alternative method bodies, which are easy to encode in variant simulators. Notably, none of the problems discussed in Section 5.4 occurred in the respective case studies, which suggests that coarse-grained variability mechanisms, such as feature modules, sufficiently facilitate variability encoding.

5.5.2 Variability Encoding with C

In a parallel line of research, we developed the tool Hercules, an extension of TypeChef with functionality for variability encoding of large-scale configurable C programs that use the C preprocessor to implement compile-time variability. In the motivating section of this chapter, we describe a practical application scenario of Hercules (Section 5.1.1). We used Hercules to generate variant simulators for code slices of the Linux kernel and used a model-checking tool to verify these simulators. In this section, we present (1) a short description of variability encoding in Hercules, (2) an evaluation of the code size of variant simulators compared to ifdef code (overhead), and (3) an evaluation of behavior preservation of Hercules based on the configurable system SQLite.

Variability encoding in Hercules The process of constructing variant simulators for C programs with preprocessor directives is much more challenging than for feature modules in Java, because of very fine grained variability.[7] Preprocessor-induced variability in C programs occurs frequently at the level of field declarations and type definitions [LAL^{+}10; LKA11]. At this granularity, dynamic variability cannot be directly encoded in C (if statements are allowed only in method bodies). Therefore, one has to duplicate field declarations and type definitions and embed expressions that use them into conditional statements (i.e., feature choices), as we have illustrated in various examples in Section 5.4. As a further challenge, C does not support method overloading, so a variant simulator for a C program, similar to the example in Figure 5.15a on page 108, requires more code duplication to represent all configurations !Opt1, Opt1 && !Opt2, and Opt1 && Opt2. As a result, one has to introduce new method names for each duplicate and modify all calls to the method correspondingly. Furthermore, we identified several language features of C that required special attention in our implementation:

- **Type declarations**. Since the C preprocessor is often used to implement portable code (e.g., for different hardware architectures or operating systems), ifdef directives often implement variability at the level of type definitions (e.g., choosing between types u32 and u64). Similar to the handling of alternative method types (Section 5.4), one has to create multiple variants of type definitions that are included in different configurations.
- **Jump labels and goto statements**. C code allows to annotate every line in a method with a jump label and jump to this label using goto statements. Both, labels and goto statements, can be enclosed in ifdef directives. In particular, labels that are defined in multiple positions

[7] Our Java subject systems are implemented with superimposition [AL08] in FeatureHouse [AKL09], where variability is usually at function level (coarse-grained).

```
char params[] =
    "-a\0" + "-o\0"
#if (FIND_NOT)
"!\0"
#endif
#if (DESKTOP)
"-and\0"+ "-or\0"
#if (FIND_NOT)
"-not\0"
#endif
#endif
"-print\0"
#if (FIND_PRINT0)
"-print0\0"
#endif
... // 49 additional lines of
... // code with string literals
;
```

Figure 5.18: Exponential explosion in variability encoding of Busybox

in alternative configurations are problematic because we cannot include more than one alternative in the simulator (label identifiers must be unique). We encode such labels by renaming them and by inserting feature switches at the corresponding goto statements.

- **Switch statements**. Large C programs such as SQLite often include methods for parsing user input. Such parsers are often implemented with switch statements where each case corresponds to a command or token. Often entire cases are optional, depending on whether the user selects support for a functionality. If the functionality is not chosen these cases are handled by a default case. A Hercules-generated variant simulator must provide both possible control flows and ensure that always the correct control flow is executed. In addition to this situation, (optional) break and continue statements can further increase complexity of the control flow. We realized variability encoding of such situations by inserting jump labels and goto statements in the variant simulator.

Summarized, we often used code duplication and renaming of identifiers in Hercules to solve variability-encoding problems. However, we also encountered extreme patterns with very many variants that require manual intervention. We would like to illustrate one extreme variability pattern that we encountered in systems such as the Linux kernel, SQLite and the Busybox tool suite [LvRK+13]. In Figure 5.18, we show a variable declaration (taken from Busybox) storing a string for the evaluation of command-line parameters.

	Before Variability Encoding	Overhead
Typedefs	128 617	0
Structs/Unions	85 096	612
Enums	24 891	0
Global Variables	29 614	101
Methods	14 825	548
Forward Declarations	726 115	2 120

Figure 5.19: Statistics of variability encoding in Busybox; program elements before variability encoding and additional elements (overhead) introduced by duplications; summarized over all files

The string itself is variable, as its substrings are annotated with 21 features. In the worst case, this implementation requires the generation of 2^{21} different string variants to be included in the corresponding variant simulator, which is infeasible in practice. The only way to solve this problem is to compute the desired string variant at run or load time using a manually implemented function that resembles the compile-time computation for generating the string variants. However, this extreme pattern occurs infrequently compared to other patterns that can be handled without duplication by variability encoding.

Overhead introduced by Hercules We conducted an experiment to evaluate how often patterns that require code duplication occur, and how much code overhead Hercules generates. The results of this experiment are shown in Figure 5.19. Overall, we found that variability encoding is feasible for real-world C programs. The relatively low amount of code duplication indicates the feasibility of the overall approach, because a variant simulator without sharing cannot be analyzed faster than analyzing all of its variants. We measured how often code duplication is necessary when building variant simulators for Busybox. In particular, we measured the number of program elements before variability encoding and how many additional program elements (overhead) arise from expressing this variability in the variant simulator, as shown in Figure 5.19. The statistic represents 518 C files and the included header files.[8] 5 of the 518 transformed files still contain patterns of extreme variability, such as the one shown in Figure 5.18 which need to be handled manually. As a key result, we found that the number of additional elements that we need to generate in code duplications is always below 4%. Hence, the generated overhead in terms of source-code expansion from variability encoding is negligible.

[8] We skipped 1 C-file due to type errors in the transformation result.

Behavior preservation of Hercules Hercules is a complex, evolving program that is based on TypeChef, which is also a complex and steadily evolving system. It would be ineffective to attempt a behavior-preservation proof for Hercules. Instead, we conducted an experiment with a real-world system, showing that Hercules preserves variant behavior in most cases.

We used Hercules to generate variant simulators for SQLite and set the feature variables, such that one specific variant is simulated (each feature variable is set to 0 or 1). Then, we tested the configured simulator and the corresponding variant using the test suite TH3[9] for SQLite. If the test results are equal, we assume that Hercules correctly encoded the variability of the variant.

hypothesis

Our hypothesis is that Hercules can encode the variability of SQLite and its test suite TH3 such that testing the generated simulator (with a configuration Φ) yields the same result as testing the variant built with configuration Φ.

subject system

For testing, we focused on variability induced by 23 configuration options. We disabled options, such as SQLITE_OMIT_SUBQUERY and SQLITE_OMIT_VIEW, which disable functionality that is required by tests in the TH3 test suite. We generated configurations for feature-wise and pair-wise coverage of these options (23 and 11 configurations, respectively).

In TH3, a test case is generated from a folder of *test specifications* and a *test configuration* (not to be confused with the configuration of ifdef options; test configurations set, for example, cache sizes). For our experiment, we excluded (1) stress tests, which would not have scaled due to the number of variants that we test, (2) a test that prints the current system time, which makes result comparison impossible, and (3) a test that stores its test script in a very large struct with 100 ifdefs and 33 554 432 variants. Furthermore, we excluded a test configuration that focuses on filesystems that only support short filenames.[10] We selected 12 folders with test specifications and 25 test configurations from the TH3 test suite. From each folder and each test configuration, we generated a test case. Each test case sequentially executes several test specifications. We ran each test case for each feature-wise and pair-wise configuration in a simulator and in the corresponding variant and compared the results. In total, these are 6900 comparisons for the feature-wise configurations and 3300 comparisons for the pair-wise configurations. In each comparison, we checked whether the test result in the variant is equal to the result in the simulator.

manual preparation

We encountered one pattern of extreme variability in the SQLite source code that we had to encode manually. SQLite has a global array azCompileOpt

[9] `https://www.sqlite.org/th3/`

[10] The *8.3 filename scheme* was enforced in old versions of DOS whereas we use UNIX for our experiments.

Table 5.1: Comparison of TH3 test results for variability encoding on SQLite

Test result			Number of tests	
			Feature wise	Pair wise
Same test results	Full coverage	No failed tests	4629	1662
		Failed tests	1368	819
	Partial coverage	No failed tests	56	9
		Failed tests	84	209
Different test results	Full coverage		161	273
	Partial coverage		602	326
Timeout			0	2

that stores the names of all enabled configuration options (very similar to the pattern shown in Figure 5.18). The contents of the array are determined at compile time using 105 ifdef statements, leading to 67 108 864 variants of this declaration, which makes duplication infeasible. Furthermore, we cannot directly encode these ifdefs because if statements are not allowed in array declarations. Therefore, we manually moved the array initialization to a new method that is called at the beginning of the main function. We also checked that the array is not referenced in other initialization statements, which would be executed before the main function. We provide the setup and results of this experiment on the supplementary website, however, we can not include the TH3 test suite due to its proprietary license.

We summarize the results of our experiment in Table 5.1 and in Figure 5.20. The table shows that the variant-simulators and variants generate the same test result in 88 (feature wise) and 81 (pair wise) percent of all test cases. This result indicates that, for most test cases, we correctly encode variant behavior, which confirms our hypothesis.

Some of the test cases terminated with segmentation faults or other irregular test terminations. Such faults occur, for example, if a test specification, that is part of a test case, tries to execute code that is not included in a system variant. If a test terminates with a segmentation fault, we cannot execute test specifications that would be executed after the fault, so our test does not cover the entire set of specifications in the test case. However, we argue that this is a problem in the TH3 test suite which only partially documents dependencies of test specifications on configuration options. We indicate tests that did not execute all test specifications with “partial coverage” in Table 5.1 and Figure 5.20.

Furthermore, there are tests which generate different results in a variant and a simulator. We manually inspected several of these tests and we found several reasons for different behavior:

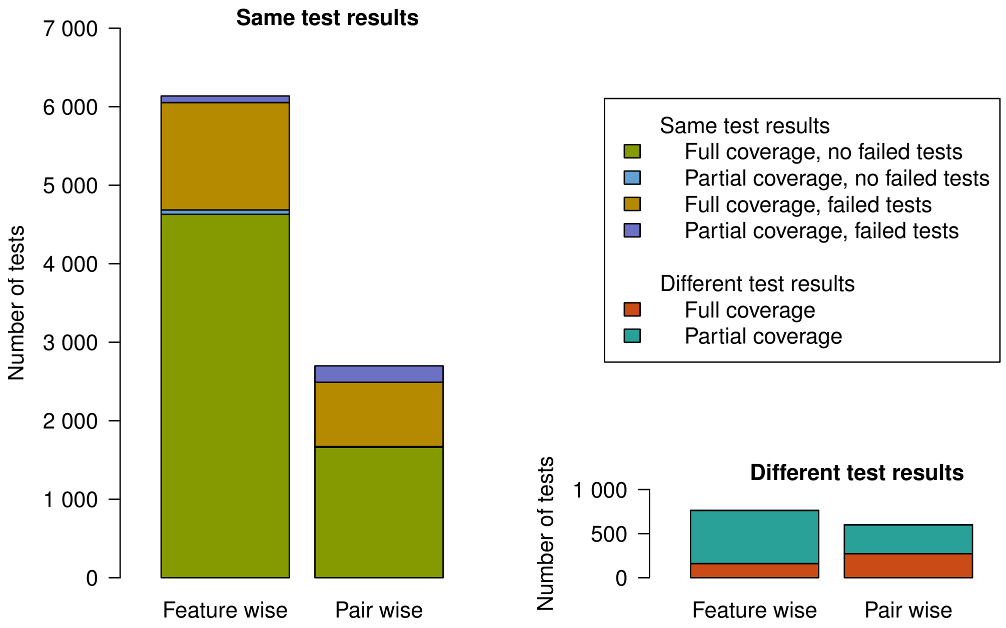


Figure 5.20: Results of the comparison of TH3 tests on variants and simulators. Timeouts do not show because of the scaling.

- Test cases sometimes print execution times, memory sizes or even memory addresses in error messages. Such values are very likely different in different executions of the same program. Therefore we cannot reproduce such behavior with a simulator and the variant behavior is different from the simulator behavior. We identified several statements in TH3 and SQLite where such values are printed and replaced them with default strings. This solution fixed some cases with different results, but for some cases we could not identify the responsible statements.
- Some test cases seem to have non-deterministic behavior. We fixed the order in which test specifications are executed in a test case, but if a test, for example, spawns parallel threads different behavior could be generated. In such cases, we cannot show correctness of simulators with testing.
- We identified undocumented dependencies of test cases on configuration options. For example, some tests used CAST expressions to convert values between different data types. One of the configuration options, SQLITE_OMIT_CAST disables support for such expressions and consequently these tests fail. For this reason, we excluded the configuration option SQLITE_OMIT_CAST from our experiment, but we cannot rule

out that other similar errors exist in our experiment setup. The undocumented dependencies that we identified suggest that TH3 was not tested with a large number of different configurations, which increases the value of our experiment and explains some results.

Threats to validity Our choice of sample configurations and test cases threatens the internal validity of our experiment results. We covered only a very small fraction of the possible SQLite configurations, and we excluded test cases such as the stress tests. However our evaluation covers a large part of the code of SQLite and a very diverse and realistic set of problems to variability encoding.

We used testing as an approximate means to compare the behavior of variants and simulators. Of course, this approach cannot prove behavior preservation, which threatens our internal validity. However a formal proof would require more expensive approaches (e.g., model checking) to explore and compare all possible (perhaps non-deterministic) execution paths of variants and simulators. Our results show that variability encoding generates a simulator that, in the vast majority of cases, generates the same test results as the corresponding variant.

We discovered and corrected several defects in our experiment setup, such as hidden dependencies between test cases and configuration options. We cannot rule out that similar bugs still exists, and these may affect our results. Correcting some of these bugs would mean complex modification of SQLite source code, which, for itself, would threaten internal validity.

Our experiment is based on one configurable system and it is not clear whether Hercules would generate similar results for other systems. However, SQLite is a real-world database engine that is used in many production systems and we used the official TH3 test suite for SQLite. Furthermore, this is the first experiment that evaluates the correctness of variability encoding on a large-scale configurable system.

5.6 Related Work

The importance of formal foundations for configurable programs has been recognized before [Hey12; SH11; TAK+14]. There are several publications that define and refine formal definitions of systems with variability [AtBFG10; DKB14; FG07; GLS08; Kap12] and also prove behavioral equivalence [FUB06; Loc12] between different configurable program representations. However, to the best of our knowledge, we are the first to define variability encoding based on a canonical representation of the syntax and semantics of the configurable program, and not starting from abstract representations, such as transition systems. This is an important aspect because, in practice, configurable programs are not implemented with transition systems or state machines, but in programming languages, such as C using ifdef directives. Therefore, it is important to investigate the effect of different language constructs on the correctness of variability encoding.

There are several publications in which formal representations for configurable programs are defined and discussed. Gruler et al. [GLS08] propose PL-CCS based on the process algebra CALCULUS OF COMMUNICATING SYSTEMS (CCS). Kapus [Kap12] proposes the language TLA+ based on featured transition systems [CCS$^+$13]. Fischbein et al. [FUB06], Asirelli et al. [AtBFG10], and Fantechi et al. [FG07] discuss formal languages based on modal transition systems. These formalisms (especially the modal transition systems) are similar to the transition systems we used in Section 5.3.1. Fischbein et al. [FUB06] also use a variant of bisimulation to prove that their modal transition system correctly models the behavior of the system variants. In contrast to our work, all these formalisms rely on a graph-based representation of configurable programs, and they do not discuss programming language constructs.

Gnesi and Petrocchi [GP12] define the Controlled Language for Software Product Lines (CL4SPL). It is designed to be used by engineers of configurable programs and to be translated to executable languages for automated verification. This work is similar to ours in that they offer a language in which developers can express variability and verify that the implementation satisfies its specifications. However, this language is quite far from common programming languages and no proof for the correctness of the transformation is given.

Erwig and Walkingshaw propose the *choice calculus* [EW11] as fundamental representation of software variation, which has been extended and used for various scenarios such as type inference [CEW12]. In our work, we depend on formal rules that model the behavior of JAVA. Unlike CFJ, the choice calculus is not bound to any mainstream language, and therefore does not encode variability in the host language, as we do.

Midtgaard et al. [MBW14] developed a formal framework to derive variability-aware static analyses from standard static analyses. They prove that the variability-aware part of analyses adapted with their framework is correct by construction. The overall goal of their research is the same as ours, but our proof for variability encoding enables the reuse of existing analyses without further adaption. A similar approach is proposed in SPL$^{\text{LIFT}}$ [BTR$^+$13], a tool for automatic transformation of analyses that are phrased in the IFDS framework [RHS95]. However, IFDS supports only a certain class of data-flow analyses and other static analyses such as type checking and model checking cannot be expressed as data-flow analyses.

There are several other publications [AKGL10; BDS13; DCB09] in which researchers define frameworks for compile-time variability and variant generation based on JAVA. They formally define syntax and typing rules as well as how variants are derived from a configurable program. They also show that the derivation process preserves type correctness. However, they do not use variability encoding or other load-time variability mechanisms. As type correctness is a requirement for behavioral analysis, we see this work on type checking and others (e.g., [KATS12]) as premise to variability encoding.

Classen et al. [CHSL11] and Post and Sinz [PS08] used approaches that are similar to variability encoding. Classen et al. developed a verification engine based on *featured transition systems* (FTS), which represent control-flow graphs with presence conditions on edges. As input language, they use the language fSMV [PR01], which is based on PROMELA, and encodes variability with the conditional language constructs of PROMELA. fSMV programs are automatically loaded as FTS and verified. Post and Sinz *manually* encoded variable statements of LINUX device drivers with guarding conditional statements and verified the resulting variant simulator. Both approaches rely on manual encoding of variability; correctness proofs are not available.

Beuche and Weiland [BW09] Matlab-Simulink models with support for different binding times. Users can move configuration options from one binding time (e.g., compile time) to another one (e.g., run time) and the model is automatically modified accordingly. Their tool chain is used in a project for production code development of Mercedes passenger cars, which emphasizes the practical relevance of binding-time transformation. In difference to this project, we only support transformation from compile-time configuration options to run-time configuration options. However, our approach focuses on common mainstream programming languages, which enables a broad set of application scenarios such as software model checking of configurable systems (Chapter 6).

CHAPTER 6

Family-based Model Checking

This chapter shares material with the following publications:

- "Domain Types: Abstract-Domain Selection Based on Variable Usage" in *HVC'2013* [ABF$^+$13],
- "Strategies for Product-Line Verification: Case Studies and Experiments" in *ICSE'2013* [AvRW$^+$13], and
- "Detection of Feature Interactions using Feature-Aware Verification" in *ASE'2011* [ASW$^+$11].

In this chapter we present our approach of efficient, family-based model checking of configurable systems. Our approach relies on a modular, two-step process: First, we use variability encoding (Chapter 5) to generate variant simulators representing all behavior of all variants of analyzed systems. Because we use standard language concepts to encode variability in simulators (conditional statements), standard tools can be used to analyze simulators. In the second step, we use a model-checking tool to verify that the generated simulators satisfy all specifications of the respective systems. If the model checker can prove this property, all variants satisfy their specifications, too (given that variability encoding preserves variant behavior, Chapter 5).

For model checking of simulators, we can use off-the-shelf model-checking tools, which will generate valid verdicts given enough time and memory. However, usually efficiency can be improved with optimizations for configurable-system analysis. We call such optimizations *variability-aware* because they use knowledge about how variability is encoded in simulators. Variability-aware optimizations make off-the-shelf tools "aware" of the static variability

variability-aware model checking

```
1  int x,y;
2  #if A
3    x=1;
4  #else
5    x=0;
6  #endif
7  #if B
8    y=1;
9  #else
10   y=0;
11 #endif
12 y=10/(x+y);
13 ...
```

(a) Ifdef example code

```
boolean A=nondet();
boolean B=nondet();
int x,y;
if (A) {
  x=1;
} else {
  x=0;
}
if (B) {
  y=1;
} else {
  y=0;
}
y=10/(x+y);
...
```

(b) Variability-encoded example code

Figure 6.1: Running example to describe model checking

encoded in the simulators. Both model-checking approaches (with and without variability-aware optimization) are *family-based* because they analyze one simulator instead of all variants separately.

In this chapter we describe how model-checking tools can be made variability-aware. We extended two concrete model checkers: Java Pathfinder [VHB+03] (extension jpf-bdd [vRAR11]) and CPAchecker [BK11]. Both tools allow relatively clean, focused extensions using their plugin-oriented architectures.

running example

Figure 6.1 shows our running example for the description of the model checking extensions. The program is written with ifdef directives and has different behavior depending on the feature selection. It has two configuration options (A and B) and four variants. The variant in which both options are disabled contains a division-by-zero error: x and y are set to zero in lines 5 and 10 and therefore the divisor of the division in line 12 is zero (lines referring to Figure 6.1a). Figure 6.1b shows a simulator for the example generated with variability encoding. In the simulator, the feature variables are initialized to an *unknown* value by calling the nondet function; we discuss this in Section 6.1. We use the running example to illustrate our descriptions of explicit-state model checking and of our extensions. However, application of the extensions is of course not limited to the running example.

chapter structure

We begin with illustrating explicit-state model checking of variant simulators and the problems occurring in this approach (Section 6.1). As explicit-state model checking is a well-established technique, Section 6.1 does not contain novel technical contributions. Based on the problems identified during the explicit-state section, we introduce variability-aware model checking in Section 6.2. In Section 6.4, we describe an evaluation of the performance of family-based model checking with and without our variability-aware extensions. In Section 6.5, we discuss an evaluation of family-based model checking in comparison to variant-based and sample-based model checking strategies. In Section 6.6, we present a set of strategies that combine family-based and variant-based verification. We discuss an evaluation of these strategies and show that they improve over the basic family-based and variant-based strategies.

6.1 Explicit-State Model Checking of Variant Simulators

Among the various model checking approaches, explicit-state model checking is the simplest approach [CGP99]. Therefore it is suited well as basis for the description of our extension. As our extension is modular, it can relatively easily be applied to other model checking approaches such as symbolic model checking.

explicit-state model checking

When verifying a program, a software model checker constructs a reachability graph in which each node represents a concrete state (or a set of concrete states[1]) that the program can reach during execution. In *explicit-state model checking*, the concrete values of variables are stored in the states. Typically, each state contains a map from variable names to variable values. Figure 6.2 shows a reachability graph for the variant simulator of the running example. Each node in the figure maps the five variables *pcounter*, A, B, x, and y to values. The variable *pcounter* represents the *program counter*, which represents the instruction in the program that is executed in the next evaluation step. For simplicity, the values of the *pcounter* variable refer to the line numbers in the variability-encoded program (Figure 6.1b). Furthermore, A and B are Boolean variables, mapped to values T for *true* and F for *false*. The value **?** is the *unknown* value. It denotes that the current state does not contain information on the value of a variable. We initialize the feature variables A and B with **?**, to allow the model checker to explore all variants of the program.

state-space exploration

To verify a program, a software model checker explores its state space. Algorithm 6.1 shows a basic breath-first algorithm for building and exploration of such a state space. The algorithm is based on the *reached_set*, which stores the states that were already visited, and the *wait_queue*, which stores

[1] A reachability graph in which nodes can represent more than one concrete state is *abstract*.

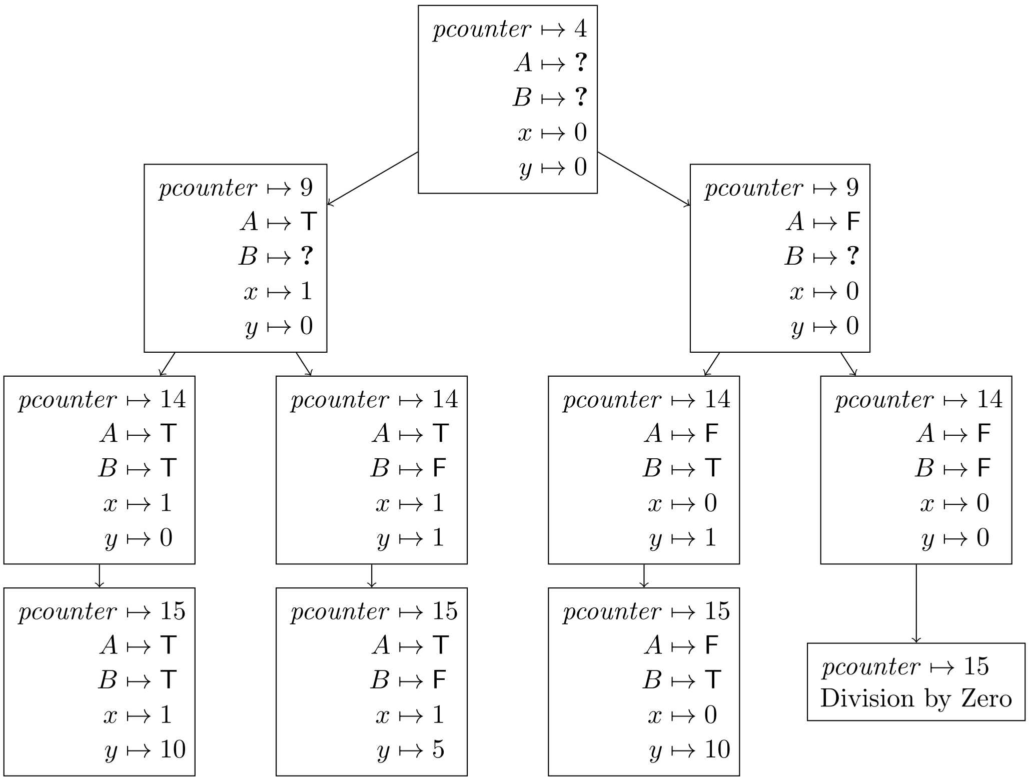


Figure 6.2: Reachability graph of the example variant simulator. T and F represent the Boolean values for *true* and *false*.

the states that need to be explored further. The exploration starts with an initial state q_0 that corresponds to the first statement of the program (e.g., initialization or main function). The algorithm adds q_0 to the *wait_queue* and to the *reached_set*. Then, it takes states from the *wait_queue* until it is empty. For each state v, it checks whether v violates a specification given for the system (Line 5) and then computes all successor states of v (based on a CFG of the system, which is not shown in Algorithm 6.1). Each successor state that has not been reached before is then added to the *wait_queue*. When all states from the *wait_queue* have been explored, the algorithm returns the *reached_set*, representing the state space of the verified system.

Next, we discuss how the reachability graph in Figure 6.2 is constructed using Algorithm 6.1. In the initial state shown in Figure 6.2, the program counter refers to Line 4 of the program, which is a feature choice on variable A. Because A is unknown in the state, two different following states are possible (Line 6 in Algorithm 6.1) and the exploration splits. To reduce the size of the

Algorithm 6.1: Basic breadth-first model-checking algorithm. Function hasNoPropertyViolations(v) returns *true* if a system specification is violated in state v and function direct_succ(v) computes the states that are reachable from v in one step.

```
Data: initial state $q_0$
Result: Set $reached\_states$
add($reached\_states$, $q_0$)
enqueue($wait\_queue$, $q_0$)
while $\neg$ empty($wait\_queue$) do
    let $v$ := dequeue($wait\_queue$)
    if hasNoPropertyViolations($v$) then
        foreach $w \in$ direct_succ($v$) do
            if $w \notin reached\_states$ then
                add($reached\_states$, $w$)
                enqueue($wait\_queue$, $w$)
            end
        end
end
return $reached\_states$
```

graph, we omit some nodes which have non-divergent behavior.[2] The next path divergence happens in the next step on variable B. At the end of this small exploration, the frontier of the reachability graph[3] has four states. One of these states has a division-by-zero error. We can reconstruct how to reproduce this error in the simulator by examining the values of the variables in the error state, where both feature variables (A and B) are mapped to F. Therefore, configuring the simulator with both feature variables set to F instead of **?** will lead directly to this error. Because of the construction of the simulator (Chapter 5), we can infer that the same error would occur in the variant where features A and B are disabled.

state-merging prevented by feature variables

While this example illustrates how variant simulators can be analyzed with standard model-checking approaches, the analysis is not efficient. The performance (execution time and memory consumption) of a model checking approach is directly influenced by the size of the generated reachability graph [CGP99]. The example reachability graph shows that a new sub-graph is generated after each choice of a previously *unknown* variable. Typically, model-checking tools

[2] For example, lines 1 to 3 contain variable initializers. Their effect is subsumed by the initial state in the reachability graph.

[3] The frontier of the reachability graph is the set of nodes that need to be explored further (stored in the *wait_queue* in Algorithm 6.1). It is the frontier of the explored region of the state space. In our example graphs, the lowermost states form the frontier.

try to reduce the size of the state space by merging states that represent the same variable values.[4] When exploring the state space of a simulator, the value maps for states in different sub-graphs are always different in their assignments of the feature variables. As an example, consider the feature choice on variable A in Figure 6.2. After the feature choice, the variable is mapped to T in the left sub-graph and to F in the right sub-graph. States from the sub-graphs can never be merged because of these different values for A and because the value of A is never changed in subsequent states. If all feature variables are used on all exploration paths, the reachability graph has at least as many leaves (termination states) as the configurable system has variants. This is a large problem for scalability of simulator-based verification.

6.2 Variability-aware Model Checking of Variant Simulators

We introduce a variability-aware extension of model checking that focuses on optimizing the state-space exploration of variant simulators. Our extension uses *symbolic* model-checking [JM09] for the compact representation of variablity during verification. In particular, we use BDDs to represent variability, which has already been done in previous work on configurable systems [CHSL11] (but not on JAVA/C source code) and in previous work on verification of non-configurable software (e.g., [BS13]).

The size of state spaces has been long recognized as a main problem of model checking, and many optimizations have been developed to reduce the set of states that must be explored. Example optimizations include partial-order reduction, symbolic representations, or counter-example–guided abstraction refinement [CGJ$^+$03; CGP99]. Our optimizations have the same goal (e.g., reducing state space) and use in part the same methods (e.g., abstraction and symbolic representation), however, our optimizations are targeted to the specifics of configurable-systems. Such systems introduce additional complexity in the form of static variability (or feature variables in simulators) and we develop optimizations to deal with this complexity.

properties of variant simulators

When verifying simulators we have the advantage of knowing how simulators are constructed. The feature variables in simulators are constrained in several ways: First, they are always Boolean variables, second, they are only used in conditions of if-statements, third, they are only read and never written (except for the *unknown* initialization), and fourth, they are never used in combination

[4] Other model checking techniques also allow to merge states even if they are not identical (e.g., if an abstract state subsumes all concrete states represented by another abstract state).

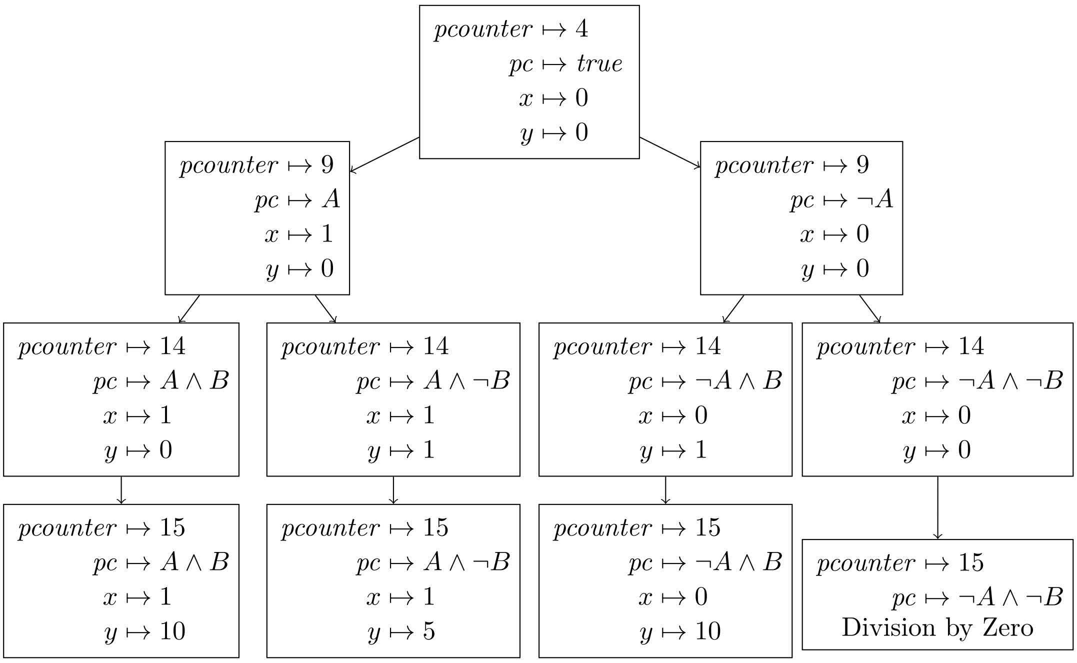


Figure 6.3: Reachability graph of the example program with presence conditions. In each state, *pc* denotes the presence condition of the state.

with non-feature-variables. This narrow usage profile of feature variables enables us to develop model-checking extensions that focus on optimizing the representation of feature variables.

extension structure

To introduce the variability-aware extension for simulator-based model checking, we introduce four successive optimizations. First, we use *presence conditions* to represent values of feature variables (Section 6.2.1). Second, we represent the *presence conditions* in a symbolic component of each state of the reachability graph to allow state merging (Section 6.2.2). Third, in Section 6.2.3, we discuss an optional extension that allows us to exclude variants from the search process (e.g., if we have already found defects for a variant). Fourth, we describe how we include the variability model of a subject system in the search process to avoid exploring states that do not belong to valid system variants (Section 6.2.4). We describe the optimizations one after another and argue for each why it does not compromise the correctness of the model checking result.

6.2.1 Presence Conditions

The special properties of feature variables (e.g., boolean and read-only) distinguish them from other program variables. These properties allow us to group the feature variables and handle them with a different, more efficient data structure than normal program variables. We choose BDDs as data structure because they allow for very efficient handling of functions over Boolean variables. Figure 6.3 shows how we move the handling of feature variables from the explicit map to a predicate-based data structure (also called symbolic representation). We introduce a special variable *pc* in each state. In each state, *pc* maps to a Boolean function over the feature variables. The function represents the presence condition that must be satisfied such that the state can be reached during program execution. For example, in Figure 6.3 the error state (division-by-zero error) is only reached if A and B are set to *false*, represented by the presence condition $\neg A \wedge \neg B$. In a reachability graph that is generated with presence conditions (e.g., Figure 6.3), each state has a *symbolic part* and an *explicit part.* The symbolic part contains *pc*, stored in a BDD, and the explicit part contains all other variables, stored in a map.

It is important that the introduction of presence conditions only modifies how states are stored. It does not require a change of the standard algorithm (Algorithm 6.1) and it does not change semantics of the model checking process. Each node of the reachability graph stores exactly the same information as it would without presence conditions, but using a different data structure. Therefore, this modification does not affect correctness of verdicts generated by the model checker. It also does not decrease the number of nodes in the graph, so we cannot expect verification speedup.

6.2.2 Path Joining

To achieve a performance improvement, we combine the presence-condition extension with a modification of how paths are joined during state-space exploration. In explicit-state model checking (partial) program path are joined when a path ends in a state that is already covered by other explored states (Line 7 in Algorithm 6.1). Normally, coverage means that the information stored in the states is equal. We weaken this requirement such that paths are joined even if only the explicit part of the states is equal. The symbolic part of the states may be different.

Algorithm 6.2 shows a pseudocode implementation of this extension, based on Algorithm 6.1. Based on our presence-condition extension (Section 6.2.1) a state w in Algorithm 6.2 has an explicit component w_e and a presence-condition component w_{pc}. Lines 1 to 6 of the algorithms remain unchanged. In Line 7, we test, for each direct successor state w, whether there is an already explored state with the same explicit component. If there is no such state, we add w to the *reached_set* and to the *wait_queue* for further exploration. If there is a state z with $w_e = z_e$, we can join the program paths leading to these states. If

Algorithm 6.2: Breadth-first model-checking algorithm with path joining. The rectangle marks the part that is changed with respect to Algorithm 6.1.

```
   Data: initial state $q_0$
   Result: Set $reached\_states$
1  add($reached\_states$, $q_0$)
2  enqueue($wait\_queue$, $q_0$)
3  while ¬ empty($wait\_queue$) do
4      let $v$ := dequeue($wait\_queue$)
5      if hasNoPropertyViolations($v$) then
6          foreach $w \in$ direct_succ($v$) do
7              if $\neg\exists\ z \in reached\_states : z_e = w_e$ then
8                  add($reached\_states$, $w$)
9                  enqueue($wait\_queue$, $w$)
10             else
11                 let $z \in reached\_states$ with $z_e = w_e$
12                 if $z_{pc} \not\Rightarrow w_{pc}$ then
13                     // new state is not completely covered by existing state
14                     let $z_{pc} := w_{pc} \vee z_{pc}$
15                     if $z \notin wait\_queue$ then
16                         // explore the state again
17                         enqueue($wait\_queue$, $z$)
18                     end
19                 end
20             end
21         end
22 end
23 return $reached\_states$
```

the presence condition of z_{pc} implies w_{pc}, w is already fully covered by z and all states that could be reached from w are also reachable from z. However, if z_{pc} does not imply w_{pc}, then w identifies configurations that are not covered by z. In this case, we increase the set of configurations of z. We set its presence condition to the disjunction of w'_{pc} and z'_{pc} (Line 14). The joint state z is an abstract state that represents both; it can be reached by satisfying w_{pc} or by satisfying z_{pc}. Because the configurations newly covered by z have not been explored, we add z to the *wait_queue*.

Figure 6.4 shows the effect of the path-joining extension on the example reachability graph. The frontier of the reachability graph (Figure 6.4) is one node smaller than the previous reachability graph (Figure 6.3). Our evaluation (Section 6.5) shows that this effect has a large impact on verification of larger systems.

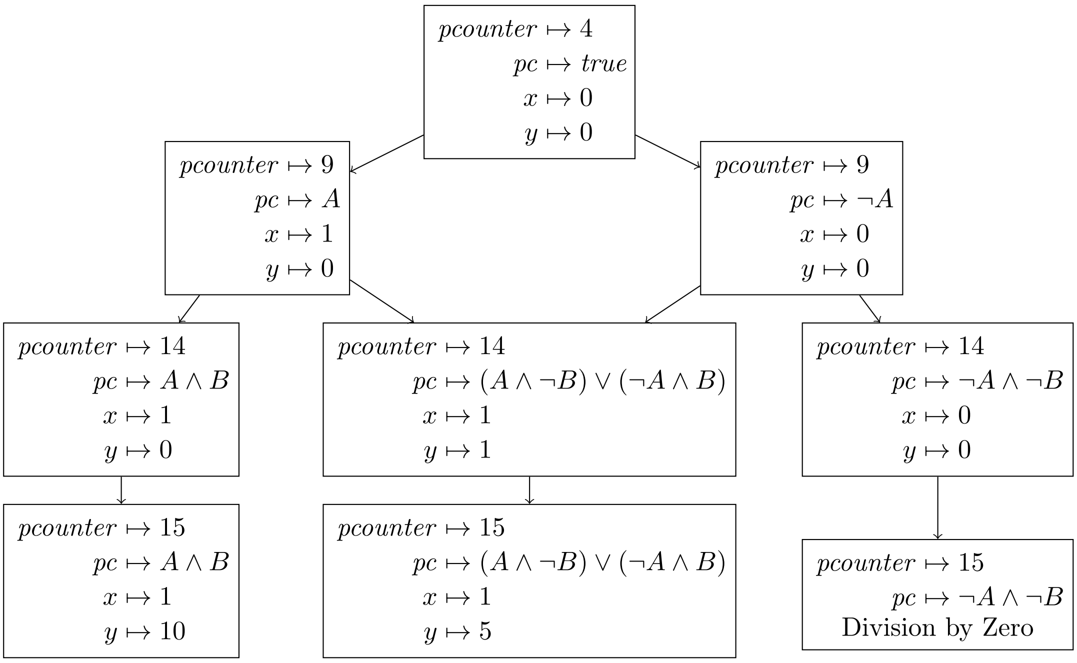


Figure 6.4: Reachability graph of the example program with presence conditions and merging

late splitting and early joining

The performance improvement of variability-aware model checking with respect to variant-based and sample-based model checking (checking variants separately) is based on the concepts of *late splitting* and *early joining*. Late splitting means that common prefixes of paths are analyzed in union as long as possible. This concept is also used in all discussed model checking approaches on the simulator (Figures 6.2,6.3, and 6.4). For example, the root node in the graph is only processed once instead of once per variant. Early joining means that paths are joined as soon as possible (during state-space exploration), which is achieved with our symbolic, path-joining extension (Figure 6.4).

correctness

We have to show that our path-joining extension does not affect the correctness of the verdicts generated by the model checker. More precisely, each concrete program state in a reachability graph without path joining must be represented by a state in the graph with path joining and the conditions under which the states are reachable must be equivalent. The same requirement must be valid in the other direction. In essence, this requires a trace-equivalence proof. At an abstract level, our path-joining extension is very similar to other, well-established *symbolic* model-checking approaches which also use symbolic data structures (e.g., BDDs) to represent reachable states [CGP99; Cla11].

Therefore, we skip the proof and provide an informal argumentation instead. First, we have to show that each abstract state $(e,(pc_1 \vee pc_2))$ represents all reachable concrete states with explicit component e that would occur without our extension and that it represents only these concrete states. This is given by construction, as the model-checking algorithm joins all reachable states with explicit component e and uses the disjunction of presence conditions as joint presence condition. Second, given an abstract state $a = (e, pc)$, we have to show that all abstract states $a' = (e', pc')$ that are direct successors of a represent concrete states that are direct successors of a's concrete states. If outgoing transitions from a are no feature choices, this is trivial, as presence conditions are not used in the transition. If the outgoing transition is a feature choice e.g., on presence condition pc_2, we generate two successor states $(e, pc \wedge pc_2)$ and $(e, pc \wedge \neg pc_2)$. These new states represent exactly the concrete states that would be generated from a in the unmodified model-checking approach. Therefore our variability-aware extension preserves correctness of the verification verdict.

6.2.3 Variability Pruning

Variability pruning is a model-checking extension that allows us to further reduce the number of analyzed states in the reachability graph. After we determined that a variant has a defect, we continue exploration of the state-space. Furthermore, we prune the state space such that the defective variant is not analyzed any more.

This optimization is based on the assumption that the goal of verification is to find at most *one* defect per variant of the system. The aim of software verification is usually to prove correctness of a program and if one counterexample (showing incorrectness) is found, the analysis stops (i.e., after detecting a property violation in Line 5 of Algorithm 6.2). In variant-based analysis, this approach detects one defect for *each* defective variant (separately). In simulator-based verification, the presence condition pc_{defect} of the defect identifies variants in which the defect occurs. However, there might be more defective variants whose defects are on other program paths. Therefore, simulator-based verification that stops after the first defect can produce a weaker verdict than variant-based verification (the verdict makes no statement about variants not identified by pc_{defect}). One goal of this chapter is to compare the family-based strategy against the variant-based strategy (Section 6.5). To enable an accurate comparison we have to ensure that both strategies generate comparable results (*one* defect for *each* defective variant).

To make simulator-based verification detect defects in *all* variants, we configure the model checker such that it does not stop after the first counterexample, but keeps exploring the reachability graph. We already implemented

continue after counterexample

Algorithm 6.3: Breadth-first model-checking algorithm with path joining and variability pruning. The rectangles mark the parts that are changed with respect to Algorithm 6.2.

```
Data: initial state $q_0$
Result: Set $reached\_states$
$global\text{-}constraint := true$
add($reached\_states$, $q_0$)
enqueue($wait\_queue$, $q_0$)
while $\neg$ empty($wait\_queue$) do
    let $v$ := dequeue($wait\_queue$)
    if hasNoPropertyViolations($v$) then
        foreach $w \in$ direct_succ($v$) do
            if sat($w_{pc} \wedge global\text{-}constraint$) then
                if $\neg\exists\ z \in reached\_states : z_e = w_e$ then
                    add($reached\_states$, $w$)
                    enqueue($wait\_queue$, $w$)
                else
                    let $z \in reached\_states$ with $z_e = w_e$
                    if $z_{pc} \not\Rightarrow w_{pc}$ then
                        // new state is not completely covered by existing state
                        let $z_{pc} := w_{pc} \vee z_{pc}$
                        if $z \notin wait\_queue$ then
                            // explore the state again
                            enqueue($wait\_queue$, $z$)
                        end
                    end
                end
            end
        end
    else
        $global\text{-}constraint := global\text{-}constraint \wedge \neg v_{pc}$
    end
end
return $reached\_states$
```

Algorithms 6.1 and 6.2 such that they continue after finding a counterexample. This configuration leads to the problem that we would explore a state (e, pc) even if we had already found defects in all variants represented by presence condition pc. Thus, we cannot prove *correctness* of any variant by exploring the state (e, pc) and we should avoid this unnecessary exploration effort.

variability pruning

Our solution for this problem is to introduce a functionality that allows to place additional constraints on the explored presence conditions. Constraints can be added at each point during the exploration of the search space. Al-

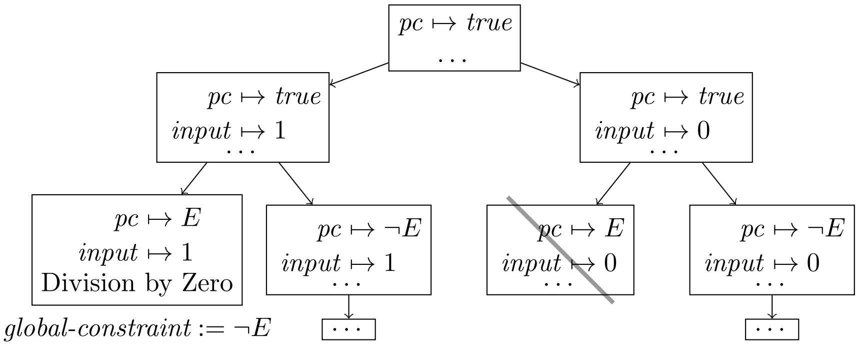


Figure 6.5: Example reachability graph with variability pruning. The global constraint $\neg E$ is set after the division-by-zero error is found.

gorithm 6.3 shows the model-checking algorithm with this extension. We introduce an additional variable *global-constraint* in Line 1. If a state violates a system specification (Line 6), we add the negation of the presence condition of this state to the global constraint (Line 27). In the further search, we only explore states which satisfy this constraint (Line 8).

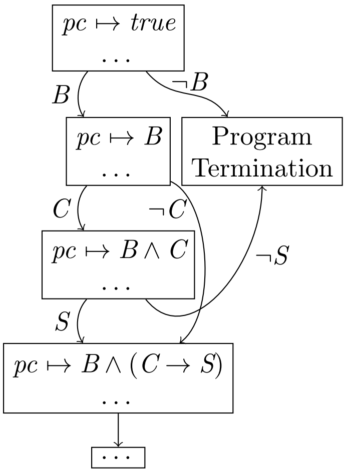
We use constraints to limit which variability is explored in the search space as illustrated in Figure 6.5. In the example, the first path splitting is not done because of feature variability, but because of a user input. User input is an example for nondeterministic input which forces the model checker to explore multiple paths. In each path, the model checker encounters a feature choice on feature E, which leads to a frontier of four states in total. One of the states with presence condition E has a division-by-zero error. Therefore part of the verification verdict is that all variants with feature E enabled have a division-by-zero defect. Then, the state-space-exploration continues for the remaining states and we prune the further exploration with the constraint $\neg E$. This enables us to focus on exploring variants for which we have not detected defects so far. In the example, we skip exploring the right state with $pc \mapsto E$ because the presence condition E is already covered by the first detected defect.

multiple defect-presence conditions

If we find more defects with presence conditions $pc_1, pc_2, \ldots, pc_n$, we build the global constraint $\neg pc_1 \wedge \neg pc_2 \wedge \ldots \wedge \neg pc_n$ as the conjunction of negations of all defect presence conditions. The state-space exploration stops when all states with presence conditions satisfying the global constraint have been explored. Then, exactly one defect has been found for each defective variant, which is equivalent of the set of all verdicts generated by variant-based model checking.

```
 1 class Printer {
 2   static boolean _FeatureDuplex_enabled=nondet();
 3   static boolean _FeatureBasicPrinter_enabled=nondet();
 4   static boolean _FeatureCopy_enabled=nondet();
 5   static boolean _FeatureScan_enabled=nondet();
 6   public static boolean variability_model() {
 7     return _FeatureBasicPrinter_enabled &&
 8       (!_FeatureCopy_enabled ||
 9        _FeatureScan_enabled)
10   }
11   public void printDuplex(Page front, Page back) {
12     if (_FeatureDuplex_enabled) {
13       //code from Duplex feature
14     } else {
15       //alternative code
16     }
17   }
18   ...
19   public static void main(String[] args) {
20     if (variability_model()) {
21         printDuplex(...); //execute test case
22       }
23   }
24 }
```

(a) Variant simulator with variability model



(b) Reachability graph

Figure 6.6: Code and reachability graph of a simulator of the printing-device system with variability model. The function gb() assigns *unknown* values to the feature variables. This forces the model checker to explore both possible values (true and false) when a feature variable is read. In the reachability graph, we encode features by their first letter.

6.2.4 Incorporating the Variability Model

Variability models of configurable systems typically limit the configuration space such that only a fraction of all combinations of configuration values represent valid configurations. In model checking of variant simulators, we use this information to limit the state space that needs to be explored. We limit the state space such that only presence conditions belonging to valid configurations are explored.

To achieve this limitation, we include the variability model in the variant simulator. Figure 6.6a shows a variant simulator of the printing device example with the variability model encoded in Line 7. The conditional statement in Line 20 ensures that only valid variants are simulated. At the beginning of verification, each feature variable is assigned an *unknown* value and, therefore, the model checker has to explore both possible values for each feature variable. The conditional statement ensures that each execution path that does not represent valid configurations terminates immediately. The corresponding reachability graph (Figure 6.6b) shows that this encoding generates a state with presence condition $B \wedge (C \rightarrow S)$, which is the variability model. This state is the "entry point" of all execution paths that explore the functionality of the configurable system.

An alternative approach to handle inclusion of the variability model would be to inject the model in the first state of the exploration (instead of starting the exploration with $pc \mapsto true$). However, this would require another extension of the model checking tool that would be specific to configurable systems. We chose to encode the variability model in the source code (Figure 6.6). This is a more flexible approach as it allows us to verify simulators with off-the-shelf model checkers that are not aware of the encoded (static) variability.

Both approaches include the variability model in the presence conditions of the reachability graph. This limits the reachability graph to valid configurations and therefore optimizes the model checking process. However, it also leads to presence conditions that are more complex than necessary. For example, consider that model checking discovers an error (e.g., division by zero) in the lowermost state in Figure 6.6b. The model checker reports that this error occurred under condition $B \wedge (C \rightarrow S)$, because that is the presence condition of the state. Ideally, we would expect the report that the error occurs in *all valid configurations*. This is exactly the problem described and solved with presence-condition simplification (Chapter 4). Using presence-condition simplification, we could report that the error occurs in all configurations that satisfy condition $VariabilityModel \wedge true$.

alternative to presence-condition simplification

As alternative solution for this problem, to avoid presence-condition simplification, we could introduce a second presence condition in all states of the reachability graph. We would initialize it with *true* after the variability-model conditional statement has been processed (lowermost state in Figure 6.6b). When values of feature variables are chosen during the model-checking process, we would update it exactly like the normal presence condition. If an error is found, we would report this new presence condition because it does not contain parts of the variability model that are not relevant for the error path. We implemented this alternative approach in a branch of our extension of the model checker JAVA PATHFINDER [VHB+03] (JPF-BDD [vRAR11], branch CustomChoiceTracking). Although this extension lets us avoid presence condition simplification, we do not use it in our experiments because it introduces additional overhead (one additional BDD operation per executed feature choice).

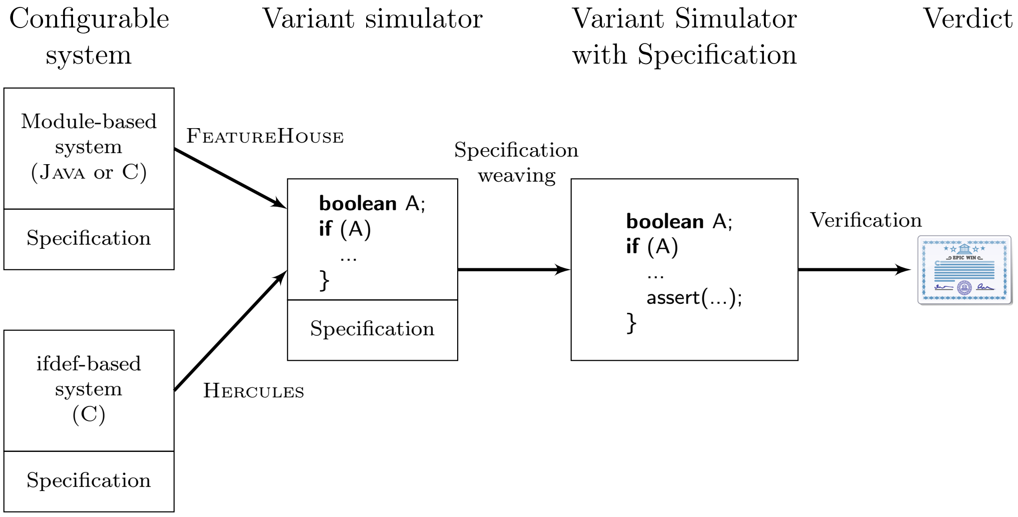


Figure 6.7: SPLverifier toolchain. The tool chain begins with the source code of a configurable system. We generate a variant simulator for the system, weave the system's specification, and verify it with model checking. If the system satisfies it's specification, the model checker generates a verdict for the system's correctness.

6.3 Implementation

Based on our work on variability encoding and the model checking extensions (Section 6.2), we developed a tool chain for configurable-system verification, called SPLverifier. It contains tools for variability encoding (FeatureHouse [AKL09] and Hercules, Section 5.5), specification weaving (AspectJ and ACC) and Java Pathfinder and CPAchecker for model checking (discussed in the following paragraphs). We provide links to the tools and to SPLverifier on the supplementary website. Figure 6.7 illustrates the verification workflow. It has three steps: variability encoding, specification weaving, and verification. We explain variability encoding in depth in Chapter 5. Specification weaving and verification are discussed in the following paragraphs.

Specification weaving In our subject systems, specifications are implemented in aspects (Aspect-Oriented Programming [KLM+97]). Some systems have multiple specification aspects that can be woven in the source code to check the specification. For example, the E-Mail system has a specification stating that e-mails that are forwarded must be encrypted in certain situations. The corresponding specification aspect contains an assertion that checks this property on every forwarded e-mail. This assertion is woven into the system before verification (further details in Section 6.5.2).

We use existing tools for specification weaving: ACC [5] for C code, and AspectJ [6] for Java code. We provide further details on the supplementary website.

Verification We implemented the model-checking optimizations described in Section 6.2 as extensions to the model checking tools Java Pathfinder [VHB^{+}03] (for Java programs) and CPAchecker[BK11] (for C programs). Both tools the verification of safety properties by means of explicit-state and symbolic model checking. Furthermore, both tools enable extensions using a plugin-oriented architecture.

We extended both tools with our variability-aware model checking optimizations (Sections 6.2). For Java Pathfinder, we bundled our implementation in the jpf-bdd extension [vRAR11]. For CPAchecker, our implementation is available in the CPAchecker distribution. It can be enabled using the configuration -explicitAnalysis-featureVars.

Both, Java Pathfinder and CPAchecker, implement verification of safety properties as opposed to liveness properties.[7] A *safety property* asserts that no bad states are reachable during program execution (e.g., a division-by-zero error). A *liveness property* asserts that a good state is reached infinitely often (e.g., "each process will enter its critical section infinitely often") [BK08]. Liveness properties are much harder to verify as they require the analysis to generate a verdict on a system that might never terminate (endless loop). In our experiments, we focus on subject systems which terminate after a given number of steps and we verify only safety properties.

6.4 Evaluation of Variability-aware Model-Checking Extensions

In this section, we describe an evaluation of the effect of our variability-aware extensions on the performance of family-based model checking. We compare the performance of family-based model checking with and without the previously described extensions (Section 6.2).

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

[6] http://eclipse.org/aspectj/

[7] CPAchecker also implements analyses that can verify liveness properties, depending on the implementation of the subject system. However, we focus on safety properties.

In other sections of this chapter (Sections 6.5 and Section 6.6.3), we describe evaluations comparing family-based model checking against other strategies. However, in this section we focus family-based model checking and evaluate the variability-aware extensions.

The evaluation is based on the model checker Java Pathfinder [VHB$^+$03] and jpf-bdd [vRAR11] (implementing the variability-aware extensions). As subject systems, we used simulators for configurable systems that do not contain defects. This way, we ensured that both verification approaches need to explore the same concrete state space.

Hypothesis Our hypothesis is that the variability-aware extensions (presence conditions and path joining) improve performance over standard verification of variant simulators. We expect this due to a reduction of the explored state space. The third improvement, variability pruning, is not used in this experiment, because the subject systems contain no defects.

Subject systems As subject systems, we selected the E-Mail system, the Elevator system, the configurable graph library GPL, and the compression tool ZipMe. These four configurable systems have been used before to assess configurable-system verification and we describe them in more detail in Section 6.5. We used the Java implementations of the systems and ensure that each simulator is free of defects.[8] In defective systems, a direct comparison of the approaches would not be possible because early aborts (after a defect is found) would blur the picture.

Experiment setup For the evaluation, we used the Java branch of our tool chain for configurable-system verification, called SPLverifier: For variability encoding, we used FeatureHouse [AKL09]. For model checking, we used the tool Java Pathfinder [VHB$^+$03] (revision 2 of version 8) and the jpf-bdd extension [vRAR11] (revision 107). jpf-bdd uses breath-first search (BFS) to explore the state space instead of depth-first search (DFS), which is the default in Java Pathfinder.[9] To show that the choice of algorithm is not responsible for the performance difference, we evaluated the standard Java Pathfinder with DFS and BFS. All experiments have been run with 7000 MB main memory.

[8] We de-activated the specification-checking code in the E-Mail and Elevator systems. GPL and ZipMe contain no defects.

[9] We use BFS in jpf-bdd because verification of configurable systems (with jpf-bdd) generates many mergeable states along different paths. After merging they would have to be re-explored, therefore it is better to merge early (more likely with BFS than with DFS).

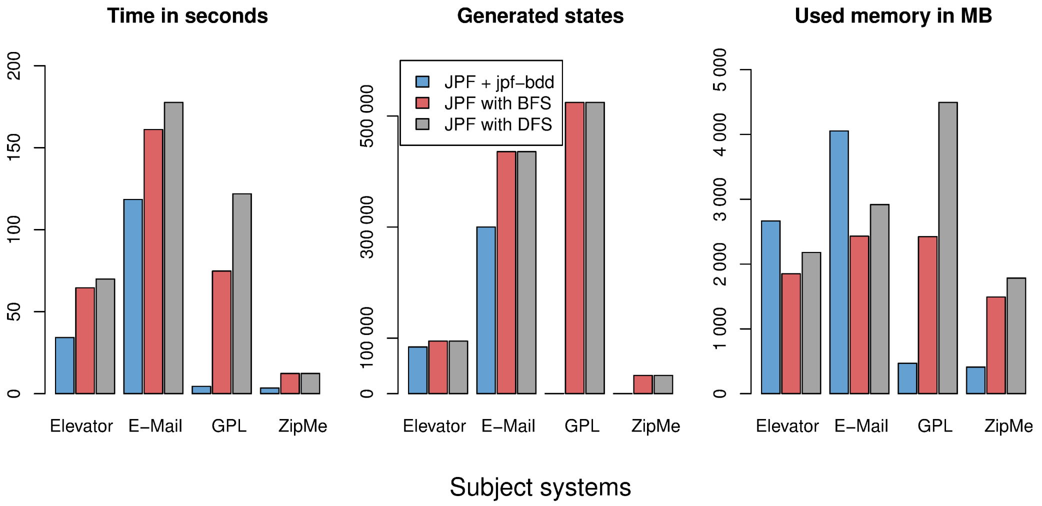


Figure 6.8: Improvement of simulator verification with variability-aware model-checking extensions

Results Figure 6.8 shows the results of our experiment. The left plot shows the time needed to verify correctness of the subject systems with the different verification approaches. On all systems Java Pathfinder with jpf-bdd needed significantly less time that the standard model checker with BFS or DFS.

time

The middle plot shows how many states were generated during the verification. In the E-Mail, GPL, and ZipMe systems, our extensions reduces the state space significantly allowing for faster exploration. In GPL and ZipMe the size of the state space is reduced by over 99%. In the E-Mail system, the effect is smaller, which is probably due to input parameters of the system that are not influenced by feature variables. These parameters guide which settings in the system are activated or deactivated and which e-mails are sent. As these settings are not reflected by feature variables, they force state separation even with our variability-aware extension. The differences between jpf-bdd and Java Pathfinder in the plot are similar to the time differences in the left plot, which indicates that the state space is one cause for the time difference.

states

The right plot shows that our extension consumes more memory than standard Java Pathfinder in the E-Mail and Elevator systems. However, it consumes less memory in GPL and ZipMe. This result shows that that our extension increases the size of each state (we add an additional BDD to each state). Even though the BDDs are stored efficiently in a BDD library (data structures shared between BDDs), each state consumes more memory than in standard Java Pathfinder. In the E-Mail and Elevator systems,

main memory

this leads to a high memory consumption of JPF-BDD because these systems generate large state spaces. In GPL and ZIPME, the optimizations of JPF-BDD reduce the state space so much that this reduction overlays the effect of larger states.

conclusion This small experiment has shown that variability-aware model checking improves over standard model checking of variant simulators. We extended this line of research to general programs in a publication that we do not discuss in detail here [ABF+13]. We extended the presence condition and path joining algorithms to general variables (in addition to feature variables) and also including linear arithmetic (instead of just Boolean operations). In this extended research topic, we first analyzed subject programs to sort variables according to their usage (*domain type*) and handle them with BDD-based extensions whenever appropriate. The domain-type analysis, the extended implementations, and experiments are documented in a research paper [ABF+13]. The outcome of these experiments was, similar to the experiment described in this section, that variables which are used in special ways (e.g., feature variables) should be handled with specialized approaches (e.g., BDDs).

These results show that our variability-aware extensions (presence conditions and path joining) improve the performance of family-based model checking compared to the off-the-shelf model checker. Therefore, we use family-based model checking *with* the extensions in our comparison of family-based, variant-based and sample-based model checking in the following section.

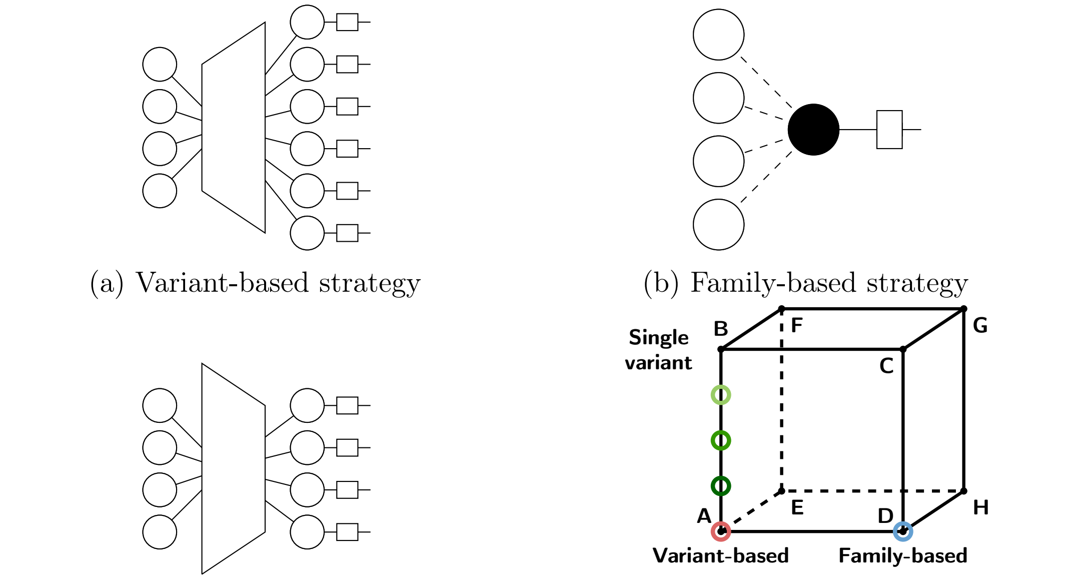
6.5 Evaluation of Family-based Model Checking

To evaluate the performance of family-based model checking, we have designed an experiment that compares family-based, sample-based and variant-based model checking. Figure 6.9 shows how the model-checking strategies are represented in the PLA model.

Sample-based and family-based strategies both promise to significantly decrease verification time, either by analyzing only a subset of variants, or by sharing analysis results among variants. To learn about the trade-offs, we compared the strategies in terms of their verification performance and their ability to detect defects; we use the variant-based strategy as a base line.

hypotheses Our hypotheses are:

- The family-based strategy is faster than the variant-based strategy because of the effects of late splitting and early joining.
- The sample-based strategies are faster than the family-based strategy (the fewer variants are checked, the less time is needed for verification), but may miss defective variants.



(a) Variant-based strategy

(b) Family-based strategy

(c) Sample-based strategies (feature-wise, pair-wise, and triple-wise sampling)

(d) Analysis strategies represented in the PLA cube

Figure 6.9: Illustration of the evaluated analysis strategies

- The family-based strategy consumes more verification time than sampling, but is exhaustive.

measurement definitions

To compare the strategies, we define the *sample rate* and the *defect recall*. The definitions are based on a configurable system with the set $\mathcal{V}$ of variants and the set $\mathcal{V}_d$ of defective variants, and on an analysis strategy focusing on a subset $\mathcal{V}_f$ of variants. The *sample rate* is defined as the relative number of variants chosen: $\textit{sample rate} = |\mathcal{V}_f|/|\mathcal{V}|$; the *defect recall* is the relative number of defective variants chosen: $\textit{defect recall} = (|\mathcal{V}_f \cap \mathcal{V}_d|)/|\mathcal{V}_d|$.

With regard to the sample-based strategy, we are interested in the tension between sample rate and defect recall and in their influence on verification time. For the family-based strategy, we are interested in the factors that influence verification performance. Especially, we want to explore whether late splitting and early joining are the driving factors for the speedups observed by us and others using family-based strategies. We quantify the number of verification steps that are saved due to sharing parts of paths among variants during the analysis.

For the experiments, we collected and prepared six case studies, which exceed the case studies used in previous work on configurable-system verification substantially, in terms of volume and complexity (Sect. 6.7). This corpus of case studies is also meant to serve as a benchmark suite in further work, which is a valuable contribution to the community.

Table 6.1: Overview of subject systems

System	Language	LOC	Features	Specs	Variants
E-Mail	Java	1233	9	9	40
	C	258	9	9	40
Elevator	Java	1046	6	9	20
	C	877	6	6	20
Mine Pump	Java	580	7	5	64
	C	279	7	5	64
AJStats	Java	13 393	20	2	200
GPL	Java	1405	18	2	42
ZipMe	Java	3636	8	1	10

6.5.1 Subject Systems

As subject systems for our evaluation, we selected six configurable systems that have been used as benchmarks in the configurable-system community before. In Table 6.1, we summarize relevant information on all case studies.

We selected three configurable systems that have been used before to assess configurable-system verification, and developed implementations in C and Java. We also provide information on these systems in Section 2.1.4.

- The **E-Mail** system of Hall [Hal05] models an e-mail communication suite. It provides several features, such as encryption, automatic forwarding, and e-mail signatures, which can be activated or deactivated.
- The **Elevator** system has been designed by Plath and Ryan [PR01]. It is an elevator model that is extensible by various features such as stopping if the elevator is empty or priority service for a special floor.
- The **mine-pump** system is based on work in the CONIC project [KMSL83]. It simulates a water pump in a mining operation, including several features that vary the pump's behavior. The pump keeps the bottom of the mine shaft dry, but must be deactivated if the mine contains combustible methane gas.

Based on the respective original systems, we created for each system a C and a Java implementation, obtaining six implementations in sum. Note that the respective Java and C implementations may differ in details. Although we aimed at comparability, the differences of the languages as well as the corresponding support of the model-checking tools forced us to diverge from a common implementation schema (e.g., the explicit-value analysis of CPAchecker did not yet support arrays and structures).

Additionally, we selected three existing configurable Java systems from the FeatureHouse repository.[10] All of them have been developed for other purposes. The primary selection criterion was that the Java code could be processed properly by Java Pathfinder.

- **AJStats** is a set of source-code–analysis tools providing several features to tailor the analysis process, for example, recognizing various syntactic program structures. It has been developed by Apel to explore the use of AspectJ [Ape10].
- **GPL** is a configurable graph library developed by Lopez-Herrejon and Batory [LB01], as a standard problem for the evaluation of product-line techniques. It allows a programmer to tailor graph data structures, including optional support for weighted and directed edges as well as different traversal strategies and algorithms.
- **ZipMe** is an open-source zip compression library for Java ME. It has been refactored into a configurable system by Kuhlemann [KBA09]. It includes features for computing check-sums and different compression techniques.

simulation of real-world systems

Note that, although the subject systems are implemented in C and Java, the implementations comprise only the key functionalities of the configurable systems. In this sense, the implementations model respective real-world configurable systems.

6.5.2 Behavior Specification

For the configurable systems E-Mail, Elevator, and Mine Pump, we adapted the original specifications, which have been distributed with their models. Mostly, the specifications concern domain-specific safety properties, such as that encrypted e-mails are never transferred in plain text, the elevator must refuse to operate if the maximum weight is exceeded, or the mine pump must be deactivated when methane gas is detected. Furthermore, all three case studies contain defects (documented by the original authors) violating, at least, one specification.

feature-based specification

In our case studies, individual features come with their own specification(s), expressed in the form of assertions that indicate erroneous executions, or by automata that are woven into the code in the form of assertions [ASW+11].

In Fig. 6.10, we show the specification of feature *Encrypt* of the E-Mail system expressed as an automaton: When the client receives an encrypted e-mail (Lines 7–9), the status (encrypted or not) of the message is stored into a field (Line 8) that has been attached as a shadow to the email structure (Line 4).

[10] http://fosd.net/fh/

```
 1 automaton EncryptSpec {
 2   // introduce an auxiliary field to store the state of an e-mail
 3   introduction {
 4     shadow struct email { int in_encrypted; };
 5   }
 6   // if an e-mail is encrypted when entering the system...
 7   before void incoming(_: struct client *, msg: struct email *) {
 8     msg->in_encrypted = isEncrypted(msg);
 9   }
10   // ...it must be encrypted as well when leaving the system
11   after void outgoing(_: struct client *, msg: struct email *) {
12     if (msg->in_encrypted && ! isEncrypted(msg)) { fail; }
13   }
14 }
```

Figure 6.10: Automaton-based specification of feature *Encrypt* of the E-MAIL system [ASW+11]

When an e-mail that was encrypted leaves the system (Lines 11–13), it must still be encrypted; if not, the e-mail client reaches an error state flagged by fail (Line 12).

For the configurable systems AJSTATS, GPL, and ZIPME, we included proper specifications based on domain knowledge (two for AJSTATS, two for GPL, and one for ZIPME). For example, the GPL implementation contains a function that builds a minimum spanning tree for a given graph. Our specification checks whether the resulting tree is really a valid minimum spanning tree. The systems AJSTATS and ZIPME do not contain defects; for GPL, we used defects introduced by others [CCR10]. Table 6.1 shows the number of specifications per case study.

6.5.3 Sampling Approaches

To compare family-based verification against sample-based verification, we choose *feature-wise*, *pair-wise*, and *triple-wise* as concrete sampling heuristics that select a subset of all valid variants. The heuristics are based on n-wise sampling of feature combinations, where n is the order of feature interactions that are covered at least by the sample set:

$$\begin{gathered}\text{for each subset } \{f_1, \ldots, f_n\} \subseteq F \text{ of } n \text{ features,}\\ \text{select a minimal variant } p \text{ with } f_1 \in p \wedge \ldots \wedge f_n \in p\end{gathered}$$

where F is the set of features of the configurable system and 'minimal variant' refers to a valid variant with the smallest possible number of enabled features.

pair-wise sampling

For $n = 2$ (pair-wise), all binary feature interactions can be detected. While this heuristic reduces the number of variants to be generated and analyzed significantly (from an exponential number, in the worst case, to a polynomial number), interactions between more than two features can be missed—so the analysis is not exhaustive. In our experiments, we also used two other sampling heuristics: single-wise ($n = 1$) and triple-wise ($n = 3$).

Our sampling algorithm does not guarantee to cover all negative interactions. For example, for a system with two features f_1 and f_2 it does not guarantee that there is a variant in the pair-wise sample set where f_1 is enabled and f_2 is disabled. However, in our experiment, we manually checked that all negative interactions are covered in our sample sets. Furthermore, our sampling approach generates sample sets with variants that contain few features. In our subject systems, this lead to smaller variants which is beneficial for software verification.

related work

The n-wise sampling heuristics are inspired by previous work on feature-interaction detection [CM06; JWEG07; PR98; SRK$^+$13]; they are tailored to pin down execution paths of individual feature interactions, without being distracted by other features (i.e., they aim at small sample variants). Alternative sampling heuristics that are used in configurable-system testing [OMR10] and bug finding [TLD$^+$11] have different characteristics and tradeoffs.

6.5.4 Experiment Setup

We performed all experiments on a Ubuntu 11.10 system that has an Intel i7-2600 CPU with 3.4 GHz, 8 cores, and 16 GB RAM. We used the SPLverifier tool chain for module-based configurable systems. For feature composition and variability encoding, we used FeatureHouse [AKL09]. For model checking, we used the CPAchecker [BK11] (revision 5540; branch "explicit") for C, and Java Pathfinder [VHB$^+$03] (revision 635) with jpf-bdd [vRAR11] (revision 23) for Java.

For each subject system, we created:

1. all variants (variant-based strategy),
2. sample sets that cover all (a) single-wise, (b) pair-wise, and (c) triple-wise feature combinations, and
3. a corresponding variant simulator using variability encoding (family-based strategy).

The overall goal of the verification tasks is to identify all defective variants (which violate the specification of one feature, at least) of the given configurable system. That is, for each specification, we have to run a sequence of verification tasks: for the variant-based and sample-based strategies, one verification task per (selected) variant; for the family-based strategy, one verification task for the variant simulator.

For the variant-based and sample-based strategies, we terminated the verification task after detecting a violation, and continued with the next variant to be checked. For the family-based strategy, we determined which variants contributed to the detected violation, and proceeded with the exploration of the remaining state space that was not associated with these variants as discussed in Section 6.2.

The composition time for our subject systems is negligible compared to the verification time, thus we compared the verification times only (including the generation of counterexamples). That is, we measured five values for each specification to be checked (variant-based, single-wise, pair-wise, triple-wise, family-based). Additionally, for the sample-based strategies, we determined what percentage of defective variants has been identified (i.e., the *defect recall*), and we logged what percentage of variants has been checked.

6.5.5 Results

In Fig. 6.11, we illustrate the relative verification times of using the sample-based and the family-based strategies for each subject system, compared to the variant-based strategies (whose time defines the 100 %). As expected, sample-based and family-based strategies can improve verification performance significantly compared to a variant-based strategy (except for triple-wise sampling in ZipMe, which effectively selects all possible variants): single-wise by 75 %, pair-wise by 60 %, triple-wise by 33 %, and family-based by 73 %, on average.

The family-based strategy is in many cases faster than most of our sampling heuristics; for six subject systems, it outperforms even single-wise sampling. Note that, using a sample-based strategy, we may find only a fraction of all defective variants. In Fig. 6.12, we provide sample rate and defect recall (side by side, for later comparison) for the different sampling heuristics. The defect recall for AJStats and ZipMe is omitted because they do not contain defects. All raw data of this experiment is available in our *ICSE* publication [AvRW$^+$13] and on the supplementary website.

6.5.6 Discussion

We divide our discussion into three parts, regarding (1) the sample-based strategy, (2) the family-based strategy, and (3) a comparison of the two.

Sample-based strategy We consider how the defect recall is related to the time that is actually needed for verification. Ideally, the goal is to achieve a high

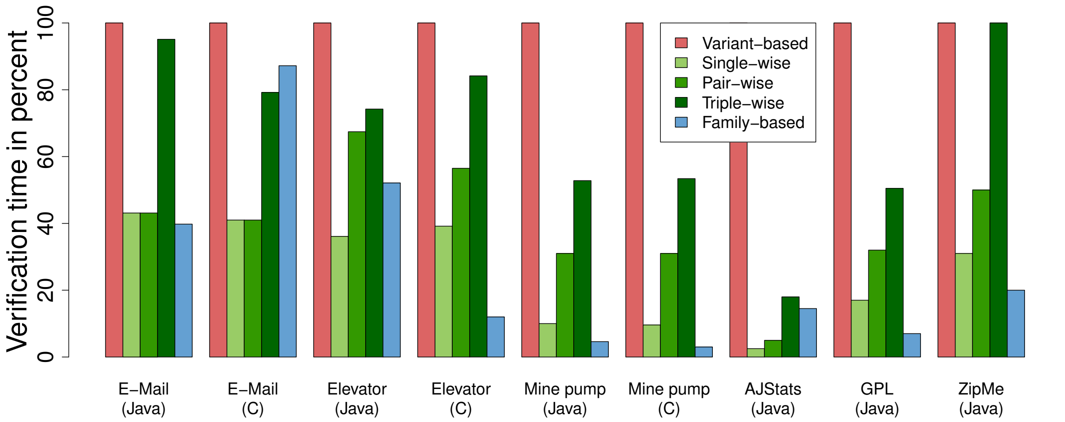


Figure 6.11: Comparison of average verification times; variant-based strategy defines the 100 %

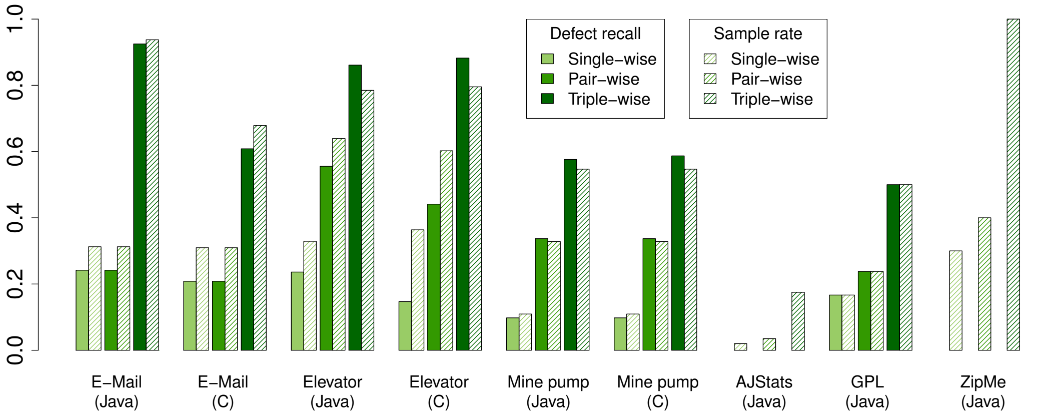


Figure 6.12: Defect recall and sample rate of the sample-based strategies (AJStats and ZipMe do not contain any defect)

defect recall and a low verification time, compared to the variant-based strategy. In Fig. 6.13, we show for each sample-based verification of our experiments the defect recall and the corresponding time fraction. The higher the defect recall is for a given time fraction that is needed for verification, the better is the sampling heuristic. In our experiments, the triple-wise sampling heuristic has a high defect recall, as compared to the time spent for verification; the ratio between defect recall and fraction of verification time is large in many cases (cf. Fig. 6.12). We get back to this ratio when we compare sample-based and family-based strategies.

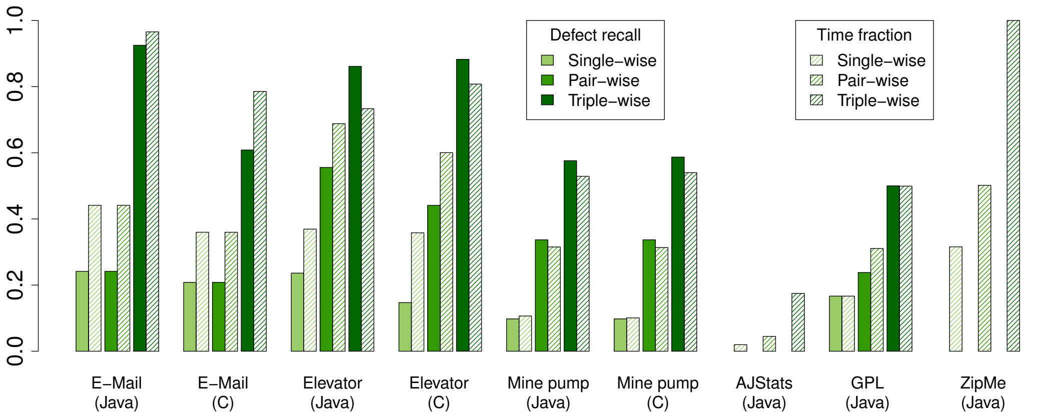


Figure 6.13: Defect recall versus fractions of verification time (AJStats and ZipMe do not contain any defect)

Family-based strategy For the family-based strategy, it is more difficult to explain the observed verification times. In our experiments, we observed speedups of up to thirty times (Mine Pump), compared to a variant-based strategy.

A key question—which has not been answered in previous work—is whether late splitting and early joining are the driving factors for the observed speedups, or whether other effects such as internal optimizations in the model checker or technical issues play a dominant role. Hence, we instrumented jpf-bdd to quantify the verification steps saved due to late splitting and early joining. Specifically, for each transition $t \in T$ (where T is the set of transitions in the configurable system), we computed the number of instructions that it contains (cost C_t) and in how many variants P_t it would have been executed. Then $\sum_{t \in T} C_t * (P_t - 1)$ is the number of verification steps (i.e., executed instructions) that are saved due to late splitting and early joining.

In Fig. 6.14, we illustrate the fraction of verification steps (in relation to the verification steps needed without late splitting and early joining) and the fraction of verification time that the family-based strategy needs (in relation to the variant-based strategy). Although the data points are not on the dotted line, a statistical analysis reveals that the fraction of verification steps and the fraction of verification time correlate (Pearson's product-moment correlation: $cor = 0.84$, $p \ll 0.05$). This correlation suggests that the principles of late splitting and early joining can explain similarity within configurable systems (i.e., the amount of shared instructions could be used as similarity degree). However, the time consumed by join operations depends on the size of the BDDs—a factor that causes the deviations in Fig. 6.14.

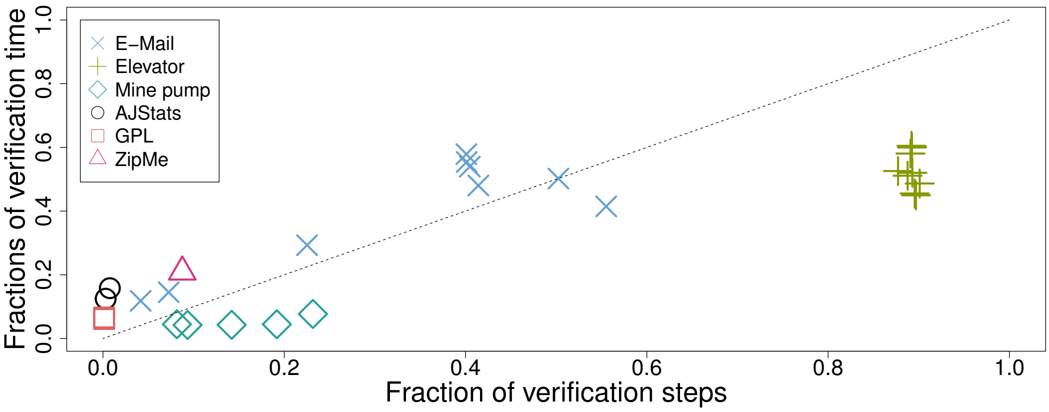


Figure 6.14: Fractions of verification steps and verification time using late splitting and early joining; correlation coefficient: 0.84 ($p \ll 0.05$)

Family-based vs. sample-based strategies Our experimental data suggest that both the family-based strategy and the sample-based strategies outperform the variant-based strategy in terms of verification performance. But which strategy is superior? We cannot compare solely their verification times, but we have to take into account that, for sample-based strategies, the defect recall decreases with the sample rate. Hence, we compared the strategies with regard to *detection efficiency*, which we define as the ratio between defect recall and the time fraction (both in relation to the variant-based strategy): *detection efficiency* = *defect recall*/*time fraction*. A verification strategy with a detection efficiency of one is similarly efficient as the variant-based strategy.

In Fig. 6.15, we show the detection efficiencies for all our experiments, grouped by subject systems. It reveals that the family-based strategy is the most detection-efficient strategy. (The notion of detection efficiency can of course not be applied to AJStats and ZipMe, which do not contain any defect.) Of the sample-based strategies, only triple-wise sampling exceeds in some cases the detection efficiency of variant-based verification. The fact that the family-based strategy is mostly superior—in terms of detection efficiency—over our sample-based heuristics is one of the main results of our experiments.

6.5.7 Threats to Validity

The kind and distribution of defects threatens internal validity, because they affect the defect recall. If defects occur only if many features interact, then sampling heuristics such as pair-wise are only of limited use. However, the

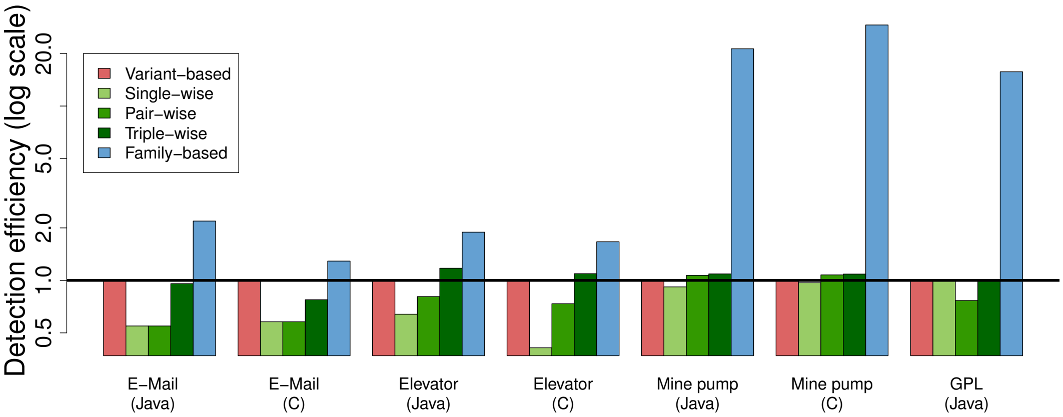
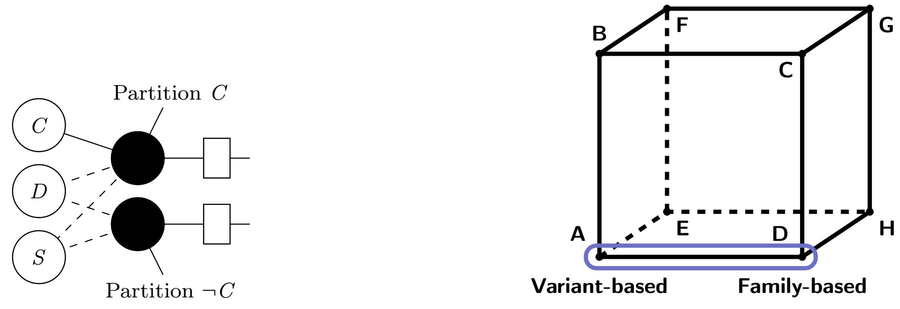


Figure 6.15: Detection efficiencies of different verification strategies (AJStats and ZipMe omitted because they do not contain any defect)

systems under investigation contained only defects that occur within single features or among pairs of features, which reflects what is known about the distribution and probability of feature interactions [CM06; KMMR00; POS^{+}12].

Much like the kind and distribution of defects, several other characteristics of a configurable system influence the benefits of the individual strategies. For example, the number and distribution of dependencies among features (as documented in the feature model) can have an influence on the sample and defect recall; the degree of code sharing among variants can influence the potential for late splitting and early joining; the granularity of variability may have an effect on the efficiency of join operations (based on BDDs). Further work should develop proper feature-model or code measures to predict the benefits of sample-based and family-based strategies and possibly combinations thereof.

The choice of the subject systems threatens external validity. Hence, we selected as many subject systems as we were able to locate, including standard benchmarks that condense the state-of-the-art in the field. But the tool chain that we used, as well as the availability of configurable systems that contain specifications and that are amenable to model checking, were limiting factors. Nevertheless, for the first time, a substantial set of different subject systems written in different languages has been considered for evaluating strategies of configurable-system verification.



(a) Family-based strategy with two configuration-space partitions

(b) Analysis strategies represented in the PLA cube

Figure 6.16: Illustration of the evaluated analysis strategies

6.6 Combining Family-based and Variant-based Model Checking

The results from the previous section have shown that family-based model checking has a higher detection efficiency than variant-based verification. However, the variant simulators were larger than any of the variants. Furthermore, we discussed (informally) that verification of simulators typically consumed more main memory than verification of a variant. In the evaluation part of this section we provide numbers showing this fact (Section 6.6.3). The memory consumption of family-based verification might limit its applicability on subject systems with many variants or on machines with low available main memory.

In this section, we discuss how this limitation of the family-based strategy can be mitigated using a combination of model-checking strategies based on the PLA model. We evaluated how combined strategies perform in comparison to basic strategies such as the family-based and variant-based strategy.

partition-based analysis

We describe a set of analysis strategies where each strategy is represented by a point between points **A** and **D** on the PLA cube. For each strategy, we partition the configuration space of a configurable system and analyze each partition with a variability-encoded simulator. The rationale behind this strategy is that all variants are covered by one of the simulators while each simulator is cheaper to analyze than the simulator that covers all variants. Furthermore, the smaller simulators can be verified in parallel processes (similar to variants in the variant-based strategy). Each strategy represents an exhaustive analysis of all variants of the system, because each variant is covered in each strategy by exactly one partition.

This section provides an initial study into how combination of strategies influences analysis performance. For this initial study, we focus on model checking

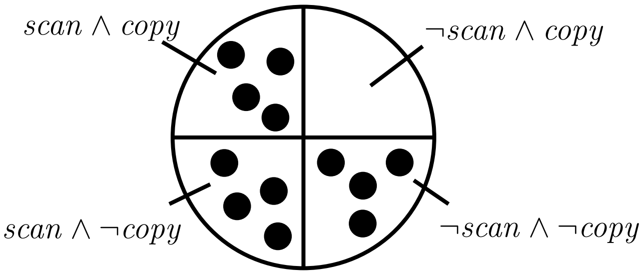


Figure 6.17: Example partitioning of the printing-device configuration space. The circle represents the configuration space and the black dots represent different configurations. The partition $\neg scan \wedge copy$ is empty because the system has no variants with *copy* and without *scan*.

of JAVA systems using the tools and subject systems we describe in Sections 6.5. We evaluated *all* possible analysis strategies between points **A** and **D** (Figure 6.16b) for four different subject systems. Figure 6.16a shows how an example strategy with two partitions of the printing-device example is represented with PLA operators. In a first step, we partition the configuration space. We build two simulators (filled circles in Figure 6.16a); one for simulating all configurations with feature *Copy* (C) and one simulating all configuration without *Copy*. In a second step, we verify the simulators separately.

The study sheds light on a fundamental question for model checking of configurable systems: Can we use combinations of strategies to improve the efficiency (run time, main-memory requirements, and defect coverage) compared to basic strategies such as family-based and variant-based model checking?

In Section 6.6.1, we describe the set of analysis strategies that we focus on in this evaluation. In Section 6.6.2, we describe how we generate our experiment setup and which subject systems we use. In Section 6.6.3, we discuss our evaluation and its results.

6.6.1 Analysis Strategy

We explore model checking strategies on the lower front edge of the cube (range **A**–**D**). In addition to verifying a simulator for the complete system (**D**) and verifying all variants (**A**), we partition the configuration space and verify each partition with a dedicated simulator.

For example, Figure 6.17 shows a partitioning of the configuration space of the printing-device system based on the features *scan* and *copy*. Each partition has a constraint (e.g., $scan \wedge \neg copy$) and it contains all valid configurations in which the constraint is satisfied. The constraints are mutually exclusive and, together, cover the whole configuration space. Therefore each configuration is covered by exactly one partition. The partition $\neg scan \land copy$ is empty because the variability model of the printing-device system prohibits configurations with *copy* and without *scan*.

To verify partitions of a system, we generated a variant-simulator for each non-empty partition (three simulators in the example) and used model checking to verify the simulators. In Section 6.6.3, we discuss how we applied this strategy to the E-MAIL, ELEVATOR, MINE PUMP, and ZIPME systems from Chapter 6. Based on the results, we evaluated how efficiency of the verification strategy changes if more (smaller) or less (larger) partitions are used. The number of partitions determines which point in the PLA cube represents the analysis:

- If we use only one partition, the strategy is equivalent to the family-based strategy (point **D**).
- If we use as many partitions as possible (one variant per partition), the strategy is equivalent to the variant-based strategy (point **A**).
- If we use partitions with more than one and less than all variants, the strategy is represented by a point between **A** and **D**.

6.6.2 Generating Partitionings for the Evaluation

The performance of a partition-based strategy depends on the number of partitions and their size. Therefore we decided to evaluate *all* possible partitions of the subject system's configuration spaces, to completely cover range **A**–**D**.

In the following paragraphs we describe the algorithm that we use to generate partitionings for our experiment and how the complexity of this algorithm limits our choice of subject systems. The complexity of the algorithm stems from the fact that it needs to generate *all* partitionings. Generating *one* partitioning is much simpler and can also be applied on larger subject systems.

Partition generation algorithm To generate all possible partitionings of a subject system's configuration space, we first computed the powerset of the set of features of the system. For example, the powerset of the the printing-device features has 32 entries:

$$\{\}, \{\mathit{BasicPrinter}\}, \{\mathit{Duplex}\}, \ldots, \{\mathit{BasicPrinter}, \mathit{Duplex}\}, \ldots$$

From each entry, we computed constraints for partitions. For example entry $\{\mathit{BasicPrinter}, \mathit{Duplex}\}$ yields the constraints

$$\mathit{BasicPrinter} \land \mathit{Duplex}, \quad \mathit{BasicPrinter} \land \neg\mathit{Duplex},$$

$$\neg\mathit{BasicPrinter} \land \mathit{Duplex}, \text{ and } \quad \neg\mathit{BasicPrinter} \land \neg\mathit{Duplex}$$

Table 6.2: Partitionings of configuration spaces. The number of partitionings is determined by the number of features and by constraints imposed by the variability model. We cannot provide a listing of the partitionings here, due to their size.

System	Features	Variants	Number of partitionings
E-Mail	9	40	64
Elevator	6	20	32
Mine Pump	7	64	64
ZipMe	8	10	17

Next, we filtered constraints that represent empty partitions (e.g., the third and fourth constraint in the example). Furthermore, we filtered constraints that represent equal sets of configurations.[11] Then we generated simulators for the remaining constraints. Each partitioning of the configuration space (generated from one entry of the powerset), corresponds to a set of simulators, and to one analysis strategy in range **A**–**D**. For each strategy, we verified all simulators, measured how long verification takes and how much memory is consumed (maximum). We discuss the evaluation in Section 6.6.3.

Subject systems In this evaluation, we focus on selected subject systems from our evaluation of family-based model checking (Section 6.5). We selected the E-Mail, Elevator, Mine Pump, and ZipMe systems. We excluded AJStats and GPL because of their large feature sets (20 and 18 features), which would cause scalability problems for our partitioning-generation algorithm. In particular, the generation of the powerset of features is infeasible for such large feature sets. Generating *one* partitioning and running the corresponding strategy would also work for AJStats and GPL. However, to cover all strategies in range **A**–**D**, we need *all* partitionings which is unrealistic for these systems. For each system, we verified one specification that is applicable to all variants (some specifications require specific features which would be inadequate in this experiment).

Table 6.2 lists the number of different strategies for each subject system. These strategies include one pure family-based strategy (one partition, point **D**) and one pure variant-based strategy (one partition per variant; point **A**).

[11] This can occur in systems with larger variability models. For example, in the E-Mail system, features *encrypt* and *decrypt* imply each other, yielding identical partitions.

6.6.3 Evaluation

We evaluated how family-based and variant-based verification performs in comparison with the verification of multiple simulators that cover mutually exclusive sets of variants. We used the strategies generated as described in the previous section and verified the simulators of each strategy independently. Each strategy represents an exhaustive analysis of all variants of the system, because each variant is covered by exactly one partition of each strategy. Therefore, each strategy detects all defects and detection- and sample rates (cf. Section 6.5) are identical for all strategies. To compare the strategies, we measured the time needed for sequential verification of all simulators of a strategy and the maximum amount of main memory needed during the verification.

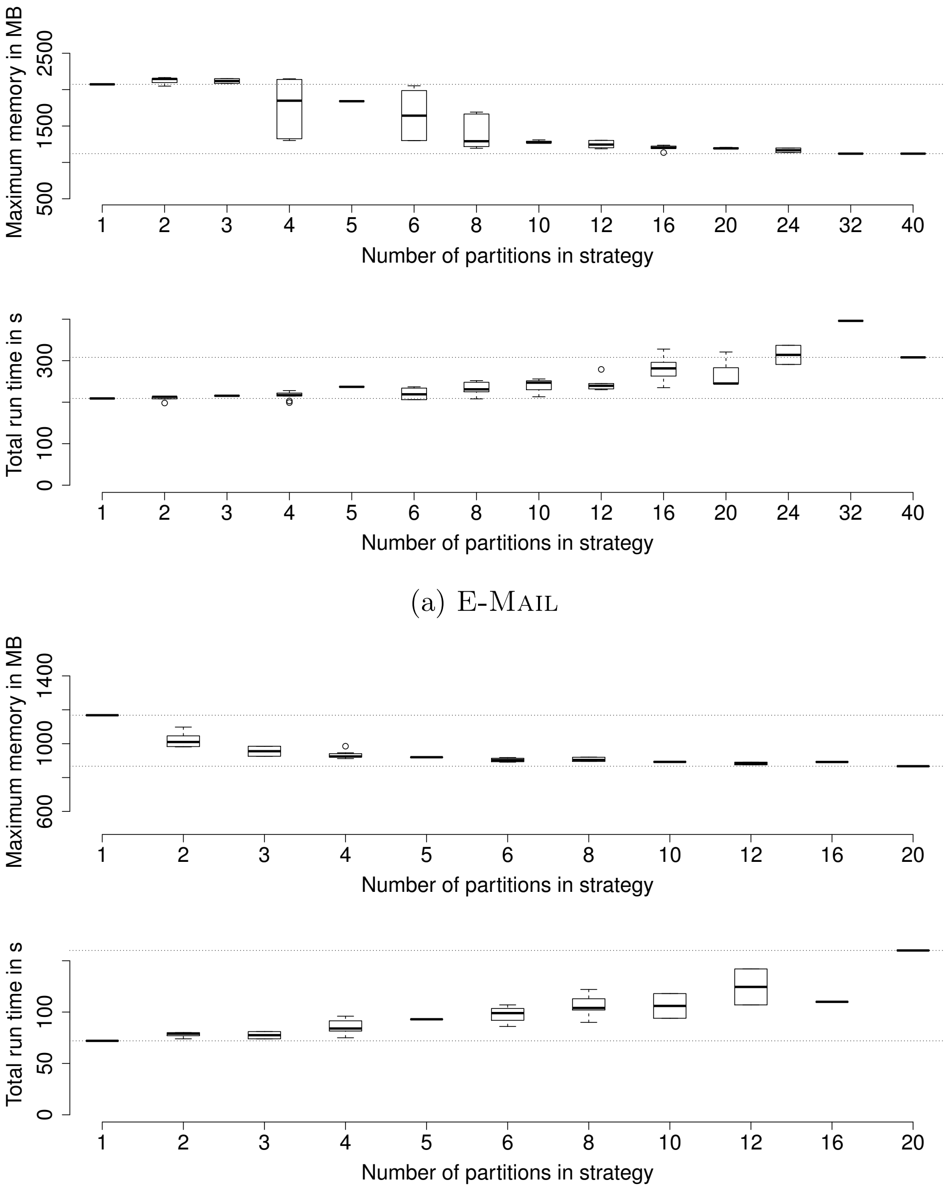
In each verification run, we used a simulator with all features of the system (as in the family-based approach). We limited the simulator such that only variants in the partition are verified and the model checker ignores all other code.[12]

Hypothesis Our hypothesis is that the combining the family-based and variant-based strategies yields improves performance compared to the pure strategies. In particular, we hypothesize (1) that combined strategies are faster than the variant-based strategy and (2) that they consume less memory than the family-based strategy.

System setup We ran the experiment on a Ubuntu 14.04 workstation with four cores (Intel Xeon Processor X3470 @ 2.93GHz). For model checking, we used the tool Java Pathfinder [VHB+03] (revision 1147 of version 6) and the jpf-bdd extension [vRAR11] (revision 101). Each verification run was allowed a maximum of 4 GB RAM. There were no out-of-memory errors or otherwise aborted verification runs.

Results Figure 6.18 and Figure 6.19 show the results of our experiments. In each plot, we aggregated strategies by the number of used simulators (equal to the number of partitions) and show them with one boxplot. For example, in the run-time plot of Figure 6.18a, the run times of all verification strategies that use 8 simulators is shown in one boxplot. In each plot, the boxplots are ordered by the number of simulators used. The left-most strategy uses only one simulator (family-based strategy). The right-most strategy uses one simulator per variant (corresponding to the variant-based strategy).

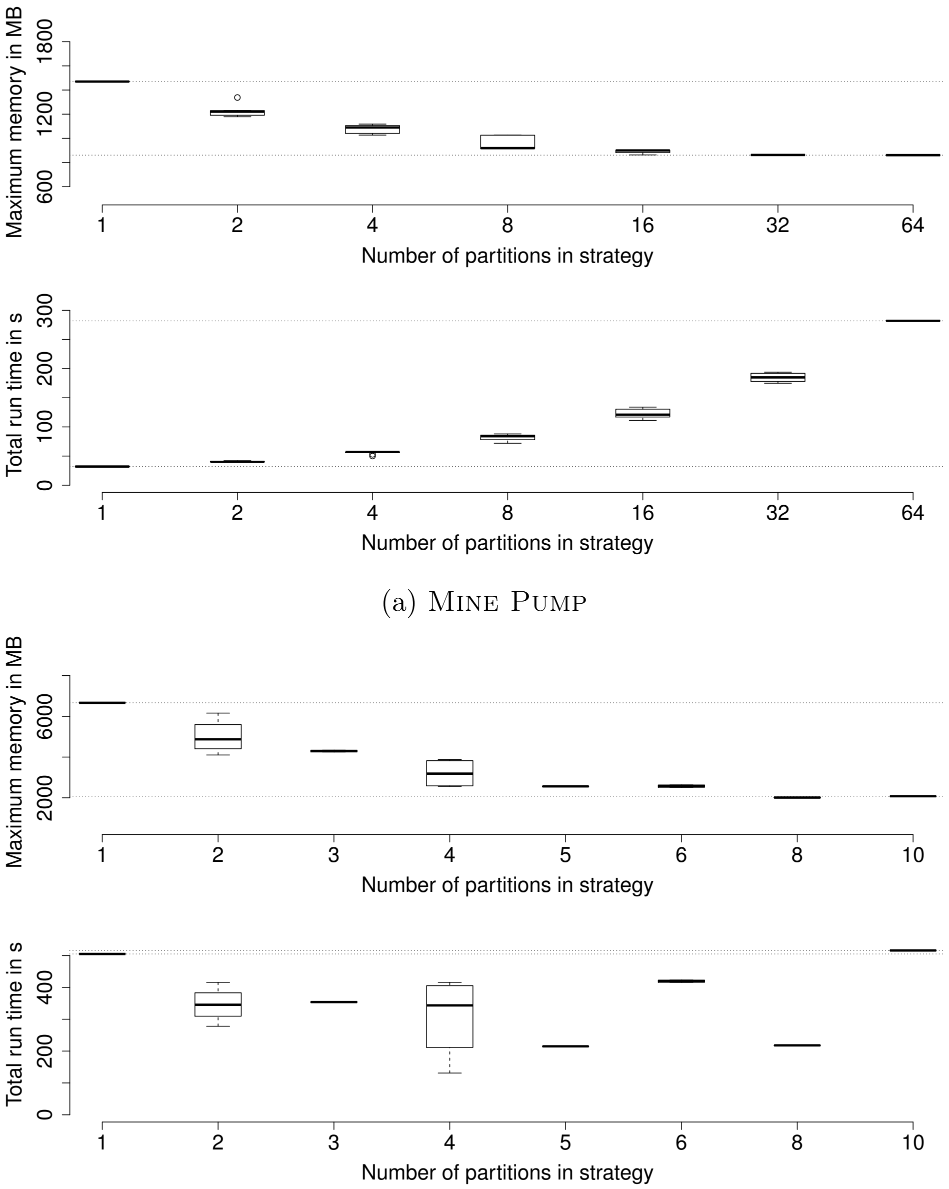
[12] Given the constraint c of a partition, we inserted an additional guard `if (c)` to the body of the system's main function (cf. Section 6.2.4).



(a) E-Mail

(b) Elevator

Figure 6.18: Verification time and maximum memory consumption of verification based on configuration-space partitions for the case studies E-Mail and Elevator. Each boxplot represents strategies with the same number of partitions/simulators. The y-axes in the memory-consumption plots are truncated (without removing results). Dotted lines mark the family-based (left; one partition) and variant-based (right) strategies.



(a) Mine Pump

(b) ZipMe

Figure 6.19: Verification time and maximum memory consumption of verification based on configuration-space partitions for the case studies Mine Pump and ZipMe. As in Figure 6.18, each boxplot represents strategies with the same number of partitions/simulators. The y-axes in the memory-consumption plots are truncated (without removing results). Dotted lines mark the family-based (left; one partition) and variant-based (right) strategies.

The plots in the left columns of Figure 6.18 and Figure 6.19 show the maximum memory consumed during the verification of all simulators of a strategy (sequentially; we did not verify simulators in parallel). For example, the plot shows that using only one simulator (family-based strategy) consumes much more memory than using one simulator per variant (variant-based strategy), which is a motivation for this evaluation. More general, the plot shows that using more simulators usually means that a strategy consumes less memory. This result was expected, as more simulators mean that each simulator represents a smaller part of the configuration space. Therefore the state space that is generated during model checking is smaller and consumes less memory. Even though there are techniques for efficient storage of the state space [EP05; Hol97], it still is the main driver for memory consumption in model checking. Of course there are exceptions to this observation: for example, the memory-consumption plot of Figure 6.18b shows that some outlier strategies with 4 simulators consumed less memory than selected strategies with 8 simulators. The reason for this exception probably lies in the design of the ELEVATOR system. For example, if one particular data structure in the system requires much memory during verification, it would be beneficial to choose partitions such that the structure is verified in only one partition.

The plots in the right columns of Figure 6.18 and Figure 6.19 show the time needed for the verification of all simulators of a strategy. Usually strategies with more simulators need more time than strategies with less simulators. This result is supported by the results of our experiments in Section 6.5. However, there are exceptions to this rule. For example, in the ZIPME system almost all strategies need less time than the pure family-based strategy. There are several possible reasons for a deviation between ZIPME and the other systems: First, ZIPME does not have any defective variants such that no program paths terminate “early”. Second, ZIPME reads from files, and depending on features, uses archiving algorithms and computes checksums. It is plausible that a simulator that has to switch between expensive algorithms is harder to verify than several simulators that each verify one algorithm. However, the purpose of our evaluation is to determine whether the combination of variant-based and family-based verification improves performance. It is beyond the scope of this experiment to evaluate the effects of different configurable-system properties on the simulator performance. Such evaluation would require a much broader set of subject systems.

Discussion Coming back to our initial question of whether combinations of strategies can improve efficiency compared to basic strategies, our experiment has shown that partition-based verification indeed often improves efficiency.

In most strategies, the consumed memory was lower than in the family-based strategy and the verification time was lower than in the variant-based strategy. The experiment has also shown some outliers to this result, which imply that one cannot easily predict performance of verification approaches. In particular, it is difficult to predict how a given configuration-space partitioning would influence verification performance.

In our evaluation, we were limited to configurable systems with 9 or less features because generation and comparison of *all* partitionings suffers from an exponential blowup with larger numbers of features. However, this does not mean that our results do not apply to configurable systems with more features. In fact, a larger future study could test our results on real-world systems by building and comparing a selected (small) set of partitionings.

rating partitionings

To further investigate which partitionings improve verification performance, we tried to isolate properties of partitionings that achieved especially good or bad results. For example, we identified partitionings that used only features that do not have dependencies on other features. For such features, the number of variants that include the feature is the same as the number of variants without the feature. Therefore, such partitionings should in theory induce simulators of roughly equal size because each simulator covers an equal size of the configuration space. However, our experiment data does not support this theory. Similar properties, such as trying to distribute lines of code uniformly between simulators of partitionings, did not yield substantially better-than-average results, either.

Determining which partitionings are beneficial under which circumstances (e.g., subject system and verification tool) remains future work. We showed that strategies which combine variant-based and family-based strategies are better than the basic strategies. Determining which properties in subject systems increase this advantage requires a more thorough study with more subject systems. Such study would go beyond the scope of this thesis where we predominantly want to show that combinations of strategies are beneficial. However, the example strategies in this section can easily be implemented in other verification tools. For example, the model checker FMC [tBFGM15] can verify simulators (of labelled transition systems) and allows to limit them to a subset of configurations before verification. It could be used to explore partition-based verification in a more controlled setting than it is possible with software model checking of real-world systems.

6.7 Related Work

In this section we discuss related work that applies verification approaches to configurable systems. Other related work is discussed in Chapter 5 (other uses of variability encoding) and in Chapter 3 (comparisons of different analysis strategies). For a comprehensive overview on strategy comparisons, we recommend a survey report [TAK$^+$14].

The family-based strategy has been used in several model-checking approaches [ASW$^+$11; AtBFG11; CHSL11; CHS$^+$10; Gru10; LTP09; PS08; tBdV14]. It has been shown that substantial performance gains are possible [ASW$^+$11; CHSL11; CHS$^+$10]: for example, an average speedup of two was observed [CHSL11; CHS$^+$10], by using a family-based strategy, compared to the variant-based strategy (and a speedup of sometimes two orders of magnitude if using BDDs for feature variables). Different related tools (e.g., SNIP [CHSL11; CHS$^+$10], VMC [tBGM15; tBM14], FMC [tBFGM15], and ABC [vRBS$^+$15]), have been developed for family-based model checking of transition-system models of configurable systems. These tools implement similar model-checking optimizations, however they focus on verification of transition systems where we focus on verification of JAVA and C source code. In particular VMC uses a family-based strategy which is very similar to ours: VMC uses a transformation similar to our variability encoding: The verified system is given as modal transition system and the specification in v-ACTL [tBM14] (both allow variability annotations). VMC transforms the artifacts doubly-labeled transition systems with ACTL [dNV90] specifications. The transformed artifacts can then be verified using existing model checkers. In difference to our work, which focuses on source code artifacts, they focus on automata-based models of configurable systems. A recent addition to VMC allows users to define input models in which integer-valued data can be passed between automata states, which improves potential for realistic case studies [tBFG14].

The family-based strategy has been combined with other verification approaches, for example with counterexample-guided abstraction refinement [CGJ$^+$03]. It has been shown that this combination can improve the performance of verification of labelled transition systems [CHL$^+$14]. In future work, counterexample-guided abstraction refinement could be applied similarly to our software-model-checking approach.

Feature-based verification, which aims at verifying features as far as possible in isolation to minimize the verification effort upon feature composition [LKF02b; LBL11; MRKN13], was not considered in our comparative experiments because practical verification tools and corresponding case studies were not available.

CHAPTER 7

Inter-App Data-Flow Analysis in ANDROID Systems

This chapter shares material with the publication "Lifting Inter-App Data-Flow Analysis to Large App Sets" [vRBS+15].

In this chapter, we discuss how the feature-based and family-based strategies can be used for scalable analysis of communication between apps in an ANDROID mobile-device. We implemented an analysis of communication between ANDROID apps that is able to detect leaks of private information. To make the analysis scale to sets of thousands of ANDROID apps, we implemented a two-step process that regards apps as features of a mobile device: First, a feature-based strategy analyzes each app in isolation and extracts information on usage of private data and on inter-app communication; Second, we use a family-based strategy to analyze communication of private data across multiple apps, based on the information gained in the first step. To optimize the performance of the second step, it can be run on partitions of the app set in parallel. This partition-based version of the second step combines the feature-based and family-based strategies. At the core of the second step (with and without partitions) lies a variability-aware data structure, which is responsible for the improved scalability. Analysis strategies that are targeting configurable systems are not typically used in the app-communication analysis domain, so this is a novelty in itself.

In Section 7.1, we introduce the problem of inter-app data leaks in the ANDROID ecosystem and motivate our analysis. In Section 7.2, we provide background on ANDROID apps, of data leaks between apps, and how leaks can be detected with existing tools. In Section 7.3, we describe the variability-aware data structure for representing inter-app data flows. In Section 7.4

we describe our implementation as a combination of a feature-based and a family-based analysis, and how these strategies are represented in the PLA cube. In Section 7.5, we describe our evaluation of the implementation based on various community benchmarks. Finally, we discuss threats to validity of our results in Section 7.6 and related work in Section 7.7.

7.1 Motivation and Scenario

The growing popularity and adoption of mobile devices—such as smartphones and tablets—has led to a tremendous rise of mobile apps. The size of Google's Play store increased from 70 000 apps in July 2010 to more than 1.6 million apps in July 2015 [App15b]. By January 2014, Apple's app store offered more than one million apps [App14] and had a yearly revenue of $10 billion. The large number of apps available and the increasing diversity of mobile devices lead to very different sets of apps installed on mobile devices today.

Privacy of data in mobile devices is an increasing concern. While apps often process private data, such as passwords, device identifiers, or position data, they also commonly possess unlimited access to communication channels, which may lead to leaks of private data. To prevent leaks of private data, mobile operating systems employ a range of methods, such as encapsulation of apps, dedicated communication mechanisms (e.g., Android Intents), a permission system for accessing sensitive data, and manual review of apps before they are added to the app store. Additionally, various analysis techniques have been developed to detect so-called *tainted data flows*—flows of data from private sources to untrusted public sinks inside an app [ARF^{+}14; LBB^{+}15].

inter-app leaks

Yet, as apps are allowed to communicate with each other, a *combination* of apps can create a private-data leak even if individual apps are considered safe [CPGW11; PWM^{+}11; SBG^{+}13]. For instance, an app could read the device's current location and send it—accidentally or maliciously—to a second app, which then forwards it via the Internet to an untrusted party. Such scenarios are hard to detect as they could in principle involve a chain of many apps [KFB^{+}14]. Malicious apps can even intercept or eavesdrop on unsecured communication between apps.

The presence of critical inter-app data flows depends on the set of apps installed on a device. Consider an accidental private-data leak, where an app sends private information (e.g., a picture) to apps that can display it. If multiple target apps are installed, most systems display a choice dialog, possibly creating awareness for a potential private-data leak. When only one alternative, potentially malicious app is present, communication occurs without user interaction. Consequently, all possible combinations of apps of a given set would need to be verified to detect inter-app leaks, whether accidental or malicious. Even without finding actual leaks, detecting apps or app combinations that forward data among apps is important, because such apps have a high risk to be exploited for realizing data leaks.

Unfortunately, inter-app data-flow analysis is expensive and does not scale well to larger app sets or even to a whole app store. There are two reasons for the poor scalability: First, many apps send and receive standard message types, which leads to substantial numbers of flows; second, the representation of flows is prone to a combinatorial explosion in the number of apps, since many apps send similar messages (many apps are also cloned or use common code [MAN+14; VGN14]). So, installing a new app may double the number of inter-app flows. scalability

Recent taint-analysis tools for Android [ARF+14; KFB+14; LBB+15; WROR14] are reasonably precise in detecting critical data flows, tackling all the peculiarities of Android apps (e.g., permissions, Android API, intents), but they do not scale well to large sets of apps. We argue that the limitation mainly lies in the representation of inter-app data flows that does not explicitly consider *variability* [BPT+14]—an app can be installed or not, thereby influencing the global data flows that exist. Instead of duplicating detected flows, variability inside flows should be modeled explicitly. We view apps as features of an Android device and adopt the feature-based and the family-based strategy discussed in Chapter 3, which incorporate variability to reduce redundancies and avoid a combinatorial explosion. In the context of this thesis, our analysis serves as an example of how configurable-system analysis strategies can be used to enable large-scale analyses.

We present a variability-aware approach for large-scale inter-app data flow analysis implemented in our tool Sifta. Our approach aims at analyzing inter-app communication, *before* concrete combinations of apps are installed by a customer. The long-term vision is to move analysis from the mobile device to the app store, checking all possible combinations of apps. Furthermore, the underlying data structure is designed in a way that allows incremental analysis, avoiding the re-calculation of data flows when new apps are added.

Our approach relies on a graph-based data structure representing flows annotated with presence conditions that denote the presence and absence of apps. We extended and combined existing tools that analyze data flows inside individual apps (feature-based), to use and aggregate their results creating a graph that efficiently represents inter-app communication (family-based). We traverse this graph to search for inter-app data flows that propagate private data (standard taint-propagation analysis). We demonstrate the scalability of our approach on two third-party benchmarks and a set of 51 935 analyzed real-world apps we mined from the Google Play app store (details on the mining process in Section 7.6). At the same time, our tool maintains an accuracy that is similar to existing tools focusing on intra-app analysis, likewise evaluated with two third-party benchmarks and with our own benchmark IACBench (available on the supplementary website).

To support the steady growth of app stores, our tool supports an *incremental* generation of the inter-app communication graph: When new apps are added or changed, the graph can be updated with information for such apps, instead of generating a new graph. We also use the incremental generation to implement a family-based analysis on app groups (feature groups) which enables experiments on a machine with otherwise insufficient main memory (Section 7.5.2). This analysis strategy is represented by a point between **D** and **H** in the PLA cube (Chapter 3). It shows that, in this scenario, combination of strategies enables analyses that were not possible otherwise.

Sifta, links to all other tools, and information on how to replicate our results are available on the supplementary website.

7.2 Background

Next, we introduce app communication in Android and discuss existing analysis strategies and their limitations. We distinguish between inter-component communication—when components inside one app communicate—and inter-app communication—when components of different apps communicate.

7.2.1 Android Apps and the Intent Mechanism

Android apps are delivered in so-called Android *application packages* (APKs) and consist of multiple components that communicate with each other. Components can be GUI elements (*activities*), shown to the user, or non-visible elements that process or store information (*services*, *broadcast receivers*, and *content providers*). Components have a dedicated lifecycle and are encapsulated. They communicate via dedicated messages, called *intents*[1], both for intra-app (inter-component) and inter-app communication. Intents contain various pieces of data, such as routing and payload information.

Intents can be *explicit* or *implicit*. The former identify the target component directly using its fully qualified name. The latter describe the minimal capabilities a target component needs to fulfill, which are then matched against the maximal capabilities of components defined in *intent filters*. Such capabilities

[1] Other means of communication (e.g., shared files, native code) exist, but are outside of our scope.

could be the ability to show a URL or to display an image of a certain type. If multiple components of installed apps qualify, ANDROID displays a choice dialog and lets the user select. Usually, intents pass information to other components. However, they can also query information (e.g., user information from a data-storage component), initiating an information flow back to the intent source.

7.2.2 Intra-App, Inter-Component Communication

Intent-based communication is the primary mechanism for data exchange between components inside an app (intra-app communication). For example, an activity could send data entered by the user to a service that stores or processes the data. Here, an *explicit* intent is typically used, since the receiving component should be unambiguously identified.

Analysis of inter-component communication inside an app is important to detect data flows that leak private data by accident (without intention of the developer). For example, a developer of a popular ANDROID app might want to analyze her own app to confirm that private user data are not passed to third-party components used in the app. In this intra-app scenario, the set of components is known.

Several analysis tools address this scenario. One comparatively precise tool is ICCTA [LBB+15], which relies on FLOWDROID [ARF+14] and EPICC [OMJ+13]. ICCTA composes all components of an app into one "super" component encoding all the flows. A challenge is to connect components—that is, mapping intent calls of one component to incoming intents of another component. Therefore, the parameters of an intent object, which is dynamically instantiated at run time, need to be known and matched to intent filters. For this purpose, EPICC performs a static analysis to retrieve the intent parameters. Once the "super" component is created, it is analyzed with FLOWDROID, a precise interprocedural data-flow analysis tool. The intra-component data flows reported by FLOWDROID connect sources and sinks, which are ANDROID API methods, intent calls, or incoming intents.

7.2.3 Inter-App Communication

Communication between apps is realized using the same intent-based mechanism as for intra-app communication. The main difference is that the set of installed apps is not pre-determined. An implicit intent can be processed by different apps (e.g., different e-mail clients) in different mobile-device configurations, with different implications for data privacy.

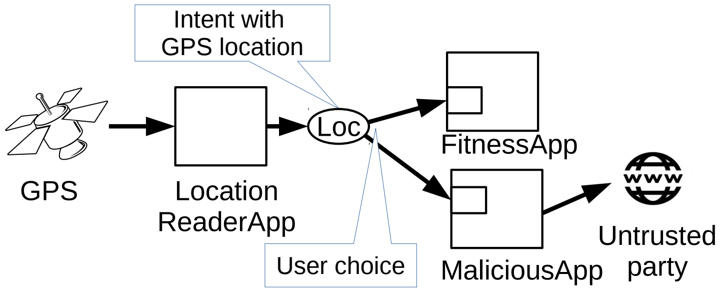


Figure 7.1: Inter-app communication example, where GPS location information is forwarded to untrusted receivers. Large rectangles represent apps, small rectangles represent intent filters, and the ellipse represents an intent.

Figure 7.1 shows a simple inter-app communication. The *LocationReaderApp* reads the current location from the GPS device and sends it via an intent (*Loc*). If installed, each of the two apps *FitnessApp* and *MaliciousApp* can receive the intent (determined by their intent filters, shown as little rectangles). If the latter obtains the data, they are forwarded over the Internet to an untrusted third party.

insufficient permission system

Similar scenarios have been reported in the literature [BR14; CPGW11; KFB+14; PWM+11; SBG+13]. Ideally, ANDROID's permission system should prevent apps from accessing private data without user consent. However, ANDROID permissions are not sufficient as commonly stated in the literature (see Section 7.7). In our scenario in Figure 7.1, *MaliciousApp* does not have the permission to read GPS data directly from the ANDROID API. However, it can receive the intent that is sent from *LocationReaderApp*, which contains (among other information) the private GPS data. Receiving this intent requires no special permission. Once *MaliciousApp* receives an intent from *LocationReaderApp*, it extracts the GPS data and leaks it to the internet. It does not matter whether *LocationReaderApp* sends the data out via any intent, or has a component that is accessible via an intent, and whether both happens accidentally or whether the app was maliciously developed to enable this scenario. This problem is generally known as *permission re-delegation* [CPGW11; PWM+11] (a variant of the *confused-deputy problem* [Har88]).

Analyzing inter-app communication is important for maintainers of app stores or pools. It is desirable to ensure that each possible combination of apps respects privacy of user data and that no inter-app data-flow leak exists. In this chapter, we show that analysis strategies for configurable-systems can help to make such analyses feasible for large sets of apps. Even without actual leaks, it is desirable to identify high-risk apps that forward data [vRBS+15], which in combination could be exploited for private-data leaks in the future.

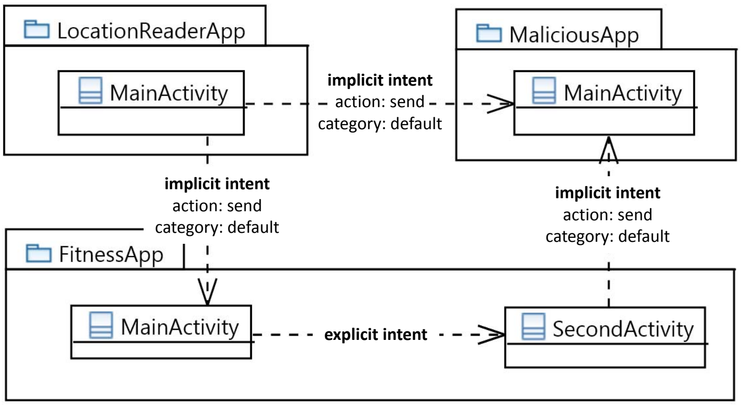


Figure 7.2: Detailed example of inter-app communication

An app-store scenario is more complex than intra-app communication, since apps can be present or absent on a device (resulting in different global flows—this is why we pursue a variability-aware approach). Furthermore, apps are regularly added, removed, and updated. Thus, analysis results of inter-app communication should be kept updated after each change in the pool, ideally, without the need of an expensive re-analysis of the entire pool. We pursue an incremental approach (described in Section 7.4) to avoid this expensive re-analysis.

To analyze inter-app communication, one could, in principle, use tools that have been developed for intra-app analysis, because these communication types both rely on intents. For example, DIDFAIL [KFB+14] uses FLOWDROID to obtain data flows within each component in each app. Based on the intent parameters obtained with another tool, called EPICC, DIDFAIL connects the possible outgoing intents to intent receivers and builds a global communication graph involving all apps.

7.2.4 Limitations of Existing Tools

Both ICCTA and DIDFAIL rely on the fact that the set of components is known, invariable, and rather small. ICCTA's approach would cause scalability problems when inter-app communication is analyzed, as the generated "super"-component easily becomes large, with many—likely redundant—flows. But the ICCTA developers focus on intra-app communication in their experiments anyway [LBB+15]. In contrast to ICCTA, DIDFAIL addresses inter-app communication explicitly. Yet, it stores detected flows in a list-like data structure that is not variability-aware. As a result, DIDFAIL does not exploit redundancies between the flows, harming scalability.

To understand how a larger app set can influence the number of flows, consider the following thought experiment, illustrated in Figure 7.2. We build a scenario based on the three apps of Figure 7.1: *LocationReaderApp* obtains private data, sending them with a valid intent to *FitnessApp*. However, *MaliciousApp* can also receive the intent—its presence establishes a data flow to an untrusted network receiver outside the phone. *MaliciousApp* has the permission to access the Internet, but not to obtain GPS data from the ANDROID API. Furthermore, we extend the scenario slightly, adding (1) a typical internal data flow between two components inside *FitnessApp* and (2) another accidental leak from *FitnessApp* to *MaliciousApp*, so that we have two leaks from a private source to a public sink.

Now, consider a larger scenario, where we have an additional alternative app for each of the three apps. The alternative apps have roughly the same functionality (the same data sources and sinks, and the same intents), but they are implemented by different developers. In this scenario, the number of flows increases: for example, both variants of *LocationReaderApp* can send information to both variants of *MaliciousApp*. Now, there are 12 leaks in total. In general, the number of flows has a cubic growth in the number of apps of each kind, which shows that an efficient inter-app analysis has to exploit redundancies between flows. Even though this thought experiment shows an extreme example, our experiments (Section 7.5.2) showed results for DIDFAIL's scalability that are similar the results of the thought experiment.[2] A key insight is that the problem lies in the representation of the underlying graph and of the flows. DIDFAIL does not address sharing between apps or redundancies between flows.

The limitations of existing tools motivated us to develop our own tool, called SIFTA, that addresses these challenges when analyzing inter-app communication. We reused parts of DIDFAIL's code, but completely re-implemented the graph synthesis and the identification of tainted data flows.

7.3 Representing Inter-App Data Flows

The communication between ANDROID components is typically analyzed in two steps. First, information on individual apps (e.g., using a static code analysis)

[2] Our experiments are based on the DIDFAIL variant published at the SOAP workshop [KFB+14]. In a more recent publication, the authors describe an improvement of DIDFAIL [BFK+15], however the focus of the improvement seems to be DIDFAIL's accuracy, not its scalability. Therefore our analysis of DIDFAIL's scalability does still hold even though the accuracy of the new DIDFAIL version is better than suggested in our experiments (Section 7.5.1).

is collected and stored in a suitable data structure. Because we consider apps as features of an Android device, this is a feature-based step. Second, the stored data is analyzed. This second step is family-based, because the data flows for the entire configurable system is stored in a single data structure. This two-step process avoids the need to deal with app internals (e.g., source code) in the second step. The key factor is how to abstract from app internals and how to store the abstracted data. In the next sections, we discuss which data from apps and intents we consider, we discuss how the data is represented in a basic data structure (reflecting the structure used by DidFail [KFB^{+}14]), and we introduce our lifted, variability-aware data representation.

7.3.1 Design Considerations

To represent app communication in the graph, we need to keep the information necessary to determine whether an intent can be accepted by a given component. Android makes this decision based on an intent's metadata and on a component's intent filter. An intent is defined by its sender component, an action key, a list of categories, and a mime type.[3] An intent filter is defined by the component for which it controls incoming intents, by a list of action keys, a list of categories, and a list of mime types. Based on this information, Android matches intents with intent filters to deliver the intent to a component [DB12].

In Android, private data originate either from system API calls (e.g., location data) or from the user (e.g., a password entered in a text field). Likewise, to send data to untrusted receivers (private sinks), one has to call API functions (e.g., to send SMS, open network connections, or write into log files). To identify such private sources and public sinks, we rely on lists with API function signatures from previous work [ARB13].

Based on these definitions, we can build an inter-app data-flow graph. We chose a directed graph as representation, in which a node represents either a start or end point of a potentially critical data flow, that is, from a private source to a public sink, or an intent that forwards information. In this graph, we already abstract from many implementation details of the apps. For example, we do not consider sources of non-private data (e.g., the current time). Edges represent apps that receive and process intents, receive data from private sources or/and forward data to public sinks. We chose this node–edge mapping to avoid dangling edges and because intents are the central entities of inter-app communication.

[3] The mime standards define content types (e.g., JPEG, GIF, or AVI) of data attached to communication messages. They are used in e-mail and http protocols. Clients use the mime type to determine how attached data should be opened.

7.3.2 DidFail's Representation

Next, we define a graph data structure that is used by DidFail and that is not variability-aware. We denote the set of all components of apps with *Comp*, the set of all intents with *Int*, the set of all private source with *Priv*, and the set of all public sinks with *Pub*. The graph is defined as a set V_{DF} of nodes with $V_{DF} \subseteq \mathit{Int} \cup \mathit{Priv} \cup \mathit{Pub}$ and a set E_{DF} of directed edges with $E_{DF} \subseteq V_{DF} \times V_{DF} \times \mathit{Comp}$. V_{DF} contains all intents, private sources, and public sinks. For each component *comp* that receives data from a private source or intent *src* and that delivers data to an intent or public sink *sink*, the set of edges contains the triple $(\mathit{src}, \mathit{sink}, \mathit{comp}) \in E_{DF}$. A path through the graph represents a potential data flow from a private source through a number of components (possibly across multiple apps) to a public sink.[4]

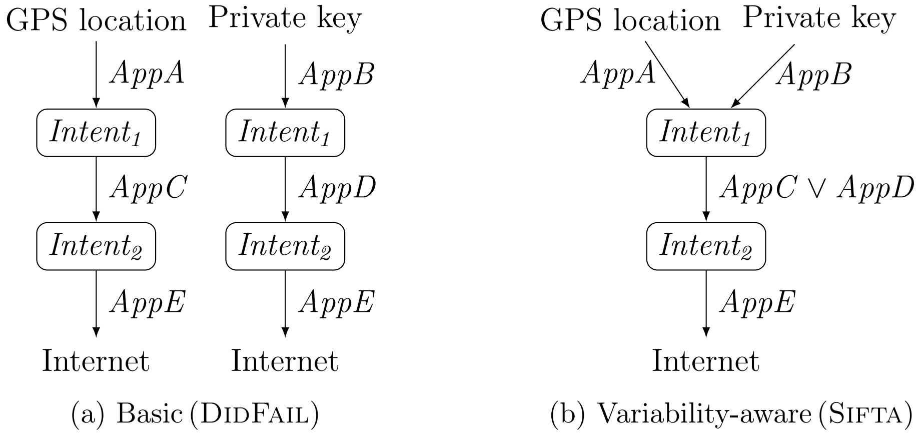
Recall our thought experiment from Section 7.2.4: With an increasing number of apps, the graph quickly becomes very large and its generation expensive. The reason is that often different apps have (partly) similar functionality. For example, they receive data from the same sources (*Int* or *Priv*) and send data to the same sinks (*Int* or *Pub*). Thus, the graph has many edges that differ only in the app component, such as the edges $(a, b, c_1) \in E_{DF}$ and $(a, b, c_2) \in E_{DF}$. Figure 7.3a shows an example with two flows of private data (GPS location and private key) to a public sink (Internet). The second edges of the flows have the same source (*Intent₁*) and the same sink (*Intent₂*), and only differ by the inner component (middle edges).

7.3.3 Variability-aware Representation

In Sifta, we represent flows within and across apps in a variability-aware fashion. The difference is that each edge in the graph is annotated with a presence condition (condition for it's presence in the system). This presence condition is a predicate over the (optional) apps in the pool. Each path in the graph represents a *variational flow* corresponding to multiple *concrete flows* (e.g., flows in the DidFail representation).

We define the set of edges such that each edge holds a set *comps* of components (or, equivalently, a predicate over *Comp*): $E_{VA} \subseteq V_{DF} \times V_{DF} \times \mathcal{P}(\mathit{Comp})$. Instead of mapping each edge to a component, we now map each edge to a *set* of components (or, equivalently a predicate over component identifiers). The semantics is that an edge (a, b, comps) is present in the graph (or on the mobile device) iff one of the components in *comps* is installed. Finally, since

[4] The flow is only a potential flow, as our analysis is static and can produce false positives (as taint analysis in general).



(a) Basic (DidFail)

(b) Variability-aware (Sifta)

Figure 7.3: Example of a basic and a variability-aware inter-app flow representation

a component is automatically present when its app is installed, we only store app names on edges (instead of component names).

Figure 7.3b shows the same scenario as Figure 7.3a, but using our variability-aware representation. The two edges from *Intent1* to *Intent2* are replaced by one, which has a presence condition denoting which components need to be installed to enable the flow.

The variability-aware representation is efficient when the app set contains many inter-app flows that share common parts (intents or partial flows). Such sharing can be caused by common intents used by many apps (e.g., intents the ACTION_VIEW action key) or by code in differently named components that process information in the same way (e.g., through code duplication [MAN+14; VGN14]).

7.4 Implementation

We implemented our approach in the tool Sifta. It is based on code from DidFail. Sifta extends DidFail with concepts discussed in Section 7.3.3 and additionally introduces handling of services and broadcast receivers, which are types of Android components that are not covered in DidFail.

two-phase approach

Sifta uses a two-phase approach. In the first phase, it uses the tools FlowDroid and Epicc to analyze one app at a time (feature-based). FlowDroid generates information on (i) which intents contain private information and (ii) which information from intents is sent to public sinks of an app. Epicc provides detailed information on the data of the intents, which is necessary to match them to intent filters of other apps (see Section 7.3.1). The second, family-based phase of SIFTA implements intent matching procedures as described in the ANDROID API to generate an inter-app data-flow graph, as described in Section 7.3.3. Furthermore, the second phase runs a simple taint-propagation analysis on the generated graph to detect inter-app privacy leaks. This phase uses the FLOWDROID and EPICC output from the first phase and the manifest files of the apps, containing details on intent filters (see Section 7.3.1). In addition to DIDFAIL's matching criteria, SIFTA implements matching of mime types as specified in the ANDROID API. Finally, note that the first phase may fail (cf. Section 7.5.2) on real-world apps, often because FLOWDROID or EPICC hit timeouts. Improving these third-party tools is well beyond the scope of this thesis. Also, some failures are to be expected, because we use static-analysis tools on real-world apps that might actively prevent analysis by code obfuscation. Still, our two-phase design allows us to easily use results from other tools in the first phase.

Paths in the variability-aware graph represent multiple private-data leaks when edges have alternative apps (i.e., a presence condition with more than one app). In contrast to presence conditions in configurable-system analysis (Chapter 4), presence conditions in the graph are simple (disjunctions), such that we do not need SAT queries to generate the graph or to derive feasible flows from it.

incremental generation

Furthermore, we implemented an incremental graph-generation procedure—a functionality that we needed experiments with large sets of apps (e.g., 51 935 apps in Section 7.5.2). Since the main computation effort lies in the first phase, we support reuse of already computed partial results. This functionality of SIFTA enables us to use the second phase of the analysis on groups of apps (feature groups) and enable analysis of larger app sets as discussed in Section 7.5.2. We support reuse of two types of intermediate results, as illustrated in Figure 7.4. The apps *AppA* and *AppB* are analyzed for the first time. Two other apps were analyzed before (results from phase 1 are reused), and two old graphs with information about more apps exists already. In the first phase, *AppA* and *AppB* are analyzed by FLOWDROID/EPICC. In the second phase, SIFTA uses the newly generated results and the reused results and integrates them with the existing graphs. In the end, SIFTA produces an updated graph containing all the edges of the old graphs and the new edges introduced by the new apps.

This persistence and reuse of results enables SIFTA to analyze large-scale, evolving sets of apps in short time. If only few apps change, SIFTA does not need to analyze the entire set of apps from scratch, but can reuse old results, if they are still valid (when the apps have not changed). To update a graph,

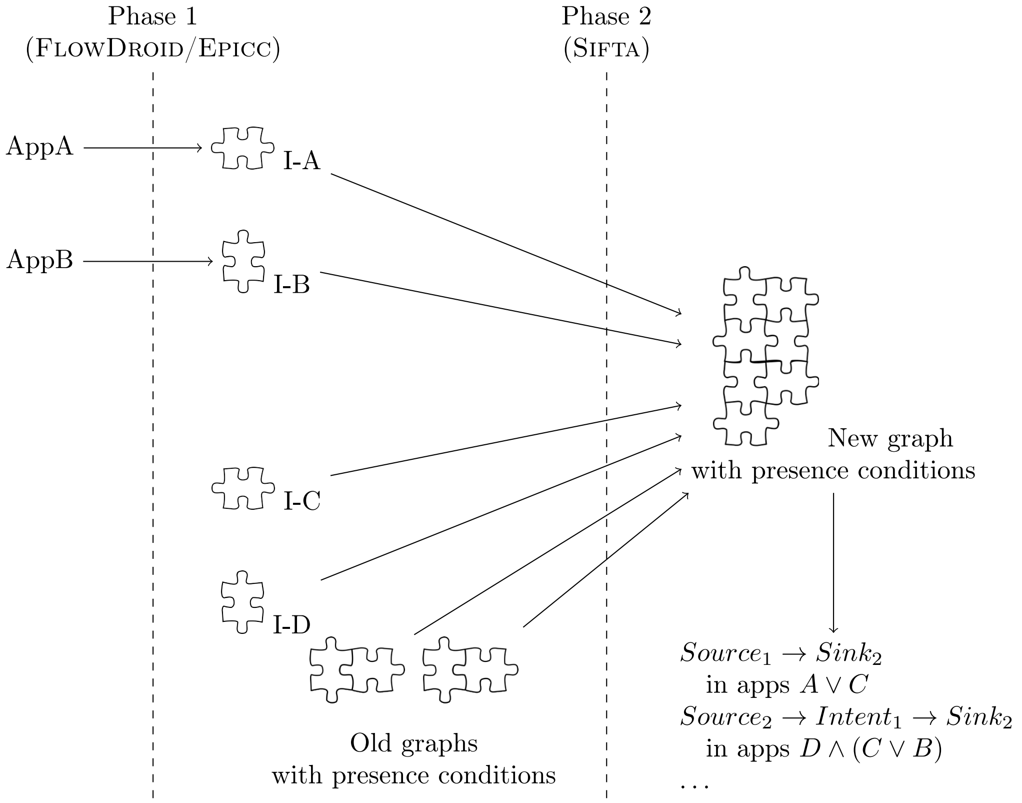
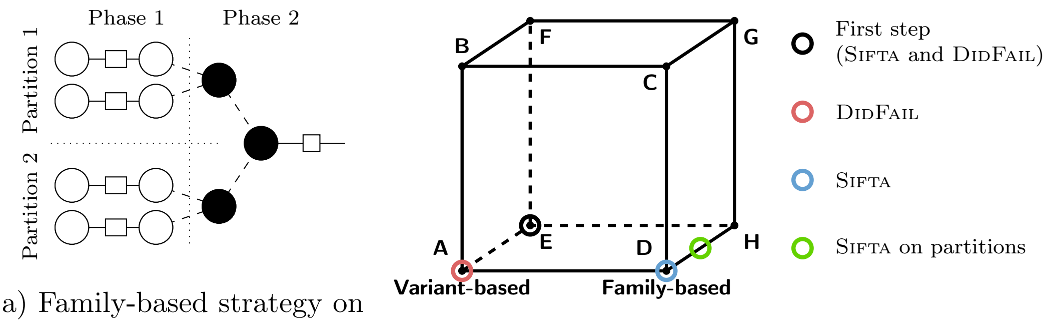


Figure 7.4: Sifta's inter-app analysis. I_A, I_B, I_C and I_D represent intermediary results generated by the first phase. I_C and I_D are reused from a previous run of the analysis. The intermediary results are added to the old graph, which is also reused from a previous analysis run.

we would remove all apps for which we have updated information (delete the app from all presence conditions), and then integrate the FlowDroid/Epicc results for the updated apps.

taint propagation

Once the communication graph has been generated, it can be used in various ways. An example, which we implemented in Sifta, is a standard taint analysis that reports all paths from sources to sinks in the graph. To generate all paths through the graph, we use a depth-first exploration algorithm on every source node of the graph. Source nodes in the graph correspond to actions that query private data and sinks correspond to actions that send data over potentially unsecure connections. Therefore, paths correspond to potentially malicious data flows. This analysis is a taint-propagation analysis as the (*tainted*) private data is forwarded along the path until it reaches a sink. The apps on edges along the path constitute the presence condition of the data flow.



(a) Family-based strategy on partitions of the app set

(b) Analysis strategies represented in the PLA cube

Figure 7.5: Illustration of the evaluated analysis strategies

Relation to the PLA model Figure 7.5a illustrates how the analysis strategies of Sifta and DidFail are represented in the PLA model. Each analysis strategy consists of two steps which are represented by two points in the PLA cube. The first step of each strategy is a feature-based analysis where each app is analyzed in isolation (with FlowDroid and Epicc). This step is represented by point **E** in Figure 7.5b.

In its second step, DidFail combines the information from the first phase and builds leak flows across app boundaries. However, it does not exploit sharing between these flows, so this is essentially a variant-based analysis (point **A**, marked red in Figure 7.5b). Sifta supports two alternative modes for the second phase. In its basic mode, it analyzes the information from the first phase in a family-based strategy (marked blue in Figure 7.5b), sharing common subpaths in a variability-aware data structure. For large app sets, graph generation with this strategy becomes expensive (e.g., memory consumption, cf. Section 7.5). In such cases Sifta allows to partition the app set and run the second phase on each partition separately (also in parallel). Figure 7.5a illustrates this family-based and feature-group based strategy (point between **D** and **H**, marked green in Figure 7.5b). Once graphs for all partitions have been generated, Sifta merges them and generates a global communication graph representing all apps.

7.5 Evaluation

In a series of experiments, we evaluated the accuracy and scalability of our approach. We compared Sifta to the other state-of-the-art tools DidFail and IccTA. Our primary goal is to evaluate whether the variability-aware analysis strategies of Sifta improve scalability over the other tools. However, we also

Table 7.1: IACBench test cases

	Test case	Description
Basic	startActivity	intent from Activity to Activity via startActivity
	startService	intent from Activity to Service via startService
	bindService	intent from Activity to Service via bindService
	sendBroadcast	intent from Activity to BroadcastReceiver via sendBroadcast
	sendOrderedBroadcast	intent from Activity to BroadcastReceiver via sendOrderedBroadcast
Advanced	multipleIntents	two identical intents originating from the same source and targeting the same sink
	loop	intent from Activity to Activity, but the first Activity can also receive its own intent, creating a loop
	intentChain	intent from Activity1 to Service, to Activity2, to Activity3, back to Activity2 (result), to BroadcastReceiver
	identicalIntentFilter	intent sent to three different components (Activity, Service, BroadcastReceiver), each of which has the same intent filter

have to show that the accuracy of Sifta is comparable to state-of-the art tools; otherwise the value of Sifta's results would be unclear.[5]

In Section 7.5.1, we discuss our evaluation of the accuracy of Sifta on benchmark sets comprising a total of 44 test cases with inter-component and inter-app leaks (Experiment 1). Yet, accuracy is only a necessary condition and highly relies on the underlying data-flow–analysis tools we use.

The potential leaks we detected in real-world apps can be used to inspect and fix apps. While this work is orthogonal to our approach, our graph can be used to identify high-risk apps that enable many flows as we demonstrate in a related publication [vRBS$^+$15].

7.5.1 Accuracy: Experiment 1

In the first experiment, E1, we measured the accuracy of our approach by calculating precision and recall of detected leaks using a ground truth of established, third-party community benchmarks and our own hand-crafted benchmark. To understand the accuracy that is currently achievable with state-of-the-art tools, we compare our results to those obtained by IccTA and DidFail. Overall, we analyzed three different sets of apps:

- IACBench contains 9 app sets (two apps per set) created by us to cover basic (intents with and without results, comprising activities, services, and broadcast receivers) and advanced (e.g., loops, intent chains) inter-app flows. We show details of the IACBench tests in Table 7.1.

[5] Without an accuracy evaluation, a (very fast) random-result generator would also be acceptable.

- ICC-Bench[6] contains 9 apps developed by the authors of Amandroid [WROR14] with intra-app flows.
- DroidBench[7] comprises 23 apps testing inter-component communication (provided by the IccTA authors [LBB+15]) and 3 sets of apps testing inter-app communication (provided by the DidFail authors [KFB+14]) among many more apps not relevant for our approach.

Our own benchmark IACBench contains test cases with undesired data flows via implicit intents across apps from the source TelephonyManager.getDeviceId to the sink Log.i. IACBench is available on our supplementary website. For the third-party benchmarks ICC-Bench and the parts of DroidBench that we used for our evaluation, we refer to the literature: Amandroid [WROR14], DidFail [KFB+14], and IccTA [LBB+15].

All benchmarks comprise apps developed to test whether analysis tools capture the specific means of communication. The apps are much smaller and cleaner than real apps and are, thus, ideal to compare the accuracy of different tools.

Hypothesis In this experiment, we compare the results of Sifta on the test benchmarks against the results of IccTA and DidFail. Our hypothesis is that Sifta achieves the same result as IccTA and DidFail on most tests.

Methodology and setup We ran Sifta and DidFail on all benchmarks and measured precision and recall. While Sifta focuses on inter-app communication, it can still analyze intra-app flows. Thus, we do not only compare against DidFail, but also against IccTA, which is specialized on inter-component, intra-app communication. Consequently, we can run IccTA only on the ICC-Bench and DroidBench-ICC benchmarks, not on IACBench.

We ran the experiment on a Ubuntu 14.04 workstation with an Intel Xeon Processor X3470 @ 2.93GHz. Timeouts and memory consumptions were not an issue for these rather small test cases.

Results Table 7.2 shows precision and recall of Experiment 1. Next, we discuss the different benchmarks, emphasizing on the test cases where Sifta produces worse results than DidFail or IccTA.

IACBench focuses on *inter*-app communication. Thus, we could not evaluate IccTA on this benchmark. Sifta solved all tests correctly. DidFail cannot be applied to four test cases, because it lacks support for Services or Broadcasts.

[6] Obtained from the authors of Amandroid.

[7] `http://github.com/secure-software-engineering/DroidBench/`

Table 7.2: Results of Experiment 1: Accuracy evaluation (IccTA results according to [LBB$^+$15])

benchmark	test case	DidFail	Sifta	IccTA
IACBench	startActivity	+	+	n/a
(basic)	startService	n/i	+	n/a
	bindService	n/i	+	n/a
	sendBroadcast	n/i	+	n/a
	sendOrderedBroadcast	n/i	+	n/a
(advanced)	multipleIntents	+	+	n/a
	loop	+	+	n/a
	intentChain	⊖	+	n/a
	identicalIntentFilter	+	+	n/a
ICC-Bench	Explicit1	⊖	+	+
	Implicit1	+	+	+
	Implicit2	+	+	+
	Implicit3	+	+	+
	Implicit4	+	+	+
	Implicit5	+	⊖	+
	Implicit6	+	⊖	+
	DynRegister1	⊖	⊖	+
	DynRegister2	⊖	⊖	⊖
DroidBench	startActivity1	⊖	+	+
(ICC)	startActivity2	⊖	+	+
	startActivity3	⊖	+	+
	startActivity4	⊕	⊕	−
	startActivity5	⊕	−	−
	startActivity6	−	⊕	−
	startActivity7	−	⊕	⊕
	startActivityForResult1	⊖	+	+
	startActivityForResult2	⊖	⊖	+
	startActivityForResult3	⊖	⊖	+
	startActivityForResult4	⊖	+	+
	startService1	n/i	+	+
	startService2	n/i	+	+
	bindService1	n/i	+	+
	bindService2	n/i	⊖	+
	bindService3	n/i	⊖	+
	bindService4	n/i	+	+
	sendBroadcast1	n/i	+	+
	stickyBroadcast1	n/i	+	+
	insert1	⊖	+	+
	delete1	⊖	+	+
	update1	⊖	+	+
	query1	⊖	+	+
(IAC)	sendBroadcast1	n/i	+	n/a
	startActivity1	+	+	n/a
	startService1	n/i	+	n/a

true positive:	+	(analysis reported an existing leak)
false positive:	⊕	(analysis reported a leak that does not exist)
true negative:	−	(no leak and no leak reported)
false negative:	⊖	(analysis misses an existing leak)
not applicable:	n/a	(intra-app tool on inter-app scenario)
not implemented:	n/i	(DidFail on services or broadcasts)

On some apps in the ICC-Bench benchmark set, Sifta failed to report private-data leaks. In particular, in the test cases Implicit5 and Implicit6, Sifta reported no flows, while DidFail correctly reported them. The reason is a limitation of the underlying tool Epicc and of DidFail, which ignores the faulty Epicc output. In both cases, the flows are enabled by mime types set on the intent objects in the code (Intent.setDataAndType). Apparently, Epicc does not handle this function as it does not include the mime type in its output. Based on this output, Sifta assumes that no mime type is given and the intent does not match the intent filter in the test case. DidFail does not implement a test for mime types and therefore correctly reports a flow. Furthermore, in the test cases DynRegister1 and DynRegister2, intent filters are registered dynamically and not declared in the manifest file. Epicc does not find such intent filters, which are therefore not visible to Sifta or DidFail. IccTA fails to detect the leak in DynRegister2 because the app uses string operations, which cannot be parsed by IccTA [LBB+15].

DroidBench is a much larger benchmark that tests many possible communication paths. Table 7.2 shows that Sifta reports correct results on 75% of the test cases. This is much more often than DidFail (43%), but not as often as IccTA (94%). The test case startActivity4 has an intent that uses an URI scheme (http:) that is not listed in the test's intent filter. Therefore, the intent does not match the filter. Sifta does not test for URI schemes, because this information is not provided by Epicc and FlowDroid. The tests startActivity6 and startActivity7 check whether information retrieval from an intent is handled correctly. In these tests, an intent with private information is accepted by an intent filter, but instead of the private information, other information is retrieved from the intent. The information available to Sifta contains no details on *which* information is retrieved from an intent. As long as the intent with private information is accepted and information from that intent is sent to a public sink, Sifta reports a flow. The bindService test cases transfer private data via an intent to a service that logs the private data. In the bindService2 and bindService3, FlowDroid did not report that an intent is sent, therefore the flow is invisible to Sifta. The tests startActivityForResult2 and 3 failed because Sifta cannot handle some aspects of the return communication in startActivityForResult intents. We intentionally omitted these aspects for scalability reasons (see Section 7.5.2). Finally, the inter-*app* communication tests (IAC) of DroidBench were all solved correctly by Sifta.

These results demonstrate that our variability-aware tool Sifta produces more accurate results than DidFail. Yet, it is less accurate than IccTA, which was to be expected as IccTA, for each test, combines all components and analyzes them in one run. Sifta has to rely on the necessarily filtered information gained in separate per-component analyses, but this is exactly the lever that enables large-scale inter-app analysis.

Surprisingly, SIFTA has some wrong results where DIDFAIL's results are correct. This is not caused by SIFTA's graph reduction (which does not influence the set of reported flows), but by additional matching criteria (for the mime types, cf. Section 7.4) that we implemented.

Hypothesis discussion The hypothesis of Experiment 1 is that SIFTA reports the correct result on most benchmarks. For the IACBENCH tests, SIFTA is correct in all cases. On some tests of the ICC-BENCH and DROIDBENCH benchmarks, SIFTA reports false results. However the majority of results is still correct (6 of 10 tests and 19 of 26 tests, respectively). Therefore, Experiment 1 confirms our hypothesis, even though ICCTA achieves a higher accuracy on intra-app tests.

7.5.2 Scalability: Experiments 2–4

In this section, we describe our evaluation of how configurable-system strategies influence scalability of inter-app communication analysis. We compared DIDFAIL, which uses a feature-based and a variant-based strategy but is not variability-aware, with our variability-aware tool SIFTA, which uses the same feature-based strategy as DIDFAIL and combines it with a family-based strategy on feature-groups. We assume that a scalability difference between SIFTA and DIDFAIL is mainly caused by the different strategies because SIFTA's implementation is based on code of DIDFAIL and both tools use the same underlying tools, programming languages, and input files. To evaluate the scalability of SIFTA and DIDFAIL, we used three sets of real-world apps:[8]

- Experiment 2 (E2): ICCRE is a set of 523 real apps coming with ICCTA. These apps leak private user data through inter-component (and intra-app) communication [LBB+15].
- Experiment 3 (E3): MALGENOME is a set of 1260 real apps published by the ANDROID Malware Genome Project [ZJ12]. They are known to be malicious, 51.1 % harvest user data, not necessarily using inter-app communication.
- Experiment 4 (E4): GOOGLEPLAYSET is a set of 172 779 apps that we randomly downloaded from Google Play, covering various categories and developers. We sought to obtain popular apps that are likely to communicate (see Section 7.6, for details about the download process).

[8] We could not use these apps in the accuracy evaluation (Section 7.5.1) because we don't have baseline results on inter-app communication in these app set.

According to an online statistic on the Google Play Store[App15a], there were 1 500 411 free apps in the Play Store as of Sept. 16, 2015. Our GooglePlaySet accounts for over a tenth of these apps, which is a very large set, compared to related studies.

Hypothesis Our hypotheses for experiments 2, 3, and 4 are (1) that Sifta has a better scalability than DidFail and (2) that Sifta scales to a large sets of apps such as our GooglePlaySet.

Methodology and setup In E2, we compared Sifta against DidFail, however, given DidFail's scalability limitations, we were not able to use it in E3 and E4.

We ran E2 on a Ubuntu machine with AMD Opteron 6386 SE @ 2,8 GHz (32 Cores) and 100 GB reserved RAM. Because DidFail and Sifta are both based on the data-flow information generated by FlowDroid and Epicc, we ran this pre-analysis separately. First, FlowDroid and Epicc analyzed all IccRE apps with a timeout of 10 minutes. This pre-analysis generated results for 324 of the 523 apps because of timeouts and because some apps did not use inter-app communication. Then, we let DidFail and Sifta build the data-flow graphs based on this output. For both tools, we measured the time needed both to generate the graphs and to report detected critical flows. To evaluate the scalability of DidFail and Sifta, we generated subsets of increasing size from the IccRE app set. We ran DidFail and Sifta on each subset (without reusing results from smaller subsets).

For E3 and E4 (MalGenome and GooglePlaySet), given their size, we switched to a cluster of 17 nodes, each with an Intel Xeon E-5 2690v2 CPU @ 3,0 GHz (10 cores and 2 hyperthreads per core). We allowed the FlowDroid and Epicc processes to use up to 6 GB RAM and set timeouts at 20 minutes for each process.

Experiment 2 (IccRE) Figure 7.6 shows the results of the scalability experiment on IccRE. Even for only five apps, Sifta generates the data-flow graph faster than DidFail. For larger app sets, the difference between the tools gets larger (speedup of up to 7620). We stopped the experiment for DidFail after analyzing the app set with size 100, as the effect was clear.

A closer look at the output of DidFail and Sifta reveals the reason for this difference in scalability. For the app set with size 100, DidFail generates a data-flow graph with 1610 nodes and 51 709 edges. Sifta's graph has only 51 nodes and 96 edges—illustrating the effectiveness our our compressed variability-aware representation. Even for the largest app set with 324 apps, Sifta's

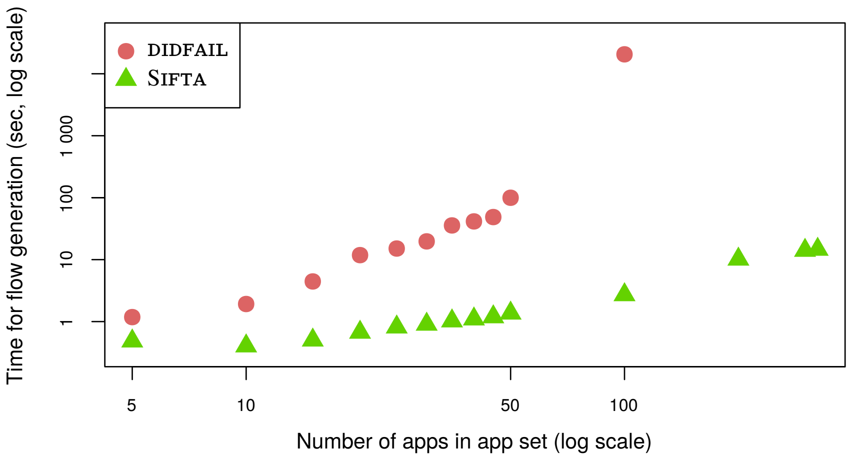


Figure 7.6: Results for SIFTA and DIDFAIL on ICCRE. Both axes have a logarithmic scale.

graph has only 66 nodes. This result shows that there is large potential for storing inter-app data-flow graphs more compact without losing information. Our variability-aware approach achieves this compression and enables efficient analysis of inter-app communication on large app sets.

Experiment 3 (MALGENOME) We analyzed the MALGENOME benchmark set only with SIFTA (DIDFAIL does not scale to this size). The analysis ran in two phases: In the first phase, FLOWDROID and EPICC ran on each of the 1260 apps. This phase is computationally very expensive. It took about 50 hours (sum across all cluster cores). This phase failed on 421 of the 1260 apps due to 20-minute timeouts or other errors outside SIFTA. In the second phase, we applied SIFTA to generate a global graph of inter-app and intra-app communication. This generation took only 26 seconds. We had to drop 9 further apps due to parsing errors on the FLOWDROID or EPICC output.

The resulting graph contained 283 flows representing 839 apps. 248 flows are intra-app flows that go directly from a private source to a public sink. These would also be found by other tools that focus on intra-app communication. However, we also found 35 flows that involve two or more apps and therefore cannot be found with intra-app analysis. The maximum number of apps annotated on an edge is 220 (average is 10), which means that we have a high degree of sharing in the graph. If we would use a tool like DIDFAIL, which is not variability aware, it would produce 220 clones of this edge instead of a single edge. This shows the benefit of our representation even if there are no inter-app data leaks.

Experiment 4 (GooglePlaySet) To evaluate the scalability of Sifta on an even larger app set, we downloaded 172 779 apps from the Google Play store. We then used Sifta to analyze inter-app communication and to build the data-flow graph. Next, we report on the time needed to execute Sifta and on characteristics of the generated graph.

The first phase of Sifta (running FlowDroid and Epicc) was executed on the AMD Opteron Cluster that we also used for E2. This phase generated results for 51 935 of the initial 172 779 apps. The others mainly failed due to FlowDroid timeouts. This phase of the analysis took 1704 days (4.6 years) in total (sum of the times consumed by cluster nodes). We set a timeout of 20 minutes each for FlowDroid and for Epicc. The rather low yield of this phase can be explained by the fact that we rely on research tools and apply them to very diverse set of real-life apps. Although FlowDroid is one of the most precise tools for data-flow analysis of Android apps [ARF+14], improving it to an industrial strength is an effort that was not taken yet.

Next, we collected the results generated by the first phase and ran the second phase of Sifta, to generate the global variability-aware data-flow graph. We executed this phase on the previously described Intel Xeon workstation, because it has not been parallelized so far. We first tried running the graph generation for all apps at once, however the machine's main memory was not sufficient. After loading less than half of the apps, the process already used more than 5.7 GB. Instead, we used the incremental, feature-groups–based graph-generation strategy of Sifta (cf. Section 7.4): We partitioned the results of the first phase into four sets and generated the graph in four steps, as shown in Figure 7.7. We generate a graph for each partition of the app set and then merge these graphs. With this incremental approach, the graph generation took 12 minutes, and each step used less than 3.5 GB RAM. This success of the incremental strategy demonstrates the advantage of using a family-based strategy based on feature-groups as opposed to a pure family-based strategy.

Overall, the graph contains 126 205 potential, variational flows from a private source to a public sink. The graph has 1387 nodes and 5848 edges. The maximum flow length is 8: 5154 of the flows pass through 8 apps before leaking private information. The edge's presence conditions contain an average of 32 (median 3) apps. The maximum of 14 164 apps has an edge that represents a set of intra-app data flows from Bundle.getBoolean to Bundle.putBoolean. This makes sense as these very common API methods can be used to read/write data from/to Bundle objects, which are the payload of intents. Figure 7.8 shows the frequency of presence condition sizes (number of unique apps) in the graph. It shows that these sizes lie between 10 and 200 apps for many edges. That is, each of these edges would be repeated 10 to 200 times in a graph that

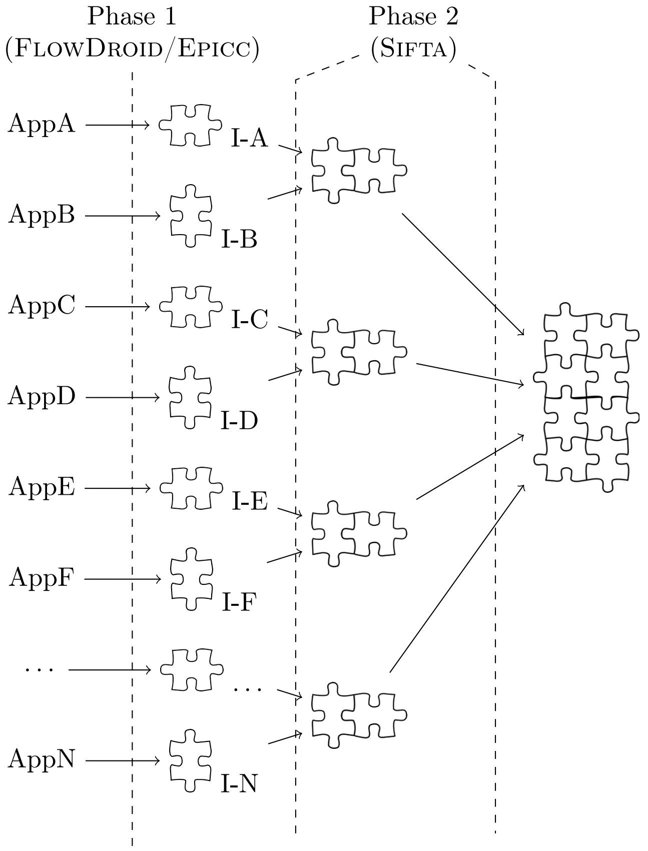


Figure 7.7: Incremental setup for E4. We chose this setup due to memory limitations when building the graph from all apps at once. In the figure, partitions consist of two apps only; in our experiment each partition had a quarter of the 51 935 apps.

does not tap into this sharing potential. For such large app sets, it would be infeasible to generate and store such a graph without variability awareness.

Discussion Experiment 2 showed that Sifta generates the data-flow graph faster than DidFail (speedup of up to 7620). Furthermore, Figure 7.6 suggests that the performance difference between Sifta and DidFail increases with the number of analyzed apps. This confirms our the first hypothesis (Sifta has a better scalability than DidFail). We assume that the scalability difference is due to the variability-aware family-based strategy of Sifta, as this is the main implementation difference to DidFail.

Our second hypothesis, is that Sifta scales to large app sets such as the GooglePlaySet. Experiment 4 confirms that Sifta can generate an inter-

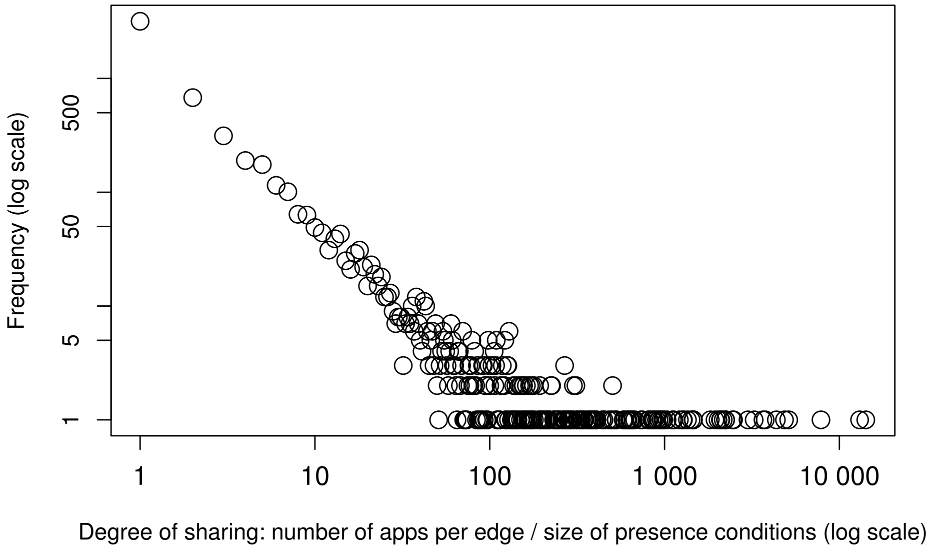


Figure 7.8: Frequencies of presence-condition sizes illustrating the reason for sharing in inter-app flows. The graph shows how often different numbers of apps on edges in the graph occur. Both axes have a logarithmic scale.

app data-flow graph involving 51 935 apps in 12 minutes (using a family-based strategy on groups of apps). Interestingly, a pure family-based strategy does not scale to the size of this app set. The success of our strategy suggests that the approach can be applied to even larger app sets (e.g., the whole Google Play Store) by adding more app groups. The bottleneck for building a graph of the Play Store would be the composition of the graphs for app groups (last step in Figure 7.7). This step is not computationally expensive, however the whole graph is stored in main memory at some point.

7.6 Threats to Validity

External validity The external validity of our evaluation depends on the choice of (i) the app benchmark sets and on (ii) the tools which we compare SIFTA.

ICC-BENCH and DROIDBENCH are established third-party benchmarks used also in other studies. To evaluate accuracy, we also created IACBENCH to include test cases not covered by ICC-BENCH and DROIDBENCH, especially advanced communication scenarios, such as loops, intent chains, and recognition of multiple identical intents. IACBench is publicly available at our supplementary website. To evaluate scalability, we go beyond existing benchmarks in this area (IccRE with 523 and MalGenome with 1260 apps) by analyzing 51 935 real-world apps from Google Play.

Empirical studies on real-world apps are commonly prone to the app-sampling problem [MHJ+15]. In fact, obtaining a truly random sample of apps of an app store is almost impossible due to the store's size. However, this problem does not apply to our evaluation, since we do not aim at such a representative sample. Instead, we sought to obtain apps that are likely to communicate. Our strategy was to start with one of the most popular apps, facebook, to scan its Play store website, and follow links to apps listed under "similar" and "more from developer". This process was continued, allowing us to download apps across various app categories. Yet, the strategy targeted apps that are likely to communicate, leading to a dataset suitable to evaluate Sifta's scalability. The mining script and the list of apps are available on our supplementary website.

In our accuracy and scalability experiments, we compared Sifta to two state-of-the-art tools for intra-app (IccTA) and inter-app communication analysis (DidFail). Further tools exist for intra-app analysis (e.g., PermissionFlow [SBG+13], CHEX [LLW+12]), but we chose the most recent and mature tool IccTA, focusing specifically on analyzing data flows. Although IccTA is more precise than Sifta, this limitation is acceptable, given that our focus is on Sifta's scalability. Achieving more precision is possible, but requires significant effort for creating an industry-strength tool. For scalability, our comparison is limited to DidFail, as the only other tool supporting inter-app communication.

Internal validity In Sifta, we implemented the matching of intents to receiving components, which is essentially a re-implementation of the Android systems' intent-matching algorithm. For this purpose, we relied on Android's documentation. Yet, a threat to validity is that we did not implement all (possibly undocumented) corner cases of intent matching or that we mis-interpreted the documentation. However, our results show that Sifta agrees with IccTA on most ICC benchmark test cases, which indicates proper matching.

In our experiments, we found that FlowDroid reports many false-positive flows on real apps (experiments 3 and 4). Usually, these arise from private data being stored in class fields and intents being instantiated in the same class. We looked at several of these flows manually: The private data are visible to the code that generates the intent, but is not attached to the intent. FlowDroid reports a flow in these situations. After consulting with a FlowDroid developer, we implemented a filter that removes such flows from FlowDroid's output. For similar reasons, we filter intent results and intents with empty actions. However, this introduces a threat of removing too many flows. Still, we argue that missing a few true positives is better than reporting thousands of false-positive leaks, which would render the analysis useless. We only used this filtering in experiments 3 and 4, which aimed at scalability anyway.

7.7 Related Work

In this section, we discuss related work that analyzes Android apps and communication between apps. Our variability-aware data-flow representation and analysis has various applications in software engineering (build secure apps, prevent accidental flows) and security analysis (detect private-data leaks or high-risk apps) [vRBS+15].

Privacy leaks in mobile apps Privacy leaks inside and across mobile apps have been extensively studied [BR14]. Several researchers argue that the permission system used in Android is insufficient to prevent tainted data flows. For instance, permissions are too coarse-grained [NKZ10; OMEM12] and surprisingly rarely used in practice [OMJ+13] (only 5 % of the publicly accessible components are protected). Furthermore, apps often ask for more permissions than are actually used [PCH+11], giving rise to accidental leaks.

Enck et al. [EOMC11] studied 1100 popular Android apps, analyzing their use of libraries and misuse of private information. They found that apps often access personal information, such as the IMEI number, often combined with account information. Many apps also heavily use of advertisement libraries [MNA+15], forcing acquisition of many permissions.

Data-flow analysis of Android apps To the best of our knowledge, our approach is the first to effectively scale inter-app data-flow analysis to large app sets.

Apart from DidFail, many tools focusing on intra-app communication exist. Yet, most stop at component boundaries, such as PermissionFlow [SBG+13], which does not incorporate intents and their flows, or FlowDroid [ARF+14], which we use for our component analysis. Some tools can track data flows across components. The most notable intra-app, but inter-component analysis tools are Amandroid [WROR14] and IccTA [LBB+15]. We explained the difference between Sifta and IccTA already in Section 7.2. Amandroid is similar in accuracy to IccTA [LBB+15], and also resolves flows across components (using its own points-to analysis, where we use Epicc). We considered Amandroid for our accuracy experiments, but were not able to execute it. However, Amandroid targets only intra-app analysis (as confirmed by the developers).

All these tools differ in their accuracy and how they handle the peculiarities of Android, such as the main Android library and native calls. Our approach can use different underlying tools, and leverage them to create highly compressed data-flow graphs effective in identifying tainted data flows.

Finally, dynamic analysis tools such as TaintDroid [EGH+14] track data flows across applications at run time. While these conceptually provide the highest accuracy, they are limited by the dynamic analysis, not being able to confirm the absence of tainted flows. Most importantly, they can only analyze fixed sets of apps.

CHAPTER 8

Concluding Remarks and Future Work

We give a summary of the topics and results of this thesis in Section 8.1. Subsequently, we summarize our contributions per chapter in Section 8.2 and discuss their scientific and practical impact in Section 8.3. We conclude with an outline of ongoing and future research directions in Section 8.4.

8.1 Summary

Configurable systems often have a huge number of variants, which makes analysis of all variants individually very expensive. In this thesis, we developed and compared strategies for the efficient analysis of configurable systems. Our results show that the *family-based* strategy is often more efficient than alternative strategies and that it can be improved even further by combining it with, for example, a *variant-based* strategy.

The LINUX kernel has over 13 165 configuration options [PPB+15], giving rise to billions of different configurations. As configurable systems are used in critical applications, such as avionics or operating systems, certain properties, such as type correctness and other system specifications, need to be ensured for all variants. In traditional software-system engineering, this is ensured with program analysis and testing. For configurable systems, such methods can be directly applied when one analyzes and tests each variant separately (brute-force, *variant-based* approach). In the presence of large configuration spaces, however, this approach does not scale.

For large configurable systems, researchers developed several specialized analysis strategies: *Sampling* strategies analyze and test a selected subset of the system variants. This implies that some defects can be missed, but scalability is improved. *Feature-based strategies* perform separate analyses for code that belongs to different configuration options. This approach, by design, misses interactions between options, but it avoids the combinatorial complexity of building system configurations. *Family-based strategies* combine all variants of a configurable system into one analysis subject and analyze this subject in one run. The approach resolves system variability at a late point (in the analysis) and, therefore, can benefit from commonalities between variants of the system.

thesis goal

The goal of this thesis is to enable efficient analysis of highly configurable software systems. We addressed the question of how program analyses (e.g., data-flow analysis and model checking) can be applied to configurable software systems. To this end, we developed the PLA model for describing analysis strategies for configurable systems (Chapter 3). The PLA model covers the four basic analysis strategies (variant-based, sampling, feature-based, and family-based) and various combinations of them. Based on the model, we discussed advantages and disadvantages of different combinations of basic strategies. Furthermore, we took first steps to explore the model by implementing several analysis strategies and by comparing their performance with each other. Our evaluations have shown that, depending on the analyzed configurable system and the type of analysis (e.g., type checking or model checking), choosing a good analysis strategy is critical for analysis performance. The results of the family-based and variant-based strategies cover all variants whereas the results of a sample based strategy are incomplete because only part of the variants is analyzed. Furthermore, the family-based strategy is often faster than the sampling and variant-based strategies. However, the analysis can consume more main memory than the sampling and variant-based strategies. A family-based analysis backed by a variability-aware data structure can enable analysis in settings with very large subject systems (Chapter 7). By combining the family-based and, for example, the variant-based strategy, one can improve analysis performance even further (Section 6.6).

8.2 Contributions

PLA model In Chapter 3, we introduced the Product-Line Analysis (PLA) Model. In comparison to a previous model by Thüm et al. [TAK$^+$14], the PLA model includes more complex combinations of strategies. Furthermore, we introduced the *PLA cube*, an illustrative, visual representation of the PLA model. Each point in the cube represents an analysis strategy for configurable systems. The model (and the cube) form the basis for the evaluation of configurable-system analysis strategies that we began in this thesis. The model guided our analysis design and helped us to optimize strategies.

Presence-condition simplification Presence conditions are used in many configurable-system analyses to denote in which configurations certain situations (e.g., type errors) occur or certain system elements (e.g., function implementations) are involved. Presence conditions are often more complex than necessary, for example because presence-condition complexity has not been considered during analysis development. In Chapter 4, we defined the problem of presence-condition simplification formally. The key idea is to use context information (e.g., the variability model) to reduce the size of presence conditions. As a result, the reduced presence conditions are easier to understand for users and their usage improves analysis' performance. We identified three algorithms, Restrict [CM92], Quine-McCluskey [McC56; Qui52], and Espresso [BSMH84] that can be used for presence-condition simplification. In a series of experiments, we evaluated the different algorithms with respect to their effectiveness, processing time, and their potential in different application scenarios. In our experiments, we found considerable potential for simplification of presence conditions in many real use cases (e.g., performance prediction or defect reporting). Our experiments show that the algorithms are suited well for simplification and that size reduction factors are usually better with Restrict than with Quine-McCluskey or Espresso. Concerning scalability, we have shown that Restrict can handle larger problems than Quine-McCluskey or Espresso. Resulting from our work, presence-condition simplification has been integrated already in the analysis tool TypeChef.[1]

Variability encoding In Chapter 5, we presented variability encoding, an automatic code transformation of compile-time configurable code into load-time configurable code. The result of the transformation is a variant simulator, which includes the functionality of all system variants. We use variant simulators, for example, in family-based model checking (Chapter 6). We formally defined variability encoding and proved that generated simulators correctly simulate the behavior of all variants. Furthermore, we presented a tool, Hercules, that implements variability encoding for C programs with preprocessor directives. We evaluated the accuracy of Hercules based on the configurable data-base management system SQLite and its test suite TH3. Our results show that Hercules correctly encodes the variability of over 80% of the variants and tests we executed. This is an impressive result, given the complexity of the

[1] http://ckaestne.github.io/TypeChef/

setup and implementation of TH3 and SQLite, which caused many false errors in our evaluation. Simulators that were generated with variability encoding, have enabled various family-based analyses [ASW+11; AvRW+13; BLB+15; KvRE+12; Mei14; SvRA13] in the configurable-systems research.

Family-based model checking We implemented an efficient, family-based software-model-checking approach for configurable systems by extending the software model checkers CPAchecker (for C programs) and Java Pathfinder (for Java programs). Our model-checking approach is based on isolating presence conditions in states of the reachability graph that is generated by a model checker. We represent the presence conditions of the states in binary decision diagrams. This representation allows us to implement several optimizations that make the model-checking process more efficient for configurable systems. We implemented this optimization as extensions for CPAchecker and Java Pathfinder. We described the model-checking optimization and our implementation in Chapter 6. We demonstrated that the optimization improves performance in comparison to off-the-shelf versions of the model checkers.

Utilizing our optimized family-based model-checking approach, we evaluated and compared the performance of family-based, sample-based, and variant-based model checking. Our evaluation is partly based on subject systems that we implemented based on system specifications from other researchers [Hal05; KMSL83; PR01]. All subject systems are publicly available, and some have already been used by other researchers [Bey15; BLB+15; Mei14]. Our results show that family-based model checking reduces verification time by 73%, compared to variant-based model checking, while delivering the same level of information on defects of the configurable system. Model checking based on configuration sampling is faster, but may miss defects.

Our evaluation of family-based model checking shows that it is faster than variant-based model checking. However, it typically consumes more main memory because larger programs are verified. To show how this problem can be mitigated, we explored combinations of family-based and variant-based model checking strategies (Section 6.6), guided by our PLA model. In a nutshell, we partitioned the set of variants of a configurable system and verified each partition in a separate simulator. We evaluated how such analyses perform in comparison to the pure family-based and pure variant-based strategies. Our results show that combining basic strategies has potential to reduce the disadvantages one experiences when using only family-based or variant-based strategies. Furthermore, it shows the potential of the PLA model for guiding the development of efficient analysis strategies for configurable systems.

Inter-application data-flow analysis Private-data leaks on mobile devices are a growing concern, as more and more consumers use smart phones and apps get more powerful. In Chapter 7, we described our extension of a tool that builds a data-flow graph for communication between ANDROID apps. Our extension is based on the fact that each considered app can be installed or not (like a feature). In this view, a mobile ANDROID device is a configurable system. Our main contribution to the tool is a variability-aware data structure that enables us to scale the graph generation to 51 935 apps, which is much more than other tools can handle at this time. The generated graph can be used to detect potentially malicious data flows and give valuable information for a manual review of the apps that contribute to malicious data flows. In terms of the PLA model, the tool we extended implements a variant-based analysis. We modified the tool by integrating a family-based analysis based on a variability-aware data structure that improves scalability. Furthermore, we extended the analysis such that it, optionally, uses the family-based strategy on groups of apps (which corresponds to a combination of the family-based and feature-based strategies in the PLA model). This combined strategy enabled us to generate the graph for our set of 51 935 apps on a machine where the pure family-based strategy failed due to memory limits.

8.3 Impact

We discuss the impact of our work on the research community and on practice separately.

Impact on research We established a model representing combinations of basic analysis strategies of configurable systems (Chapter 3). In combination with other work on configurable-system analysis (e.g., [TAK+14]), the PLA model provides an overview of advantages and disadvantages of individual analysis strategies, how strategies are used in practice, and which combinations of strategies promise good analysis performance. The PLA model is meant to serve as a common basis for the development of advanced analysis strategies by the research community.

Presence-condition simplification (Chapter 4) and variability encoding (Chapter 5) have impact beyond our own research. Presence-condition simplification provides a simple method to improve the output or the internal data structure of configurable-system analyses. It is already adopted in the tool TYPECHEF, which is used for type checking, variability encoding (extension HERCULES), and refactoring (extension MORPHEUS [LJG+15]) of large configurable systems (e.g., SQLITE and LINUX).

Variability encoding provides a relatively simple means for transforming compile-time variability to run-time variability, which enables many family-based analyses [AvRW+13; BLB+15; KvRE+12; SvRA13]. Our formal definition and our proof of behavior preservation increases confidence in the correctness of variability encoded simulators and suggest their use in analyses.

Our implementations and evaluations of analyses of configurable systems (Chapters 6, 7, and especially Section 6.6) serve as examples for the exploration of the PLA cube. Our evaluations helped to broaden the community knowledge base on analyses of configurable systems. Based on the PLA model, we showed how different analysis strategies influence analysis performance and how the model can be explored in future work.

Impact on practice The impact of this thesis on practice is mainly represented by our tools (HERCULES, SPLVERIFIER, and SIFTA) and our evaluations. Even though our tools are research prototypes and do not have industry-level quality, the corresponding evaluations (Sections 5.1.1, 5.5.2, 6.5 and 7.5) show that the tools can, in principle, be applied to analyze real-world configurable software systems. These tools could also serve as blueprints for industry-level re-implementations.

Furthermore, we had contact with a corporate research group at BOSCH who expressed interest in using HERCULES for variability encoding of configurable systems with static variability. In particular, they planned to use variability encoding for analyses (similar to our experiments in Chapters 5 and 6) and to enable customers to choose configuration options at load-time without re-compilation of the system.

8.4 Future Work

Our work lays the foundation for different research directions in the area of analysis of configurable software systems. In the following paragraphs, we highlight some promising directions.

PLA-model exploration In this thesis, we developed, evaluated, and compared several analysis strategies and discussed how they are represented in the PLA model. This work could be extended by a systematic exploration of additional regions in the PLA cube (similar to our evaluation in Section 6.6). This would provide further insights on which (combinations of) strategies are efficient in which application scenarios and for which subject systems. For example, one could explore the effect of different sampling strategies on a family-based model-checking approach (range **C**–**D** in the PLA cube) with different software systems (different in system size, feature size, and number of features).

Another possible direction is to use models of software (e.g., labelled transition systems instead of JAVA/C source code) as subject systems, because at this level, system differences (e.g., size and variability) are easier to control. For example, one could use a model checker for labelled transition systems to extend our evaluation of the family-based strategy on configurable-system partitions (range **A**–**D** in the PLA cube, Section 6.6). This way, one could explore partition-based verification in a more controlled setting than it is possible with software model checking. The goal of such study could be to determine which properties of partitionings (e.g., similarity of partitions) improve performance of the partition-based verification approach.

Presence-condition simplification In our work on presence-condition simplification, we identified three algorithms that can be used in practice. However, these algorithms do not scale for extremely large inputs. For example, when a scenario requires using the LINUX variability model as context, the algorithms do not scale (in Chapter 4, the application scenarios in which we used LINUX did not involve its variability model). It is highly desirable to come up with algorithms for presence-condition simplification that cover problem sizes at the scale of the LINUX variability model. Furthermore, our evaluation should be extended to more subject systems and additional application scenarios for presence-condition simplification. We consider presence-condition simplification a very useful tool, which has likely more application scenarios in configurable-system research than we identified so far.

Variability encoding We have shown that variability encoding and HERCULES can be applied to code from real-world systems, such as LINUX and SQLITE. In future work, HERCULES should be applied to other configurable systems, which could then be verified to that they are safe in all variants. Even though we have reached a relatively stable development state, new subject systems will probably lead to discovery of new bugs in our implementation. Real-world C code can be very complex to analyze and therefore, it would be much, but rewarding, work to harden HERCULES to industry-level or implement its concepts in a industry-level tool. Furthermore, implementation of variability encoding for other programming languages, such as C++ or ADA might be interesting. For example, avionics systems are often implemented in ADA and need to be verified rigorously [WSA+15], which could be made easier with variability encoding and family-based verification.

Family-based model checking The extensions for variability-aware, family-based model checking that we described in Chapter 6 can, in principle, be combined with other model-checking optimizations. It would be desirable to use our extension in concert with novel model-checking optimizations to verify large-scale real-world systems. For example, a recent project [KT14] enabled model checking on a large number of applications in the debian package-management system. They automatically extracted source code from the packages and verified interesting properties, such as valid memory accesses, absence of arithmetic overflow, or absence of not-a-number in floating point operations. Combining this automatic approach and family-based model checking on configurable systems, such as BUSYBOX or SQLITE, would potentially discover many valuable, configuration-dependent defects. Such bugs would be difficult to obtain with traditional, sample-based verification because they occur only in some variants.

Inter-app communication We showed that our tool SIFTA is able to generate an inter-app communication graph for 51 935 ANDROID applications. Based on this graph, we implemented a light-weight analysis that reports potential private-data leaks. However, the graph has more potential: It could be used for more advanced netwok analysis methods, such as centrality measures and community detection. These methods can give important insights in the graph structure (e.g., groups of apps that collaborate) which could be used, for example, to organize app stores. Another example would be to compute the minimum cut of the graph. This cut identifies the minimum number of apps that must be removed to prevent any data flows from private sources to public sinks and thereby to prevent leaks of private user data.

Bibliography

[ABW14] I. Abal, C. Brabrand, and A. Wąsowski, “42 Variability Bugs in the Linux Kernel: A Qualitative Study”, in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, ACM, 2014, pp. 421–432.

[Ach11] M. Acher, “Managing Multiple Feature Models: Foundations, Language and Applications”, PhD thesis, Université Nice-Sophia Antipolis, 2011.

[ALSU06] A. V. Aho, M. S. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.

[AMS03] F. A. Aloul, I. L. Markov, and K. A. Sakallah, “FORCE: A Fast and Easy-to-Implement Variable-Ordering Heuristic”, in *Proceedings of the Great Lakes Symposium on VLSI*, ACM, 2003, pp. 116–119.

[And97] H. R. Andersen, “An Introduction to Binary Decision Diagrams”, Lecture Notes, available online at `http://www.cs.unb.ca/~gdueck/courses/cs4835/bdd97.pdf`, accessed March 3, 2014, 1997.

[AF96] T. W. Anderson and J. D. Finn, *The New Statistical Analysis of Data*. Springer-Verlag, 1996.

[Ape10] S. Apel, “How AspectJ is Used: An Analysis of Eleven AspectJ Programs”, *Journal of Object Technology*, vol. 9, no. 1, pp. 117–142, 2010.

[ABKS13] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines – Concepts and Implementation*. Springer-Verlag, 2013.

[ABF+13] S. Apel, D. Beyer, K. Friedberger, F. Raimondi, and A. von Rhein, "Domain Types: Abstract-Domain Selection Based on Variable Usage", in *Proceedings of the Haifa Verification Conference (HVC)*, Springer-Verlag, 2013, pp. 262–278.

[AH10] S. Apel and D. Hutchins, "A Calculus for Uniform Feature Composition", *ACM Transactions on Programming Languages and Systems*, vol. 32, no. 5, 19:1–19:33, 2010.

[AKGL10] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer, "Type Safety for Feature-Oriented Product Lines", *Automated Software Engineering*, vol. 17, no. 3, pp. 251–300, 2010.

[AKL09] S. Apel, C. Kästner, and C. Lengauer, "FeatureHouse: Language-Independent, Automated Software Composition", in *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE, 2009, pp. 221–231.

[AKL13] S. Apel, C. Kästner, and C. Lengauer, "Language-Independent and Automated Software Composition: The FeatureHouse Experience", *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 63–79, 2013.

[AL08] S. Apel and C. Lengauer, "Superimposition: A Language-Independent Approach to Software Composition", in *Proceedings of the International Symposium on Software Composition (SC)*, vol. 4954, Springer-Verlag, 2008, pp. 20–35.

[ALMK10] S. Apel, C. Lengauer, B. Möller, and C. Kästner, "An Algebraic Foundation for Automatic Feature-Based Program Synthesis", *Science of Computer Programming*, vol. 75, no. 11, pp. 1022–1047, 2010.

[ASW+11] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer, "Detection of Feature Interactions using Feature-Aware Verification", in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, IEEE, 2011, pp. 372–375.

[AvRW+13] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer, "Strategies for Product-Line Verification: Case Studies and Experiments", in *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 482–491.

[App15a] AppBrain, *Free vs. paid* ANDROID *apps*, `http://www.appbrain.com/stats/free-and-paid-android-applications`, (accessed September 17, 2015), 2015.

[App15b] AppBrain, *Number of available applications in the Google Play Store from December 2009 to July 2015*, `http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/`, (accessed September 24, 2015), 2015.

[App14] Apple, *App Store Sales Top $10 Billion in 2013*, `http://www.apple.com/pr/library/2014/01/07App-Store-Sales-Top-10-Billion-in-2013.html`, (accessed October 16, 2015), 2014.

[ARB13] S. Arzt, S. Rasthofer, and E. Bodden, "Susi: A Tool for the Fully Automated Classification and Categorization of Android Sources and Sinks", University of Darmstadt, Tech. Rep. TUD-CS-2013-0114, 2013.

[ARF+14] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps", in *Proceedings of the International Conference on Programming Languages Design and Implementation (PLDI)*, ACM, 2014, pp. 259–269.

[AtBFG10] P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi, "A Logical Framework to Deal with Variability", in *Proceedings of the International Conference on Integrated Formal Methods (IFM)*, Springer-Verlag, 2010, pp. 43–58.

[AtBFG11] P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi, "A Model-Checking Tool for Families of Services", in *Proceedings of the Joint International Conference on Formal Techniques for Distributed Systems (FORTE/FMOODS)*, Springer-Verlag, 2011, pp. 44–58.

[BK08] C. Baier and J. Katoen, *Principles of Model Checking.* MIT Press, 2008.

[BCR94] V. R. Basili, G. Caldiera, and H. D. Rombach, "The Goal Question Metric Approach", in *Encyclopedia of Software Engineering*, Wiley, 1994.

[Bat04] D. Batory, "Feature-Oriented Programming and the AHEAD Tool Suite", in *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE, 2004, pp. 702–703.

[Ber13] T. Berger, “Variability Modeling in the Real–An Empirical Journey from Software Product Lines to Software Ecosystems”, PhD thesis, Universität Leipzig, 2013.

[BPT+14] T. Berger, R.-H. Pfeiffer, R. Tartler, S. Dienst, K. Czarnecki, A. Wasowski, and S. She, “Variability Mechanisms in Software Ecosystems”, *Information and Software Technology*, vol. 56, no. 11, pp. 1520–1535, 2014.

[BSCW10] T. Berger, S. She, K. Czarnecki, and A. Wąsowski, “Feature-to-Code Mapping in Two Large Product Lines”, Department of Computer Science, University of Leipzig, Tech. Rep., 2010.

[BSL+10] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wąsowski, “Feature-to-Code Mapping in Two Large Product Lines”, in *Proceedings of the International Software Product Line Conference (SPLC)*, ACM, 2010, pp. 498–499.

[BSL+13] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki, “A Study of Variability Models and Languages in the Systems Software Domain”, *IEEE Transactions on Software Engineering*, vol. 39, no. 12, pp. 1611–1640, 2013.

[BDS13] L. Bettini, F. Damiani, and I. Schaefer, “Compositional Type Checking of Delta-oriented Software Product Lines”, *Acta Informatica*, vol. 50, no. 2, pp. 77–122, 2013.

[BW09] D. Beuche and J. Weiland, “Managing Flexibility: Modeling Binding-Times in Simulink”, in *Proceedings of the European Conference on Model Driven Architecture–Foundations and Applications (ECMDA-FA)*, Springer-Verlag, 2009, pp. 289–300.

[Bey15] D. Beyer, “Software Verification and Verifiable Witnesses (Report on SV-COMP 2015)”, in *Proceedings of the International Conference on Tools and Algorithms for the Construction and of Analysis Systems (TACAS)*, Springer-Verlag, 2015, pp. 401–416.

[BHTV13] D. Beyer, A. Holzer, M. Tautschnig, and H. Veith, “Information Reuse for Multi-goal Reachability Analyses”, in *Proceedings of the European Symposium on Programming (ESOP)*, Springer-Verlag, 2013, pp. 472–491.

[BK11] D. Beyer and M. Keremoglu, “CPAchecker: A Tool for Configurable Software Verification”, in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, Springer-Verlag, 2011, pp. 184–190.

[BS13] D. Beyer and A. Stahlbauer, "BDD-Based Software Model Checking with CPAchecker", in *Proceedings of the Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS)*, Springer-Verlag, 2013, pp. 1–11.

[BR14] N. A. Bidani and M. V. Raffay, "A Systematic Literature Review of Mobile Inter-Application Security", Master's thesis, IT University of Copenhagen, 2014.

[BHvMW09] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, vol. 185.

[BTR$^+$13] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini, "SPL$^{\text{LIFT}}$ – Statically Analyzing Software Product Lines in Minutes Instead of Years", in *Proceedings of the International Conference on Programming Languages Design and Implementation (PLDI)*, ACM, 2013, pp. 355–364.

[BRB91] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package", in *Proceedings of the Design Automation Conference (DAC)*, IEEE/ACM, 1991, pp. 40–45.

[BSMH84] R. K. Brayton, A. L. Sangiovanni-Vincentelli, C. T. McMullen, and G. D. Hachtel, *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer, 1984.

[Bry92] R. E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams", *ACM Computing Surveys*, vol. 24, no. 3, pp. 293–318, 1992.

[BU11] D. Buchfuhrer and C. Umans, "The Complexity of Boolean Formula Minimization", *Journal of Computer and System Sciences*, vol. 77, no. 1, pp. 142–153, 2011.

[BLB$^+$15] J. Bürdek, M. Lochau, S. Bauregger, A. Holzer, A. von Rhein, S. Apel, and D. Beyer, "Facilitating Reuse in Multi-goal Test-Suite Generation for Software Product Lines", in *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, Springer-Verlag, 2015, pp. 84–99.

[BFK$^+$15] J. Burket, L. Flynn, W. Klieber, J. Lim, W. Shen, and W. Snavely, "Making DidFail Succeed: Enhancing the CERT Static Taint Analyzer for Android App Sets", Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-2015-TR-001, 2015.

[CCR10] I. Cabral, M. B. Cohen, and G. Rothermel, “Improving the Testing and Testability of Software Product Lines”, in *Proceedings of the International Software Product Line Conference (SPLC)*, Springer-Verlag, 2010, pp. 241–255.

[CM06] M. Calder and A. Miller, “Feature Interaction Detection by Pairwise Analysis of LTL Properties: A Case Study”, *Formal Methods in System Design*, vol. 28, no. 3, pp. 213–261, 2006.

[CEW12] S. Chen, M. Erwig, and E. Walkingshaw, “An Error-Tolerant Type System for Variational Lambda Calculus”, in *Proceedings of the International Conference on Functional Programming (ICFP)*, ACM, 2012, pp. 29–40.

[CEW14] S. Chen, M. Erwig, and E. Walkingshaw, “Extending Type Inference to Variational Programs”, *ACM Transactions on Programming Languages and Systems*, vol. 36, no. 1, 1:1–1:54, 2014.

[CPGW11] E. Chin, A. Porter Felt, K. Greenwood, and D. Wagner, “Analyzing Inter-application Communication in Android”, in *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, ACM, 2011, pp. 239–252.

[CGJ$^+$03] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-Guided Abstraction Refinement for Symbolic Model Checking”, *Journal of the ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[CGP99] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking.* MIT Press, 1999.

[CKNZ12] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, “Model Checking and the State Explosion Problem”, in *Tools for Practical Software Verification*, Springer-Verlag, 2012, pp. 1–30.

[CKL04] E. M. Clarke, D. Kroening, and F. Lerda, “A Tool for Checking ANSI-C Programs”, in *Proceedings of the International Conference on Tools and Algorithms for the Construction and of Analysis Systems (TACAS)*, Springer-Verlag, 2004, pp. 168–176.

[Cla11] A. Classen, “Modelling and Model Checking Variability-Intensive Systems”, PhD thesis, University of Namur, Belgium, 2011.

[CCH$^+$11] A. Classen, M. Cordy, P. Heymans, P.-Y. Schobbens, and A. Legay, “SNIP: An Efficient Model Checker for Software Product Lines”, PReCISE Research Center, University of Namur, Tech. Rep. P-CS-TR SPLMC-00000003, 2011.

[CCS$^+$13] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin, “Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and their Application to LTL Model Checking”, *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1069–1089, 2013.

[CHSL11] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay, “Symbolic Model Checking of Software Product Lines”, in *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM, 2011, pp. 321–330.

[CHS$^+$10] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, “Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines”, in *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM, 2010, pp. 335–344.

[CN01] P. Clements and L. M. Northrop, *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2001.

[CDS07] M. B. Cohen, M. B. Dwyer, and J. Shi, “Interaction Testing of Highly-configurable Systems in the Presence of Constraints”, in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, ACM, 2007, pp. 129–139.

[CKM12] J. Corbet, G. Kroah-Hartman, and A. McPherson, *Linux Kernel Development*, http://go.linuxfoundation.org/who-writes-linux-2012, 2012.

[CCP$^+$12] M. Cordy, A. Classen, G. Perrouin, P.-Y. Schobbens, P. Heymans, and A. Legay, “Simulation-Based Abstractions for Software Product-Line Model Checking”, in *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM, 2012, pp. 672–682.

[CHL$^+$14] M. Cordy, P. Heymans, A. Legay, P.-Y. Schobbens, B. Dawagne, and M. Leucker, “Counterexample guided abstraction refinement of product-line behavioural models”, in *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, ACM, 2014, pp. 190–201.

[CSRL01] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms.* McGraw-Hill Higher Education, 2001.

[CBM90] O. Coudert, C. Berthet, and J. C. Madre, “Verification of Synchronous Sequential Machines Based on Symbolic Execution”, in *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, Springer-Verlag, 1990, pp. 365–373.

[CM92] O. Coudert and J. C. Madre, “Implicit and Incremental Computation of Primes and Essential Primes of Boolean Functions”, in *Proceedings of the Design Automation Conference*, IEEE, 1992, pp. 36–39.

[CS02] O. Coudert and T. Sasao, “Two-level Logic Minimization”, in *Logic Synthesis and Verification*, Kluwer, 2002, pp. 1–27.

[CE00] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[CW07] K. Czarnecki and A. Wąsowski, “Feature Diagrams and Logics: There and Back Again”, in *Proceedings of the International Software Product Line Conference (SPLC)*, IEEE, 2007, pp. 23–34.

[DAg99] M. D’Agostino, “Tableau Methods for Classical Propositional Logic”, in *Handbook of Tableau Methods*, Springer-Verlag, 1999, pp. 45–123.

[DLvL15] C. Damas, B. Lambeau, and A. van Lamsweerde, “Generating Process Models in Multi-View Environments”, in *Dependable Software Systems Engineering*, ser. NATO Science for Peace and Security Series - D: Information and Communication Security, IOS Press, 2015, pp. 105–127.

[DM02] A. Darwiche and P. Marquis, “A Knowledge Compilation Map”, *Artificial Intelligence Research*, vol. 17, pp. 229–264, 2002.

[Dav58] M. Davis, *Computability and Unsolvability.* Dover Publications, 1958.

[dNV90] R. de Nicola and F. Vaandrager, “Action Versus State Based Logics for Transition Systems”, in *Proceedings of the LITP Spring School on Theoretical Computer Science on Semantics of Systems of Concurrent Processes*, Springer-Verlag, 1990, pp. 407–419.

[DCB09] B. Delaware, W. R. Cook, and D. Batory, “Fitting the Pieces Together: A Machine-Checked Model of Safe Composition”, in *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, ACM, 2009, pp. 243–252.

[DB12] S. Dienst and T. Berger, *Static Analysis of App Dependencies in Android Bytecode*, Tech. note, available at `http://informatik.uni-leipzig.de/~berger/tr/2012-dienst.pdf`, 2012.

[DTSL12] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann, "A Robust Approach for Variability Extraction from the Linux Build System", in *Proceedings of the International Software Product Line Conference (SPLC)*, ACM, 2012, pp. 21–30.

[Dom12] A. L. Dominguez, "Detection of Feature Interactions in Automotive Active Safety Features", PhD thesis, University of Waterloo, 2012.

[DBT11] F. Dordowsky, R. Bridges, and H. Tschope, "Implementing a Software Product Line for a Complex Avionics System", in *Proceedings of the International Software Product Line Conference (SPLC)*, IEEE, 2011, pp. 241–250.

[DH09] F. Dordowsky and W. Hipp, "Adopting Software Product Line Principles to Manage Software Variants in a Complex Avionics System", in *Proceedings of the International Software Product Line Conference (SPLC)*, Carnegie Mellon University, 2009, pp. 265–274.

[DKB14] C. Dubslaff, S. Klüppelholz, and C. Baier, "Probabilistic Model Checking for Energy Analysis in Software Product Lines", in *Proceedings of the International Conference on Modularity (MODULARITY)*, ACM, 2014, pp. 169–180.

[EGH$^+$14] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones", *Transactions on Computer Systems*, vol. 32, no. 2, 5:1–5:29, 2014.

[EOMC11] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A Study of Android Application Security", in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, USENIX Association, 2011, pp. 21–21.

[EW11] M. Erwig and E. Walkingshaw, "The Choice Calculus: A Representation for Software Variation", *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 1, pp. 1–27, 2011.

[EP05] S. Evangelista and J.-F. Pradat-Peyre, “Memory Efficient State Space Storage in Explicit Software Model Checking”, in *Proceedings of the International SPIN Workshop*, Springer-Verlag, 2005, pp. 43–57.

[FG07] A. Fantechi and S. Gnesi, “A Behavioural Model for Product Families”, in *Proceedings of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE): Companion Papers*, ACM, 2007, pp. 521–524.

[FKA$^+$13] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake, “Do background colors improve program comprehension in the #ifdef hell?”, *Empirical Software Engineering*, vol. 18, no. 4, pp. 699–745, 2013.

[FMJJ92] J.-C. Fernandez, L. Mounier, C. Jard, and T. Jéron, “On-the-fly Verification of Finite Transition Systems”, *Formal Methods in System Design*, vol. 1, no. 2-3, pp. 251–273, 1992.

[FUB06] D. Fischbein, S. Uchitel, and V. Braberman, “A Foundation for Behavioural Conformance in Software Product Line Architectures”, in *Proceedings of the Workshop on the Role of Software Architecture for Testing and Analysis (ROSATEA)*, ACM, 2006, pp. 39–48.

[GP12] S. Gnesi and M. Petrocchi, “Towards an Executable Algebra for Product Lines”, in *Proceedings of the International Software Product Line Conference (SPLC)*, ACM, 2012, pp. 66–73.

[Gru10] A. Gruler, “A Formal Approach to Software Product Lines”, PhD thesis, Technical University of Munich, 2010.

[GLS08] A. Gruler, M. Leucker, and K. Scheidemann, “Modeling and Model Checking Software Product Lines”, in *Proceedings of the International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, Springer-Verlag, 2008, pp. 113–131.

[GPFW97] J. Gu, P. W. Purdom, J. Franco, and B. W. Wah, “Algorithms for the Satisfiability (SAT) Problem: A Survey”, in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, 1997, pp. 19–152.

[HCF05] V. Haldar, D. Chandra, and M. Franz, "Dynamic Taint Propagation for Java", in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, IEEE, 2005, pp. 303–311.

[Hal05] R. J. Hall, "Fundamental nonmodularity in electronic mail", *Automated Software Engineering*, vol. 12, no. 1, pp. 41–79, 2005.

[Hal77] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier, 1977.

[Har88] N. Hardy, "The Confused Deputy (or Why Capabilities Might Have Been Invented)", *ACM Operating Systems Review*, vol. 22, no. 4, pp. 36–38, 1988.

[HS11] E. Hemaspaandra and H. Schnoor, "Minimization for Generalized Boolean Formulas", in *International Joint Conference on Artificial Intelligence (IJCAI)*, AAAI, 2011, pp. 566–571.

[HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy Abstraction", in *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*, vol. 37, ACM, 2002, pp. 58–70.

[Hey12] P. Heymans, "Formal Methods for the Masses", in *Proceedings of the International Software Product Line Conference (SPLC)*, ACM, 2012, p. 4.

[Hol97] G. J. Holzmann, "State Compression in SPIN: Recursive Indexing And Compression Training Runs", in *Proceedings of the International SPIN Workshop*, 1997.

[HD05] J. Huang and A. Darwiche, "On Compiling System Models for Faster and More Scalable Diagnosis", in *Proceedings of the Conference on Artificial Intelligence (AAAI)*, MIT Press, 2005, pp. 300–306.

[JWEG07] P. Jayaraman, J. Whittle, A. M. Elkhodary, and H. Gomaa, "Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis", in *Proceedings of the International Conference on Model-Driven Engineering, Languages, and Systems (MODELS)*, Springer-Verlag, 2007, pp. 151–165.

[JM09] R. Jhala and R. Majumdar, "Software Model Checking", *ACM Computing Surveys*, vol. 41, no. 4, 21:1–21:54, 2009.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[Kap12] T. Kapus, "Specifying System Families with TLA+", in *Proceedings of the International Conference on Software Engineering, Parallel and Distributed Systems, and Proceedings of the International Conference on Engineering Education (WSEAS)*, World Scientific, Engineering Academy, and Society, 2012, pp. 98–103.

[KAK09] C. Kästner, S. Apel, and M. Kuhlemann, "A Model of Refactoring Physically and Virtually Separated Features", in *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, ACM, 2009, pp. 157–166.

[KAK08] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in Software Product Lines", in *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM, 2008, pp. 311–320.

[KATS12] C. Kästner, S. Apel, T. Thüm, and G. Saake, "Type Checking Annotation-Based Product Lines", *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 3, 14:1–14:39, 2012.

[KGR+11] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation", in *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, ACM, 2011, pp. 805–824.

[KOE12] C. Kästner, K. Ostermann, and S. Erdweg, "A Variability-Aware Module System", in *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, ACM, 2012, pp. 773–792.

[KvRE+12] C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann, "Toward Variability-Aware Testing", in *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, ACM, 2012, pp. 1–8.

[KLM$^+$97] G. Kiczales, J. Lamping, A. Mendhekar, C. V. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming", in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Springer-Verlag, 1997, pp. 220–242.

[KKB12] C. Kim, S. Khurshid, and D. Batory, "Shared Execution for Efficiently Testing Product Lines", in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2012, pp. 221–230.

[KFB$^+$14] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android Taint Flow Analysis for App Sets", in *Proceedings of the International Workshop on the State of the Art in Java Program Analysis (SOAP)*, ACM, 2014, pp. 1–6.

[KMMR00] M. Kolberg, E. Magill, D. Marples, and S. Reiff-Marganiec, "Results of the Second Feature Interaction Contest", in *Proceedings of the Workshop on Feature Interactions in Telecommunications and Software Systems (ICFI)*, IOS Press, 2000, pp. 311–325.

[KvRHA13] S. Kolesnikov, A. von Rhein, C. Hunsen, and S. Apel, "A Comparison of Product-based, Feature-based, and Family-based Type Checking", in *Proceedings of the International Conference on Generative Programming: Concepts & Experience (GPCE)*, ACM, 2013, pp. 115–124.

[KV07] B. Korte and J. Vygen, *Combinatorial Optimization: Theory and Algorithms*. Springer-Verlag, 2007.

[KMSL83] J. Kramer, J. Magee, M. Sloman, and A. Lister, "CONIC: An Integrated Approach to Distributed Computer Control Systems", *Computers and Digital Techniques*, vol. 130, no. 1, pp. 1–10, 1983.

[KT14] D. Kroening and M. Tautschnig, "Automating Software Analysis at Large Scale", in *Proceedings of the Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS)*, Springer-Verlag, 2014, pp. 30–39.

[KBA09] M. Kuhlemann, D. Batory, and S. Apel, "Refactoring Feature Modules", in *Proceedings of the International Conference on Software Reuse (ICSR)*, Springer-Verlag, 2009, pp. 106–115.

[LTP09] K. Lauenroth, S. Toehning, and K. Pohl, "Model Checking of Domain Artifacts in Product Line Engineering", in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, IEEE, 2009, pp. 269–280.

[LT98] Y. Lei and K.-C. Tai, “In-Parameter-Order: A Test Generation Strategy for Pairwise Testing”, in *Prodceedings of the International Symposium on High-Assurance Systems Engineering (HASE)*, IEEE, 1998, pp. 254–261.

[LKF02a] H. Li, S. Krishnamurthi, and K. Fisler, “Interfaces for modular feature verification”, in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, IEEE, 2002, pp. 195–204.

[LKF05] H. Li, S. Krishnamurthi, and K. Fisler, “Modular Verification of Open Features Using Three-Valued Model Checking”, *Automated Software Engineering*, vol. 12, no. 3, pp. 349–382, 2005.

[LKF02b] H. Li, S. Krishnamurthi, and K. Fisler, “Verifying Cross-cutting Features as Open Systems”, in *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM, 2002, pp. 89–98.

[LBB$^+$15] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, “IccTA: Detecting Inter-Component Privacy Leaks in Android Apps”, in *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE, 2015, pp. 280–292.

[Lib05] P. Liberatore, “Redundancy in Logic I: CNF Propositional Formulae”, *Artificial Intelligence Research*, vol. 163, no. 2, pp. 203–232, 2005.

[Lie15] J. Liebig, “Analysis and Transformation of Configurable Systems”, PhD thesis, University of Passau, Germany, 2015.

[LAL$^+$10] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, “An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines”, in *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM, 2010, pp. 105–114.

[LALL09] J. Liebig, S. Apel, C. Lengauer, and T. Leich, “RobbyDBMS: A Case Study on Hardware/Software Product Line Engineering”, in *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, ACM, 2009, pp. 63–68.

[LJG$^+$15] J. Liebig, A. Janker, F. Garbe, S. Apel, and C. Lengauer, “Morpheus: Variability-Aware Refactoring in the Wild”, in *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE, 2015, pp. 380–391.

[LKA11] J. Liebig, C. Kästner, and S. Apel, “Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code”, in *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, ACM, 2011, pp. 191–202.

[LvRK$^+$13] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer, “Scalable Analysis of Variable Software”, in *Proceedings of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ACM, 2013, pp. 81–91.

[LBL11] J. Liu, S. Basu, and R. Lutz, “Compositional Model Checking of Software Product Lines using Variation Point Obligations”, *Automated Software Engineering*, vol. 18, no. 1, pp. 39–76, 2011.

[Loc12] M. Lochau, “Model-Based Conformance Testing of Software Product Lines”, PhD thesis, TU Braunschweig, 2012.

[LOGS12] M. Lochau, S. Oster, U. Goltz, and A. Schürr, “Model-based pairwise testing for feature interaction coverage in software product line engineering”, *Software Quality Journal*, vol. 20, no. 3–4, pp. 567–604, 2012.

[LB01] R. Lopez-Herrejon and D. Batory, “A standard problem for evaluating product-line methodologies”, in *Proceedings of the International Conference on Generative and Component-Based Software Engineering (GCSE)*, Springer-Verlag, 2001, pp. 10–24.

[LLW$^+$12] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, “CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities”, in *Proceedings of the Conference on Computer and Communications Security (CCS)*, ACM, 2012, pp. 229–240.

[MHJ$^+$15] W. Martin, M. Harman, Y. Jia, F. Sarro, and Y. Zhang, “The App Sampling Problem for App Store Mining”, in *Proceedings of the Conference on Mining Software Repositories (MSR)*, IEEE, 2015, pp. 123–133.

[McC56] E. J. McCluskey, “Minimization of Boolean functions”, *Bell System Technical Journal*, vol. 35, no. 5, pp. 1417–1444, 1956.

[Mei14] J. Meinicke, “VarexJ: A Variability-Aware Interpreter for Java Applications”, Master’s thesis, University of Magdeburg, 2014.

[Men09] M. Mendonça, “Efficient Reasoning Techniques for Large Scale Feature Models”, PhD thesis, University of Waterloo, 2009.

[MBC09] M. Mendonça, M. Branco, and D. Cowan, "S.P.L.O.T.: Software Product Lines Online Tools", in *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, ACM, 2009, pp. 761–762.

[MWC09] M. Mendonça, A. Wąsowski, and K. Czarnecki, "SAT-based Analysis of Feature Models is Easy", in *Proceedings of the International Software Product Line Conference (SPLC)*, SEI, 2009, pp. 231–240.

[MWCC08] M. Mendonça, A. Wąsowski, K. Czarnecki, and D. Cowan, "Efficient Compilation Techniques for Large Scale Feature Models", in *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, ACM, 2008, pp. 13–22.

[MBW14] J. Midtgaard, C. Brabrand, and A. Wąsowski, "Systematic Derivation of Static Analyses for Software Product Lines", in *Proceedings of the International Conference on Modularity (MODULARITY)*, ACM, 2014, pp. 181–192.

[MRKN13] J.-V. Millo, S. Ramesh, S. N. Krishna, and G. K. Narwane, "Compositional Verification of Software Product Lines", in *Proceedings of the International Conference on Integrated Formal Methods (IFM)*, Springer-Verlag, 2013, pp. 109–123.

[Mil99] R. Milner, *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.

[MAN+14] I. J. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, and A. E. Hassan, "A Large Scale Empirical Study on Software Reuse in Mobile Apps", *IEEE Software*, vol. 31, no. 2, pp. 78–86, 2014.

[MNA+15] I. J. Mojica, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. E. Hassan, "On Ad Library Updates in Android Apps", *IEEE Software*, 2015, preprint, accepted for publication.

[MS04] G. J. Myers and C. Sandler, *The Art of Software Testing*. John Wiley & Sons, 2004.

[NH13] S. Nadi and R. Holt, "The Linux Kernel: A Case Study of Build System Variability", *Journal of Software: Evolution and Process*, 2013.

[NKZ10] M. Nauman, S. Khan, and X. Zhang, “Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints”, in *Proceedings of the Symposium on Information, Computer, and Communications Security (ASIACCS)*, ACM, 2010, pp. 328–332.

[NR69] P. Naur and B. Randell, Eds., *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee*, Retrieved November 12, 2014, from `http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF`, Scientific Affairs Division, NATO, 1969.

[NL11] C. Nie and H. Leung, “A Survey of Combinatorial Testing”, *ACM Computing Surveys*, vol. 43, no. 2, 11:1–11:29, 2011.

[OMJ+13] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, “Effective Inter-Component Communication Mapping in Android with Epicc: An essential Step Towards Holistic Security Analysis”, in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, USENIX Association, 2013, pp. 543–558.

[Oec03] P. Oechslin, “Making a Faster Cryptanalytic Time-Memory Trade-Off”, in *Proceedings of the International Cryptology Conference (CRYPTO)*, Springer-Verlag, 2003, pp. 617–630.

[OSC+14] G. Ofenbeck, R. Steinmann, V. C. Cabezas, D. Spampinato, and M. Püschel, “Applying the Roofline Model”, in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2014, pp. 76–85.

[OMEM12] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, “Semantically Rich Application-centric Security in Android”, *Security and Communication Networks (SCN)*, vol. 5, no. 6, pp. 658–673, 2012.

[OMR10] S. Oster, F. Markert, and P. Ritter, “Automated Incremental Pairwise Testing of Software Product Lines”, in *Proceedings of the International Software Product Line Conference (SPLC)*, Springer-Verlag, 2010, pp. 196–210.

[PNX+11] L. Passos, M. Novakovic, Y. Xiong, T. Berger, K. Czarnecki, and A. Wąsowski, “A Study of Non-Boolean Constraints in a Variability Model of an Embedded Operating System”, in *Proceedings of the International Software Product Line Conference (SPLC)*, ACM, 2011, pp. 1–8.

[PPB$^+$15] L. Passos, J. Padilla, T. Berger, S. Apel, K. Czarnecki, and M. T. Valente, “Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers”, in *Proceedings of the International Conference on Modularity (MODULARITY)*, ACM, 2015, pp. 81–92.

[POS$^+$12] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. L. Traon, “Pairwise Testing for Software Product Lines: Comparison of Two Approaches”, *Software Quality Journal*, vol. 20, no. 3–4, pp. 605–643, 2012.

[PSK$^+$10] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon, “Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines”, in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2010, pp. 459–468.

[Pie02] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002.

[PR01] M. Plath and M. Ryan, “Feature Integration using a Feature Construct”, *Science of Computer Programming*, vol. 41, no. 1, pp. 53–84, 2001.

[PR98] M. Plath and M. Ryan, “Plug-and-Play Features”, in *Proceedings of the Workshop on Feature Interactions in Telecommunications and Software Systems (ICFI)*, IOS Press, 1998, pp. 150–164.

[PCH$^+$11] A. Porter Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, “Android Permissions Demystified”, in *Proceedings of the Conference on Computer and Communications Security (CCS)*, ACM, 2011, pp. 627–638.

[PWM$^+$11] A. Porter Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, “Permission Re-delegation: Attacks and Defenses”, in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, USENIX Association, 2011, pp. 22–22.

[PS08] H. Post and C. Sinz, “Configuration Lifting: Verification meets Software Configuration”, in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, IEEE, 2008, pp. 347–350.

[Qui52] W. V. Quine, *The Problem of Simplifying Truth Functions*. Mathematical Association of America, 1952.

[RP06] P. Rechenberg and G. Pomberger, *Informatik-Handbuch*. Carl Hanser Verlag, 2006.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv, "Precise Interprocedural Dataflow Analysis via Graph Reachability", in *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*, ACM, 1995, pp. 49–61.

[DO-178B] *RTCA DO-178B Software Considerations in Airborne Systems and Equipment Certification.* RTCA/EUROCAE Std. ED-12B/DO-178B, 1992.

[SBG$^+$13] D. Sbîrlea, M. G. Burke, S. Guarnieri, M. Pistoia, and V. Sarkar, "Automatic detection of inter-application permission leaks in Android applications", *IBM Research and Development*, vol. 57, no. 6, 10:1–10:12, 2013.

[SH11] I. Schaefer and R. Hähnle, "Formal Methods in Software Product Line Engineering", *IEEE Computer*, vol. 44, no. 2, pp. 82–85, 2011.

[SLB$^+$11] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki, "Reverse Engineering Feature Models", in *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE, 2011, pp. 461–470.

[SKK$^+$12] N. Siegmund, S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake, "Predicting Performance via Automated Feature-Interaction Detection", in *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE, 2012, pp. 167–177.

[SRK$^+$13] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. Kolesnikov, "Scalable Prediction of Non-functional Properties in Software Product Lines: Footprint and Memory Consumption", *Information and Software Technology*, vol. 55, no. 3, pp. 491–507, 2013.

[SvRA13] N. Siegmund, A. von Rhein, and S. Apel, "Family-Based Performance Measurement", in *Proceedings of the International Conference on Generative Programming: Concepts & Experience (GPCE)*, ACM, 2013, pp. 95–104.

[SSSS07] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk, "Is The Linux Kernel a Software Product Line?", in *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL)*, 2007.

[Som10] I. Sommerville, *Software Engineering.* USA: Addison-Wesley, 2010.

[Sub05] S. Subbarayan, "Integrating CSP Decomposition Techniques and BDDs for Compiling Configuration Problems", in *Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, Springer-Verlag, 2005, pp. 351–365.

[Tar13] R. Tartler, "Mastering Variability Challenges in Linux and Related Highly-Configurable System Software", PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2013.

[TDS+14] R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann, "Static Analysis of Variability in System Software: The 90,000 #ifdefs Issue", in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, USENIX Association, 2014, pp. 421–432.

[TLD+11] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero, "Configuration Coverage in the Analysis of Large-scale System Software", *ACM Operating Systems Review*, vol. 45, no. 3, pp. 10–14, 2011.

[tBdV14] M. H. ter Beek and E. P. de Vink, "Using mCRL2 for the Analysis of Software Product Lines", in *Proceedings of the Workshop on Formal Methods in Software Engineering (FormaliSE)*, ACM, 2014, pp. 31–37.

[tBFG14] M. H. ter Beek, A. Fantechi, and S. Gnesi, "Challenges in Modelling and Analyzing Quantitative Aspects of Bike-Sharing Systems", in *Proceedings of the International Symposium on Leveraging Applications of Formal Methods, Verification, and Validation (ISOLA)*, Springer-Verlag, 2014, pp. 351–367.

[tBFGM15] M. H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti, "Using FMC for Family-Based Analysis of Software Product Lines", in *Proceedings of the International Software Product Line Conference (SPLC)*, ACM, 2015, pp. 432–439.

[tBGM15] M. H. ter Beek, S. Gnesi, and F. Mazzanti, "From EU Projects to a Family of Model Checkers - From Kandinsky to KandISTI", in *Software, Services, and Systems - Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering*, Springer-Verlag, 2015, pp. 312–328.

[tBM14] M. H. ter Beek and F. Mazzanti, “VMC: Recent Advances and Challenges Ahead”, in *Proceedings of the International Software Product Line Conference (SPLC)*, ACM, 2014, pp. 70–77.

[TAK+14] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, “A Classification and Survey of Analysis Strategies for Software Product Lines”, *ACM Computing Surveys*, vol. 47, no. 1, 6:1–6:45, 2014.

[TBK09] T. Thüm, D. Batory, and C. Kästner, “Reasoning About Edits to Feature Models”, in *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE, 2009, pp. 254–264.

[TMB+14] T. Thüm, J. Meinicke, F. Benduhn, M. Hentschel, A. von Rhein, and G. Saake, “Potential Synergies of Theorem Proving and Model Checking for Software Product Lines”, in *Proceedings of the International Software Product Line Conference (SPLC)*, ACM, 2014, pp. 177–186.

[TSAH12] T. Thüm, I. Schaefer, S. Apel, and M. Hentschel, “Family-Based Deductive Verification of Software Product Lines”, in *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, ACM, 2012, pp. 11–20.

[TDP03] O. Tkachuk, M. B. Dwyer, and C. S. Păsăreanu, “Automated Environment Generation for Software Model Checking”, in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, Elsevier, 2003, pp. 116–129.

[TCO00] P. Toft, D. Coleman, and J. Ohta, “A Cooperative Model for Cross-divisional Product Development for a Software Product Line”, in *Proceedings of the International Software Product Line Conference (SPLC)*, Kluwer, 2000, pp. 111–132.

[vLSR07] F. J. van der Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action.* Springer-Verlag, 2007.

[vLS79] A. van Lamsweerde and M. Sintzoff, “Formal Derivation of Strongly Correct Concurrent Programs”, *Acta Informatica*, vol. 12, no. 1, pp. 1–31, 1979.

[VGN14] N. Viennot, E. Garcia, and J. Nieh, “A Measurement Study of Google Play”, in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, ACM, 2014, pp. 221–233.

[VHB+03] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model Checking Programs", *Proceedings of the International Conference on Automated Software Engineering (ASE)*, vol. 10, no. 2, pp. 203–232, 2003.

[vRAK+13] A. von Rhein, S. Apel, C. Kästner, T. Thüm, and I. Schaefer, "The PLA Model: On the Combination of Product-Line Analyses", in *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, ACM, 2013, pp. 73–80.

[vRAR11] A. von Rhein, S. Apel, and F. Raimondi, "Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code", `http://www.infosun.fim.uni-passau.de/cl/publications/docs/JPF2011.pdf`, presented at the Java Pathfinder Workshop, 2011.

[vRBS+15] A. von Rhein, T. Berger, N. Schalck Johansson, M. Mark Hardø, and S. Apel, "Lifting Inter-App Data-Flow Analysis to Large App Sets", Department of Computer Science and Mathematics, University of Passau, Tech. Rep. MIP-1504, Sep. 2015.

[vRGA+15] A. von Rhein, A. Grebhahn, S. Apel, N. Siegmund, D. Beyer, and T. Berger, "Presence-Condition Simplification in Highly Configurable Systems", in *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE, 2015.

[vRTS+16] A. von Rhein, T. Thüm, I. Schaefer, J. Liebig, and S. Apel, "Variability Encoding: From Compile-Time to Load-Time Variability", *Journal of Logical and Algebraic Methods in Programming*, vol. 85, no. 1, pp. 125–145, 2016.

[WCO04] L. Wall, T. Christiansen, and J. Orwant, *Programming Perl: 3rd Edition*. O'Reilly, 2004.

[WROR14] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps", in *Proceedings of the Conference on Computer and Communications Security (CCS)*, ACM, 2014, pp. 1329–1341.

[WSA+15] A. Wölfl, N. Siegmund, S. Apel, H. Kosch, J. Krautlager, and G. Weber-Urbina, "Generating Qualifiable Avionics Software: An Experience Report", in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, preprint, accepted for publication, IEEE, 2015,

[ZJ12] Y. Zhou and X. Jiang, “Dissecting Android Malware: Characterization and Evolution”, in *Proceedings of the Symposium on Security and Privacy (SSP)*, IEEE, 2012, pp. 95–109.

CHAPTER 9

Appendix

Pseudocode of the Restrict Algorithm

In Chapter 4, we used the Restrict algorithm [CM92] for presence condition simplification (simp_{BDD}). In the original publication of the algorithm, it is described only in prose and, therefore, we decided to provide the pseudocode here. Algorithm 9.1 shows our implementation of the Restrict algorithm, enhanced with a cache to achieve polynomial run-time complexity (also suggested in the original publication [CM92]). The implementation is equivalent to a pseudocode implementation in a set of lecture notes on binary decision diagrams [And97]. In the evaluation (Section 4.4) we actually used the native Restrict implementation from the JavaBDD library for efficiency reasons. The pseudocode presentation of Restrict uses some functions that are commonly used in publications about BDDs. We introduce these functions in the following paragraph.

An expression f stored in a BDD has the structure $(x \wedge f_x) \vee (\neg x \wedge f_{\neg x})$ where x is a single Boolean variable. f_x and $f_{\neg x}$ are expressions that are implied by x and $\neg x$, respectively. f_x and $f_{\neg x}$ are again stored in BDD form unless they are tautologies (*true*) or contraditions (*false*). We denote an expression stored in a BDD with $\text{ite}(x, f_x, f_{\neg x})$. The BDD for the expression in Figure 2.8b on page 30 is $\text{ite}(c, f_c, f_{\neg c}) = (c \wedge f_c) \vee (\neg c \wedge f_{\neg c})$ where f_c and $f_{\neg c}$ are again BDDs. We denote the "top" variable of a given BDD f with $\text{var}(f)$ (c in the example). When comparing variables, we follow the variable ordering ($c < s < b$ in the example). The sub-trees of expression f are denoted with

Algorithm 9.1: RESTRICT (simp$_{BDD}$)

```
Data: BDD $p$, BDD $m$, Table $Cache$
Result: BDD $c$
if $Cache(p, m) \neq NIL$ then return $Cache(p, m)$
else
    if $p = false$ then $r \leftarrow false$
    else if $m = false \vee m = true$ then $r \leftarrow m$
    else if $p = true$ then
        $r \leftarrow \text{ite}((\text{var}(m), \text{low}(u), \text{Restrict}(p, \text{high}(m))))$
    else if $\text{var}(p) = \text{var}(m)$ then
        if $\text{low}(p) = false$ then $r \leftarrow \text{Restrict}(\text{high}(p), \text{high}(m))$
        else if $\text{high}(p) = false$ then
            $r \leftarrow \text{Restrict}(\text{low}(p), \text{low}(m))$
        else
            $r \leftarrow \text{ite}(\text{var}(p), \text{Restrict}(\text{low}(p), \text{low}(m)),$
            $\text{Restrict}(\text{high}(p), \text{high}(m)))$
        end
    else if $\text{var}(p) < \text{var}(m)$ then
        $r \leftarrow \text{ite}(\text{var}(p), \text{Restrict}(\text{low}(p), m), \text{RESTRICT}(\text{high}(p), m))$
    else
        $r \leftarrow \text{ite}(\text{var}(m), \text{Restrict}(p, \text{low}(m)), \text{RESTRICT}(p, \text{high}(m)))$
    end
    $Cache(p, m) \leftarrow r$
    return $r$
end
```

high(f) and low(f). An expression f that is stored in a BDD can be denoted with $\text{ite}(\text{var}(f), \text{high}(f), \text{low}(f))$.

The RESTRICT algorithm (simp$_{BDD}$ in Chapter 4) takes two expressions: p (the presence condition) and m (the context) represented as BDDs and generates a third BDD $x = \text{RESTRICT}(p, m)$, that satisfies the invariant of Equation 4.1 on page 64 [CBM90].

RESTRICT traverses BDDs p and m in parallel and builds the result BDD p' with $p' = \text{RESTRICT}(p, m)$. Based on local comparisons of p and m, RESTRICT is either called recursively on sub-problems or the recursion is terminated. To achieve polynomial run-time complexity, we added a cache that stores already computed results for sub-problems. For further details, we refer to the original publication [CBM90].

Like many other BDD operations, RESTRICT is a polynomial-time graph manipulation algorithm (when caching is used). As said before, in the worst case, the size of the graph may be exponential in the number of the variables, which also renders the algorithm exponential in the number of variables. In our experience, BDDs were usually much smaller, but we have no hard evidence on this.

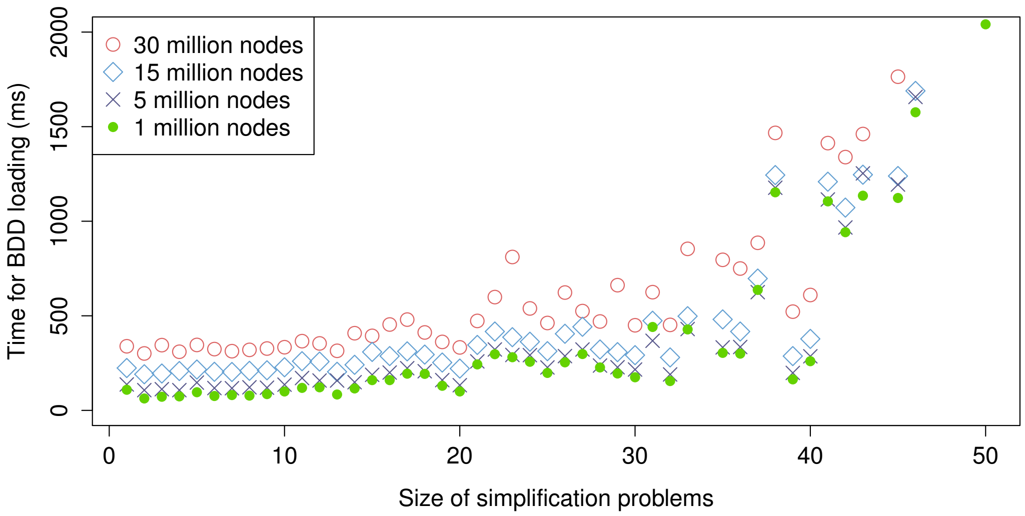


Figure 9.1: Times for simplification-task loading with different numbers of initial nodes in the BDD library (based on Experiment 5 4.4). The plot is truncated at 2000ms to show more details. The distances between the truncated points are similar to the shown points (but some are at 10 000ms).

Reducing the Startup Time of the simp_{BDD} Algorithm

One of the results of our evaluation of presence-condition simplification is that the simp_{BDD} algorithm has a significantly higher startup time compared to simp_E and simp_{QC}. We did further experiments to investigate the cause for this. In particular, we investigated how the *initial number of nodes* in the BDD library influences the startup time. A BDD library organizes many BDDs in a shared graph-based data structure. Upon initialization of the BDD library the user has to specify an initial number of nodes and a number of nodes that is used for re-organization (as a cache). In earlier experiments with BDDs (see Section 6.5) we experienced that re-organization can consume a significant amount of time. Therefore we set the initial number of nodes rather high to avoid re-organization and its influence on the measured run time.

In the experiment of Chapter 4, we set the initial number of nodes to 30 million.[1] To investigate the startup time, we repeated Experiment 5 (Section 4.4)

[1] We also tried larger values but got out-of-memory errors due to the experiment's limitation to 4GB RAM.

with 1 million, 5 million, 15 million, and 30 million initial nodes. Simplification of a presence condition with simp_{BDD} is done in three phases: (1) the presence condition and the context are parsed from a file, (2) the presence condition is simplified, and (3) the simplified condition is written to a file. The BDD library is initialized in the first phase. Figure 9.1 shows the time consumed by the first phase with different numbers of initial nodes. In each call of simp_{BDD} the times for the simplification and result writing phases were below 2ms and 10ms, respectively.

The results from our experiment show that the initial number of nodes in the BDD library influences the costs for startup of the simp_{BDD} algorithm. Fewer nodes mean that the BDD library has to reserve and initialize less memory and the presence condition and context are loaded faster. The results also show that the time for the simplification can be neglected. This makes sense as in BDDs, the main work is done during loading of the presence condition and the context in BDDs (many AND and OR operations). Once this is done, the simplification operation has a computational cost en par with AND and OR.

This experiment shows that simp_{BDD} could be optimized for a simplification task by initializing the BDD library with an initial number of nodes that is *just enough* to accomondate the condition and context. However, it is difficult to estimate the number of necessary nodes precisely.

Verifying Bugs of the Variability Bug Database

To illustrate the practical use of our tool Hercules, we inspected real-world bugs from the Variability Bug Database. The database contains known bugs from different real-world systems. Each listed bug is related to configuration options and only causes bug behavior in certain configurations. We looked at the 43 bugs that are listed for the Linux kernel. For each bug, the database contains a description and simplified code that can be used to reproduce the bug.

We used the bugs to illustrate an application of Hercules, namely program verification with model checking. We used Hercules to encode the variability in the bug's source code in a variant simulator. Then, we used the model checking tool CPAchecker to verify the simulator. If both tools perform as expected, the model checker finds the bug and reports it's presence condition (cf. Chapter 6). The reported presence condition should be the same as the condition given in the bug description. We focus on CPAchecker as a model checking tool because it is able to report the bug's presence condition using our extension described in Section 6.2. Competing model checkers such as CBMC [CKL04] cannot do this out-of-the-box.

For each bug, we evaluated whether we can apply model checking in the first place, and whether CPAchecker is able to find the bug. We evaluated the bugs based on 3 criteria:

- The bugs must require at least two configuration options to be enabled or disabled. In code with only one configuration option, variability encoding is often trivial.
- The bugs must be executable, otherwise CPAchecker cannot verify the code. This excludes bugs that are based on type- and syntax errors.
- The bugs must not require precise modelling of the heap or of system functions. We used two model checking configurations implemented in CPAchecker, *valueAnalysis* and *predicateAnalysis*. Both configurations do not support precise modelling of the heap or of system functions.

Figure 9.1 shows the results of our evaluation. We filtered most bugs based on the three criteria. For some bugs we had to limit variability which render the bug uninteresting. For example, in bug `657e964`, we had to enable one (of two) options because of syntax errors. Several other bugs were rejected because CPAchecker was unable to detect the bug even in the variant (without variability). The bugs rely on pointer which are passed between several functions (often the pointers refer to uninitialized memory). The CPAchecker configurations that we used do not implement pointer tracking in most situations.

In the end, we found three bugs in which the tool chain of Hercules and CPAchecker is applicable and reports the bug with the expected presence condition. This evaluation shows that Hercules can be used on real-world programs and that the tool chain can be used to detect real-world variability-dependent bugs.

Table 9.1: Overview of the LINUX bugs in the Variability Bug Database. Each bug description can be accessed at `http://vbdb.itu.dk/#bug/linux/<BugID>` where `<BugID>` is substituted with the ID of the bug.

Bug ID	Type	Result	Reason
60e233a	buffer overflow	Not tried	First degree interaction
f3d83e2	buffer overflow	Not tried	Every line of code has the error presence condition; variability encoding trivial
8c82962	buffer overflow	Not tried	Usage of malloc essential for bug; would be ignored by CPACHECKER
809e660	dead code	Not tried	Dead code; not interesting for model checking
d7e9711	double lock	Not tried	First degree interaction
ae249b5	assertion violation	Fail	Depends on function memset and memory handling. CPACHECKER ignores this.
0988c4c	assertion violation	Not tried	First degree interaction
657e964	assertion violation	HERCULES and CPACHECKER ok, not interesting	Had to fix one (of two) option before HERCULES run due to compile error
d549f55	assertion violation	**HERCULES and CPACHECKER ok**	
63878ac	assertion violation	**HERCULES and CPACHECKER ok**	
e1fbd92	incompatible type	Not tried	Type errors, not interesting for model checking
d6c7e11	incompatible type	Not tried	Type errors, not interesting for model checking
c708c57	index out of bounds	Not tried	Requires pointer arithmetic and memory modelling (array bounds)
218ad12	memory leak	Not tried	Array-Out-Of-Bounds; not checked by CPACHECKER
e68bb91	multiple definitions	Not tried	Function defined multiple times; Type error, not interesting for model checking
0dc77b6	not enough memory	Not tried	Memory allocation; errors cannot be found without memory modelling
1f758a4	not enough memory	Not tried	Memory allocation; errors cannot be found without memory modelling
6252547	null dereference	Not tried	Pointer tracking necessary
76baeeb	null dereference	Not tried	Pointer tracking necessary
ee3f34e	null dereference	Not tried	Pointer tracking necessary
f7ab9b4	null dereference	Fail	CPACHECKER reports too many infeasible paths
51fd36f	numeric truncation	Not tried	Problem is conversion between 64-bit int and 32-bit int. CPACHECKER ignores this.
0f8f809	out of bounds read	Not tried	Pointer tracking necessary
91ea820	out of bounds read	Not tried	Depends on function memset and memory handling. CPACHECKER ignores this.
472a474	sysfs api violation	HERCULES and CPACHECKER ok, but not interesting	Had to fix one (of two) option (x86) before HERCULES run due to compile error
f48ec1d	undeclared identifier	Not tried	Type errors, not interesting for model checking
6651791	undeclared identifier	Not tried	Type errors, not interesting for model checking
6515e48	undefined symbol	Not tried	Type errors, not interesting for model checking
7c6048b	undefined symbol	Not tried	Type errors, not interesting for model checking
2f02c15	undefined symbol	Not tried	Type errors, not interesting for model checking
242f1a3	undefined symbol	Not tried	Type errors, not interesting for model checking
30e0532	uninitialized variable	Fail	Pointer tracking necessary; effectively only one configuration option in code
bc8cec0	uninitialized variable	Not tried	First degree interaction
e39363a	uninitialized variable	**HERCULES and CPACHECKER ok**	Inserted code for variable initialization and (later) check if value was overwritten
7acf6cd	uninitialized variable	Not tried	Pointer tracking necessary
1c17e4d	uninitialized variable	Fail	Pointer tracking necessary
36855dc	unused variable	Not tried	Unused variable; not interesting for model checking
6e2b757	use after free	Not tried	Memory allocation and memory copying; not handled in CPACHECKER
d530db0	void pointer dereference	Not tried	Dereference of void pointer; bug is already detected by the compiler
208d898	assertion violation	Not tried	First degree interaction
eb91f1d	assertion violation	Not tried	Error caused by memory allocation; not handled in CPACHECKER
221ac32	write on readonly	Fail	Read protected memory; not handled in CPACHECKER
e67bc51	wrong number of function arg.	Not tried	Type errors, not interesting for model checking; First degree interaction