Variability Bugs in System Software

Der Technischen Fakultät der
Friedrich-Alexander-Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

Julio Cezar Rodrigues Sincero

Erlangen — 2013

Als Dissertation genehmigt von
der Technischen Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Tag der Einreichung:	10.12.2012
Tag der Promotion:	17.04.2013
Dekanin:	Prof. Dr.-Ing. Marion Merklein
Berichterstatter:	Prof. Dr.-Ing. Wolfgang Schröder-Preikschat
	Prof. Dr. Antônio Augusto Fröhlich

Abstract

One of the key aspects of software product line engineering (SPLE) is the handling of variation points that can be combined to form specific products. Variability management is the discipline responsible for the specification, combination and use of variation points. Many researches have contributed to this area by providing formal methods to analyze variability by means of reasoning operations built on top of SAT solvers.

The development of system software also faces many challenges regarding variability management. To cope with variability issues, practitioners have employed a diversity of independent tools to tailor system software in order to avoid the overhead of unneeded functionality. In the Linux kernel, which can also be regarded as a software product line, variability management is handled in a heterogeneous manner. A set of independent tools is employed on different levels of abstraction, which might lead to inconsistencies in the description of variability. These inconsistencies are called *variability bugs*. With the novel concepts of *source code variability models*, *model slicing*, and *multi-model reasoning operations*, variability bugs that are possibly hidden by heterogenous variability management approaches can be attacked. The Linux code base serves as an ideal testbed for these novel techniques that conjoin the reasoning operations from SPLE with the challenges imposed by the structure and the sheer size of real-world large-scale code bases.

For an efficient, scalable and automatic detection of variability bugs, the tool `undertaker` implements the concepts of source code variability models, model slicing, multi-model reasoning operations and a SAT solver backend. Applying the `undertaker` tool to the Linux kernel code base revealed an impressive number of variability bugs, to which fixes were provided to the kernel developers. Their positive feedback confirms the need of tool support for the automatic detection of variability bugs.

Variabilitätsfehler in Systemsoftware

Abstrakt: Einer der Schwerpunkte bei der Entwicklung von Softwareproduktlinien stellt die Behandlung der Variationspunkte dar, aus denen durch Kombination Produktvarianten gebildet werden können. Das Variabilitätsmanagement ist verantwortlich für die Spezifikation, Kombination und Verwendung von Variationspunkten in der Produktlinienentwicklung. Viele Wissenschaftler haben Beiträge zu diesem Gebiet in Form von formalen Methoden zur automatisierten Analyse von Variabilität auf der Basis von Erfüllbarkeitslösern (SAT solver) geleistet. Die Entwicklung von Systemsoftware steht ebenfalls vor großen Herausforderungen in Bezug auf Variabilitätsmanagement. Um die Variabilität in den Griff zu bekommen, wird von Praktikern eine Vielzahl von unabhängigen Werkzeugen verwendet, um Systemsoftware maßgeschneidert unter Vermeidung unnützer Funktionalität zu erstellen. Im Linux-Betriebssystemkern, der ebenfalls als Softwareproduktlinie aufgefasst werden kann, wird Variabilitätsmanagement in heterogener Weise durchgeführt: Eine Reihe voneinander unabhängiger Werkzeuge wird auf unterschiedlichen Abstraktionsstufen eingesetzt, was leicht zu Inkonsistenzen bezüglich der Beschreibung von Variabilität führen kann. Diese Inkonsistenzen werden *Variabilitätsfehler* (variability bugs) genannt. Mit den neuen Konzepten *Source Code Variability Models*, *Model Slicing* und *Multi-Model Reasoning Operations* können Variabilitätsfehler behandelt werden, die vom heterogenen Variabilitätsmanagement nicht entdeckt werden. Die Linux Codebasis dient als eine ideale Testumgebung für diese neuen Techniken, mittels der die formalen Methoden aus der Softwareproduktlinien-Entwicklung auf ihre Anwendbarkeit in umfangreichen, realen Softwareprojekten untersucht werden. Für eine effiziente, skalierbare und automatische Erkennung von Variabilitätsfehlern, implementiert das Werkzeug *Undertaker* die Konzepte Source Code Variability Models, Model Slicing und Multi-Model Reasoning, sowie einen SAT-solver. Durch Anwendung des Werkzeugs auf den Quellcode des Linux Betriebssystemkerns wurde eine eindrucksvolle Anzahl von Variabilitätsfehlern entdeckt. Die positive Reaktion der Betriebssystemkern-Entwickler auf die eingesandten Fehlerbeschreibungen bestätigt die Zweckmäßigkeit der werkzeuggestützten, automatischen Erkennung von Variabilitätsfehlern.

Acknowledgements

First and foremost I am very grateful to Prof. Dr. Wolfgang Schröder-Preikschat who welcomed me to his group and provided a very inspirational environment for the research in the area of system software and software engineering. I am also deeply indebted to Prof. Dr. Antônio Augusto Fröhlich, who was the first to believe in my potential as a researcher. I am also very thankful to Prof. Dr. Olaf Spinczyk, as he was the first to invite me to join the PhD program after I completed my master thesis. I owe many thanks to Dr. Jürgen Kleinöder who took care of all aspects regarding financing, contracts and visas.

Most of the research that provided the results of my thesis was performed in collaboration with the colleagues of the VAMOS group. It was a pleasure to work with you guys: Reinhard Tartler, Daniel Lohmann, Christian Dietrich and Christoph Egger. From the very beginning up to the very end, I was supported by my friend Dr. Jörg Barner, who not only proof read my manuscripts, but also introduced me to the details of the german culture and language. Many colleagues of my *generation* at I4 helped and inspired me in many different ways, also a big thank you to: Wanja Hofer, Christoph Elsner, Peter Ulbrich, Fabian Scheler, Dirk Wischermann, Wasif Gilani, Michael Gernoth, Mike Stilkerich, Tobias Distler and Moritz Strübe.

Many parts of this thesis were written after I left the I4 department. I was always encouraged and supported by the colleagues at the Balvi GmbH: Dr. Klaus-Dieter Hasse, Martin Schwabe, Jan-Henrik Baumgarten, Daniel Drünkler, Claudia Weinkauf, Susanne and Thomas Bak.

During this journey I was very fortunate to meet and marry my beloved wife Melina, whose support and kind words were there for me whenever I needed. My family also supported me not only during my PhD but in every important moment of my life: My father Geraldo, my mother Izabel, by brothers Geraldo and Eduardo, and also, my in-laws: Luis, Teresa, Eduardo and Natasha.

Contents

List of Figures

List of Tables

1
Introduction

1.1 Overview

software customization

Software customization has been an ever-present goal in the development of software engineering techniques. In simple terms, it allows us to customize a general software implementation for a specific purpose. The way we use computer systems today makes software customization essential. Nowadays we are surrounded by a diversity of computer systems like desktop systems, servers, tablets, mobile phones, etc. Many of these systems run software programs for the same purposes, for example, email client, calendars, video player, text editors, and many others. Software engineering aims at enabling the development of pieces of software that can be customized to such different scenarios, uses, platforms, levels of functionality, etc., in order to avoid re-development, improve reuse, and, consequently, reduce development and maintenance costs.

software product lines

Software Product Line Engineering (SPLE) is a modern discipline of software engineering that—among many other goals—aims at improving software customizability. The outcome of SPLE is a Software Product Line (SPL), which represents not a single software system, but a set of related products (also known as variants). From a common source base different products can be generated after the configuration process has taken place. During configuration the *user*, who employs the SPLE infrastructure to derive a variant, explicitly selects his features of interest and, in the ideal case, an appropriate product variant is automatically generated. The products of an SPL are distinguished by the set of features they contain, where each feature corresponds to an increment in functionality.

variability management

Crucial for the development of SPLs—and customizable software in general—is the identification of the functionality that is *common* to all variants, and the functionality that is *variable*. The activity in SPLE that is responsible for elicitating and documenting this information is known as *variability management*. *Variation points* determine parts in any artifact, which should be included in the final product only under specific circumstances; such points can appear in any type of artifact which is created during SPLE: Source code, documentation, configuration data for the build system, etc. Large-scale software may contain up to several thousands of features, which, in turn, may contain numerous dependencies among themselves, resulting in an abundance of variability-specific data. To make things even more complicated, these variation points are sometimes managed by different stakeholders, and, therefore, may evolve independently. Clearly, the identification, documentation, maintenance, and evolution of variation points is an intrinsically complex process.

system software

The methods and techniques fostered by the SPLE community have successfully been applied to a broad variety of domains. Interestingly, SPLE has its roots in the development of *software families*, which were originally designed to cope with the

variability challenges imposed by the domain of *system software*. In many cases, system software generates no business value on its own, but it rather links hardware and software (operating systems), or different software layers (databases, middleware, etc.). As a result, system software is influenced from two directions, since it has to meet all the requirements imposed by the software layers above and it and by the hardware layer below it, but not more. System software development is very challenging regarding variability management not only because of the amount of must-have variation points, but also because developers have be aware of any overhead that may incur from the realization of such variation points.

diversity of variability sources

In any complex software project the variation points will eventually spread over several artifacts including source code, configuration data, documentation, etc. This is no different for system software, however, in this domain special attention is given to the variability realization techniques that are employed in the source code. Serving as a basis for systems running on top of it, system software must be efficient and cannot accept overhead incurred from the variability realization techniques. As a result, a large number of system software projects rely on the mechanisms of the C Preprocessor (CPP). For decades the CPP has been the *de facto* standard for implementing variability in the software industry. However, due to its high level of flexibility, which allows virtually every kind of language abuse, it has been also sharply criticized [SC92]. Interestingly, we note a recent trend in academia [KAK08, Käs10, STLSP10, HEB+10] towards giving a second chance to the CPP as an efficient means for the implementation of variability in the source code.

In order to take advantage of its strengths (simplicity, overhead-free mechanisms, etc.), and, at the same time, avoid its disadvantages, modern projects impose guidelines on the usage of CPP for variability management. Basically these guidelines limit language abuse. Nevertheless, the integration of the CPP-based artifacts into the variability management process is still remarkably weak.

heterogeneous vs homogeneous variability management

Although the source code constitutes in most cases the largest source of variability in SPLs, a sound approach for variability management has to consider the other sources of variability as well. Many authors have studied [KCH+90, CHE04, BRCT05, Bat05, CHH09] in detail configuration models—especially feature models—in the context of SPLs. Such models can be used in different phases of SPLE, either during the development phase to support domain engineering, or during application engineering in order to support product derivation. The study of variability management for artifacts of the build systems is beginning to gain some attention too [BSL+10a]. SPLs which integrate different sources of variability, for example, the CPP for source code, feature models for configurations, and makefiles for build systems, employ what we call a *heterogeneous* variability management. On the other hand, there exit SPL projects that centralize variability management in specific

frameworks and tools [SDNB07, pvs], or employ *features* as the main abstraction unit [AK09], which makes features explicit at the programming level [Pre97]. Such SPLs employ a *homogeneous* variability management.

Two approaches for logical reasoning based on variability models have received considerable attention from the scientific community: First, reasoning on single variability models [BSRC10], for example, to detect dead features in feature models. Second, variability-aware analysis in SPLs with a homogeneous variability management [MHP+07, TBKC07], for example, to detect contradictions between two variability models of the same type. However, the challenges faced by SPLs that employ a heterogeneous variability management still remains to be addressed. In this work we raise the hypothesis that without appropriate tool support SPLs that employ a heterogeneous variability management will inevitably experience variability problems. We believe that a sound variability management approach for such scenarios has to be able to check *all sources of variability* against each other. To achieve this one has to (1) convert the variability information to a common format, (2) understand the hierarchical relations among the different models, (3) build and solve the reasoning problems that check for variability inconsistencies and (4) deal with model size explosion. These are some of the issues that make variability management in heterogeneous scenarios such a challenging undertaking.

the goal of this thesis

This thesis aims to tackle these problems by providing **efficient variability management techniques for SPLs of system software**. We focus on the **automatic detection of variability problems**. That is, problems that result from the incorrect implementation or use of variation points. Such problems might arise both from single models as well as from the interaction between any combination of different variability models. This work also aims at revealing inconsistencies among the *intended* variability—as described in explicit configuration models like feature models—and *actual* variability as described in the source code, build system, etc. Although there exist initial studies regarding variability problems, the reasoning framework that we present in this thesis, together with the results of our evaluation, enabled us to identify, analyze, understand and solve variability problems of real-world, large-scale, system-software SPLs. In the context of this work we coin the term *variability bug* and provide methods and tools for its automatic detection.

systematic and efficient crosschecking

Although there have been advances regarding reasoning operations in specific variability models, techniques that analyze the full variability of large-scale SPLs are still missing. Additionally, a systematic way of crosschecking several artifacts is crucial for the success of SPLE techniques in real-world projects. This thesis addresses these challenges by providing techniques, algorithms, heuristics and tools to efficiently examine the full variability of system-software product lines. We conjecture that in scenarios where the variability is handled by different tools in different for-

mats, the variability will eventually lose sync among the different artifacts, leading to variability bugs.

We aim to improve this situation by the design and development of methods to automate the detection of variability bugs in SPLs with several variability sources. We also aim to provide a pragmatic approach that scales efficiently to the size of real-world projects. Specifically, we provide a formalization of the `CPP` language in terms of propositional logic for a better integration of the `CPP` in the variability management process. Also, we introduce the concept of *variability model slicing* for a better scalability of reasoning techniques in large projects. Based on these two methods we build a logic framework that is capable of combining the variability information from many sources and performing *reasoning operations* to reveal variability bugs. Finally, a tool that implements all these concepts is presented and its practicability is confirmed by applying it to the Linux kernel code base.

1.2 Contributions

1. We survey both the current SPLE and system software development techniques to provide a critical analysis of their similarities and differences. Subsequently, we identify the potential synergies of these areas with regard to variability management. These initial considerations lay ground to study the Linux Kernel from the SPL perspective. As a result, we show evidence of the demand for techniques and tools to detect and avoid variability bugs in systems that employ heterogeneous variability management.

2. We introduce the concept of *source code variability models*, which enables a better integration of `CPP`-based code artifacts into the variability management process; such models are automatically extracted from the source files and converted to a common format so that consistency checks with other sources of variability can be carried out.

3. We introduce the concept of *slicing variability models*, which enables the efficient handling of very large models in reasoning operations. Our algorithm performs the extraction of a sub-model from a complete model containing only the necessary amount of information required for specific variability checks.

4. We present our reasoning framework that enables the examination of the full variability of SPLs. We show how to combine the models from different sources and how to perform reasoning operations that reveal inconsistencies. The framework can be employed in two ways. To assess the quality of existing code bases, as well as to support a consistent development of new features.

5. We perform a thorough evaluation of our algorithms and tools using the Linux Kernel as case study. As a result, we disclosed a large number of variability bugs which have been neglected for years in of one of the most important and relevant open source projects. Moreover, based on the findings of our tool we were able to provide fixes for many bugs, which to a large extent have already been incorporated into the Linux kernel mainline tree.

1.3 Outline

Chapter 2 introduces the basic concepts and terms that are necessary for the understanding of this thesis. Additionally, the chapter provides an analysis regarding the similarities, differences, and potential synergy points between SPLs and system software using the Linux kernel as the system software example project.

Chapter 3 proposes the new concept of *source code variability models*. These models are derived from `CPP`-based source files and lead to an improved integration of such artifacts into the variability management process.

Chapter 4 presents a novel concept for integrating very large models into the variability management process. We provide an algorithm for the slicing of variability models so that only the relevant parts of the model are used for specific reasoning purposes.

Chapter 5 explains how to integrate variability information from different software artifacts into a unified reasoning process. We show how this integration allows for the detection of variability bugs.

Chapter 6 contains the evaluation of the integration of all techniques presented in the previous chapters. A thorough assessment is presented using the Linux Kernel as case study.

Chapter 7 summarizes and concludes this thesis. We highlight that the techniques developed in the context of this work enabled us to build a tool that not only scales to large code bases but also achieves good performance and delivers accurate results.

2

Variability Management in *Theory* and in *Practice*

This chapter provides an introduction to the topics required for the understanding of this thesis. We basically address two main areas: *Software product lines* and the *Linux kernel*. Furthermore, we present a comparison of the variability management techniques employed in the Linux kernel and in SPLE in order to highlight their differences, similarities and potential synergies as described by theorists and employed by practitioners. Using the results of this analysis we revisit in detail the problems that will be tackled in this work, and thereafter, we set the goals of this thesis. *This chapter shares material with the OSSPL'07 paper 'Is The Linux Kernel a Software Product Line?' [SSSPS07] and the ASPL'08 paper 'The Linux Kernel Configurator as a Feature Modeling Tool' [SSP08].*

2.1 Software Product Lines

Software Product Lines (SPLs) represent a shift in paradigm where the goal is to develop a *set* of related products simultaneously in a given domain. This set of products is represented by an SPL that provides the infrastructure for the derivation of single products according to a specification. NORTHROP and CLEMENTS [NC01] define SPLs as follows:

> 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.

Among the various definitions available in the literature this is the closest to our view on SPLs. However, this definition focuses on both *business* ("market segment or mission") and *technical* ("managed set of features") issues of the development of SPLs. In this thesis we will only focus on the technical aspects of SPLs. That is, tools and techniques to support developers in the actual process of building, improving or assessing the quality of SPLs.

2.1.1 Origins of Software Product Lines

Software engineering concepts like *modularity*, *separation of concerns* and *levels of abstraction*, that are common practice today, and are sometimes even taken for granted, have actually emerged in the research of *system software* (particularly operating systems) by DIJKSTRA, PARNAS, and HABERMANN [Dij68, Par76, HFC76].

In this context PARNAS defined program families as:

> ... sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members.

There are two important points to note in this definition. First, it states that it is "advantageous" (better design, reuse, etc.) to exploit the "common properties" (commonalties) in a domain. Second, the definition has a technical appeal as it suggests the study of (concrete) *programs* and not the (abstract) domain itself. As we will see in the following sections, the current techniques for SPL development still benefit from and support the first part of PARNAS' definition. However, the second has been extended with the introduction of concepts like *problem domain*, *solution domain* and *variability management*.

2.1.2 Software Product Line Engineering

The advantages of SPLs like systematic reuse, automatic product generation, reduced time to market, etc., come of course with a price. The whole development process has to deal with a large number of additional details. For example, it is necessary to identify what is common to all possible products and what is variable. Also, mechanisms for product specification and generation are required. The testing phase is naturally also more complex as it has to cope with not only one product but a set of them. In order to achieve its goals a successful SPL has to rely on a series of processes, methods and techniques that support its development. In general terms, we define the process that employs such techniques with the goal of producing an SPL as Software Product Line Engineering (SPLE).

In our opinion SPLE has, as suggested by CZARNECKI and EISENECKER [CE00], to support the explicit definition of a *problem* and a *solution* domain. The first deals with the attributes and properties of the domain of interest and defines the set of potential members of the SPL, that is, its variability. The second handles the mapping from these properties to concrete solutions, normally in the form of generators, modules, components, etc. The separation of problem and solution domain is the distinctive characteristic of SPLs in comparison to general configurable software.

For a good separation of problem and solution domain, SPLE is normally divided in *domain engineering* and *application engineering*. The former is responsible for a detailed study of the domain, the design of a flexible architecture and, finally, the development of the *core assets*—the components that are combined to form products. The latter is performed to generate the actual products, it provides mechanisms for product specification and on this basis it assembles and customizes the necessary components to build the desired products.

Domain engineering is normally performed in three main steps. The starting point is *domain analysis* where the domain expert studies all the details of the domain and decides what functionality will be included in the SPL. These pieces of functionality are normally abstracted into *features*, which according to BATORY [Bat05] represent *"an increment of program functionality"* [1]. In this phase all features are gathered and documented together with their interdependencies. The set of identified features is normally organized in a *feature diagram*, which is the outcome of *feature modeling* [KCH+90].

Subsequently, *domain design* takes place, it gets as input the feature model and produces the *reference architecture*, which is the SPL architecture; it has to encompass the whole set of envisioned features and to be flexible enough to allow the generation of all possible products. Additionally, it has to take into consideration all *variation points*. That is, all parts of the SPL that might vary according to the specification

[1] For a detailed discussion about different definitions of feature see APEL and KÄSTNER [AK09]

of a product. Depending on this specification a different set of files must be compiled, specific parts of a source file can be selected, different libraries will be linked to the product, and so on. It is important to note that the variation points might have different granularities, from very coarse to very fine. Based on the reference architecture the *domain implementation* takes place and produces the SPL's *core assets*.

Application engineering is performed once domain engineering is finished. It starts with the *requirement analysis* for the product of interest; these requirements represent the required functionality for the new product. *Product derivation* is the process of selecting the necessary set of features that fulfils the identified product requirements, the result is a list of features—a feature selection—that represents the product specification. *Production generation* uses this specification to coordinate the construction of the product. It coordinates generators, compilers, linkers, to build the product according to its specifications.

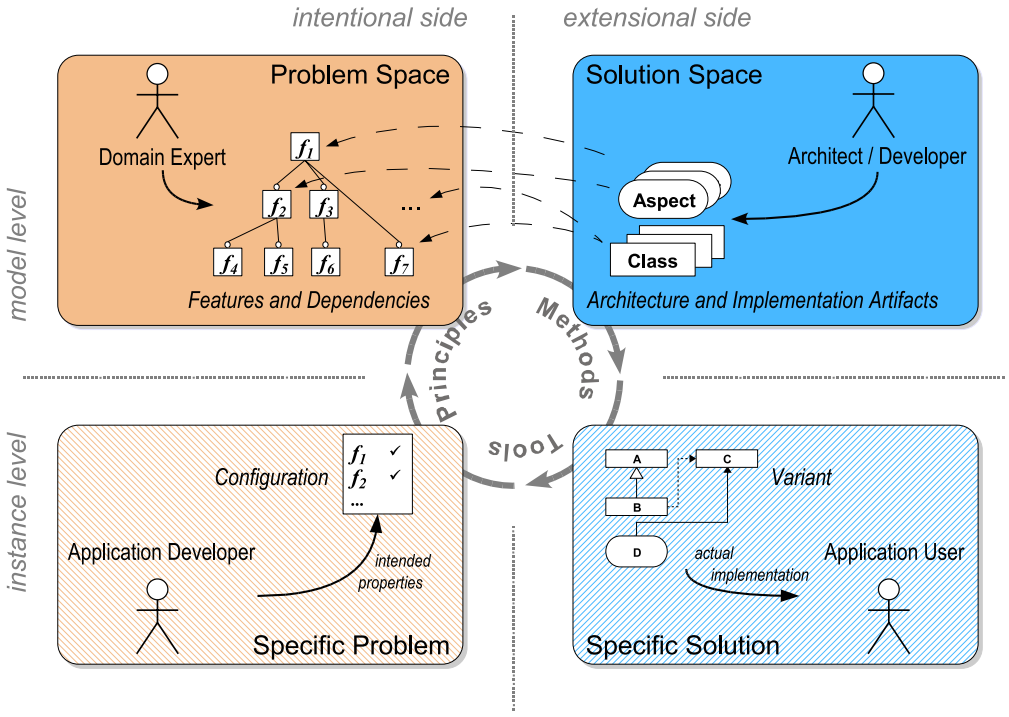


Figure 2.1: Software Product Line Engineering (SPLE) main concepts.

A broadly accepted organization of the concepts previously explained is depicted in Figure 2.1. The upper-left box represents the problem space, that is, the SPL's features, dependencies and variability in general. The upper-right box represents the SPL's core assets, which are the available resources for building products. The con-

nections between these two upper boxes represent the *mapping* from the problem to the solution domain. The lower-left box represents the product derivation process. The lower-right box represents the product (also known as *variant*) that can be generated from the SPL. Note that all boxes are connected through principles, methods and tools that compose the SPLE process.

2.1.3 Variability Management

Activities that deal with variability are present in some form in all phases of SPLE, from domain analysis to the last generation step of a product. The activities that deal with defining, controlling or using variability are generally called *variability management*. SCHMID [SJ04] defines it as follows:

> Variability Management (VM) encompasses the activities of explicitly representing variability in software artifacts throughout the lifecycle, managing dependencies among different variabilities and supporting the instantiations of those variabilities.

Furthermore, SCHMID argues that the following issues should be addressed by a variability management approach: (1) The definition of variation points in the base artifacts, (2) the definition of the elements that can be potentially bound to these variation points, (3) the definition of the relation among variation points and potentially bound elements, (4) the definition of constraints among potential variation point bindings that restrict the potential instantiations , (5) a selection mechanism to define the elements that should go into a specific product.

These are essential tasks in SPLE, according to BOSCH et al. [BFG+01] variability management is the key factor for the success of SPLs. Interestingly, there are variability management techniques that deal with problems in a specific phase of SPLE (one or a few of the five points mentioned above). Also, there are methods that focus on a unified variability management [BPSP04, SJ04] considering the full SPL lifecycle.

This thesis addresses problems in different variability models, and in their interaction. We consider models from the problem space (like feature models, discussed in the following section) as well as from the solution space, for example, models that define variation points in the source code (discussed in detail in Chapter 3).

2.1.4 Feature Models

Feature modeling is a technique for variability modeling where the unit of abstraction is a feature. It aims at identifying and documenting the commonalities and variabilities in a specific domain of study. For this reason it is commonly employed

during domain analysis, for the identification of features and their relationships. This information is organized in the form of a *feature diagram*, which is the artifact that results from feature modeling and guides the subsequent phases in SPLE. Moreover, it is also employed for product configuration, so that the SPL user can pick the features of interest to form the desired configuration.

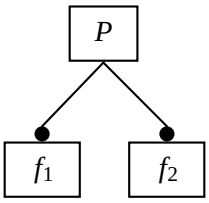
Feature models were originally introduced by KANG et al. [KCH+90]. Since then many extensions [CE00, CHE05] have been proposed and their semantics have been studied in detail [CW07, MWCC08, CHH09]. To facilitate the understanding of feature models we will focus on models with four types of features. *Mandatory features* must always be selected if their parent is selected; Figure 2.2 (a) depicts an example of a mandatory feature. Note that a filled dot marks the feature as mandatory. *Optional features* are those that can be selected if their parent is also selected, these features are connected in the diagram with an empty dot (Figure 2.2 (b)). *Alternative features* are a group of features that are connected to their parent with an empty arc, as shown in Figure 2.2 (c); from this group whenever the parent is selected at most one feature must be selected. *Or features* are a group of features that are connected to the parent with an filled arc (Figure 2.2 (d)), in these groups, whenever the parent is selected, at least one feature of the group must be selected. In short, feature models are graphs where the nodes are the features and one of the four different types of edges connects them. Moreover, each feature model determines the set of products that can be derived from it. Recent studies [STB+04, Ref09] have shown that feature models used in industrial environments might contain up to several thousands features.

The presented notation is one of many available, however, for our discussions about reasoning on feature models it suffices. As we will see in Chapter 4 the key factor to the techniques proposed in this thesis is the use of one central abstraction mechanism. Therefore, a specific notation is not restrictive as long as it can be transformed to the common abstraction format.

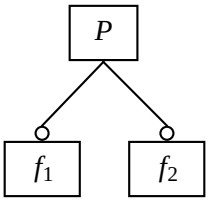
Reasoning in Software Product Lines

As we have illustrated above, feature modeling is the act of identifying and organizing the features of a domain. In principle, a feature model diagram drawn on a piece of paper would already be useful as it provides a compact and clear overview of the full variability. However, such models are even more practical when integrated to the SPL infrastructure so that they can be employed to automate tasks like product configuration or product generation.

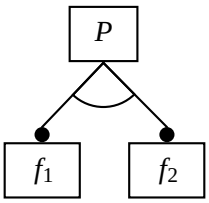
However, in order to support these automated tasks, the semantics of feature models have to be understood so that operations like proving if a specific product configuration is valid can be implemented. Many researchers have studied the rep-



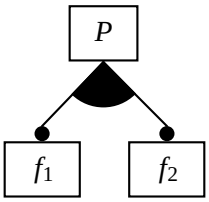
(a) Mandatory features f_1 and f_2 have to be included if their parent feature P is selected. **Formula:** $(P \leftrightarrow f_1) \wedge (P \leftrightarrow f_2)$



(b) Optional features f_1, f_2 can be included if their parent feature P is selected. **Formula:** $(f_1 \rightarrow P) \wedge (f_2 \rightarrow P)$



(c) Exactly one **alternative feature** f_1 or f_2 has to be included if the group's parent feature P is selected. **Formula:** $(P \leftrightarrow (f_1 \vee f_2)) \wedge (\neg(f_1 \wedge f_2))$



(d) At least one **Or feature** f_1, f_2 has to be included if the group's parent feature P is selected. **Formula:** $(P \leftrightarrow (f_1 \vee f_2))$

Figure 2.2: The four most common types of features in feature models: Mandatory, Optional, Alternative Groups and Or Groups.

resentation of feature models in other formats so that such operations become easier to be performed. Feature models have been transformed to *propositional logic* [Bat05, BSTRC06], *Constraint Satisfaction Problem* (CSP) [BRCT05] and Prolog [BPSP04], for example. It is generally accepted that each of these representations have their advantages and disadvantages, and no technique is optimal for all kinds of reasonings [BRCT05, Men09].

The biggest advantage of having the feature model semantics represented in such formats is the availability of well established formalisms, proofs and tooling for automated reasoning. Prolog engines and constraint solvers have been used for problems described in Prolog or in CSP, respectively. In the case of propositional formulas both SAT (boolean SATisfiability problem) [Bat05] solvers as well as BDDs (binary decision diagrams) [MWCC08] have been used as the logic engine for reasoning. Recent studies show [MWC09, Men09] that both approaches scale to feature models with thousands of features.

Reasoning operations have the power to answer questions like:

Is the feature model void? A feature model is void if no valid configuration can be derived from it. Let φ be a feature model represented as a propositional formula, if SAT(φ) is unsatisfiable[2], the feature model is void.

It a product valid? A feature selection that forms a product configuration is valid when it conforms to all constraints imposed by the feature model. Let φ be a feature model represented as a propositional formula, and $f_1 \dots f_n$ a set of n features representing the feature selection, this selection is valid if SAT($\varphi \wedge f_1 \wedge \dots \wedge f_n$) is satisfiable.

Is a feature dead? A feature is dead if it does not appear in any of all possible valid product configurations. Let f be one of the features in a feature model φ, if SAT($\varphi \wedge f$) is not satisfiable then the feature f is dead.

There is a variety of reasoning operations that can be applied to feature models, for a detailed study on them see BENAVIDES et al. [BSRC10].

For the problems tackled in this thesis we decided to use propositional formulas as the abstraction means, and SAT solvers as the reasoning tool mainly for two reasons: (1) there is an abundance of previous works that have defined several relevant reasoning operations on the basis of propositional formulas that we can build upon, (2) studies have shown that for the kinds of problems that we focus on—revealing inconsistencies—not only SAT solvers scale well, it is even considered to be easy [MWC09] to execute feature-model reasonings as SAT problems.

[2]SAT is an operation in logic solvers that gets as input a logic formula and checks whether the formula is satisfiable, that is, if there is an assignment to the logic variables in the formula that makes it evaluate to true.

2.2 The Linux Kernel

Operating system kernels are the pieces of software closest to the hardware. They are responsible for managing the system's resources. This management includes providing an abstraction layer for the hardware (processor, memory, I/O devices, etc.) that has to handle hardware interrupts, memory usage, etc. On top of that the operating system kernel provides services like inter-process communication and system calls to facilitate the development of user applications. The operating system kernel is also responsible for managing the lifetime of application processes and regulating the sharing of resources among them.

The Linux kernel is an open-source implementation of an operating system kernel. It has a monolithic design, that is, it executes the operating system code in one address space. Moreover, it aims towards POSIX and single UNIX specification compliance. Linux Torvalds created the project in 1991 and made it public shortly thereafter under the GNU General Public License version 2. Since then several thousands of developers, ranging from enthusiasts, hobbyists, scientists, to employees of small, medium and large IT companies, have contributed to the project. The interactions among this heterogeneous group of developers, with different backgrounds, goals and interests, plus the lead of Linus Torvalds (together with other key maintainers) have converged to an unorthodox—and rather successful—development process.

The development of the Linux kernel has certainly evolved in its two decades of existence, however, its essence has changed little. The developers use public mailing lists to communicate and any individual that wants to contribute has just to send its code to the list, if it catches the attention of other developers it can eventually become part of the official kernel. Suggestions for code improvement, for new functionality, bug reports or bug fixes are performed in the same fashion. However, the way up to the official tree (maintained by Linus Torvalds) might be long.

The kernel developers are hierarchically organized. On the top is the project leader who maintains the official Linux kernel tree, below him there are the subsystem maintainers, followed by the driver/file maintainers, and then the developers/-contributors. This structure forms a trust hierarchy, code flows from the first level to the top according to how it is judged, regarding quality and usefulness, by the other users, developers or maintainers. This open and democratic process has attracted a large number of individuals and companies to actively contribute to the project. As a result, Linux today—after two decades of development—offers an overwhelming amount of features.

The development does not follow a strict *roadmap*. According to Linus Torvalds the kernel is "evolution and not intelligent design". In this evolution process along the years the developers not only had to cope with the natural evolution of the avail-

able functionality, but also had to accommodate the new functionality and add support to a large number of new hardware platforms (emerging platforms as well as legacy platforms of interest). To make things even more complicated, as the kernel optimizes the use of resources under its control, it cannot impose any overhead caused by unnecessary features. It means that as new functions are added, they must be made well configurable, so that the user is able to finely customize the kernel according to his requirements, and no collateral overhead of unneeded features occur. As a result, *variability management* is a crucial task in the development of the Linux kernel.

The Linux developers are known to prefer working with tools that are specifically tailored for the kernel development. That is, the community around the kernel has developed their own set of tools to support the kernel development. The most prominent (even outside the kernel community) are: The static analysis tool `Sparse`, which is a *semantic compiler* for the C language that is able to perform several kinds of source code analysis. `Kbuild` is a *build system* for the kernel, it controls the build process by selecting files for compilation and linking. The `Kconfig` is the *kernel configurator*, its tools enable the user to customize the kernel according to his requirements. `Git` is a *distributed revision control system*. The development of all these tools was started by kernel developers to improve the kernel development infrastructure. However, all of them have made their way in several other software projects. As we will see in the following sections, some of these tools play a key role in the *variability management* of the Linux kernel.

2.2.1 Organization

One of the key factors for the success of the Linux kernel is its capability that it can be efficiently customized to a diversity of user needs and deployment scenarios. It has a modular structure that supports both static and dynamic customizations. In the following we describe the variation points of the Linux kernel from coarse to fine granularity:

Hardware Architecture An operating system kernel is the software abstraction closest to the underlying hardware. The configuration of the target architecture and platform is an essential task, and, therefore, generally it is the first feature to be set in the configuration process. The Linux Kernel offers support for more than 25 hardware architectures (e.g. Sparc, Alpha, i386, etc.) that are subdivided in about 60 different hardware platforms (e.g. ia32, DEC, sun-4, etc.).

Subsystems The Kernel subsystems are organized as follows: `kernel` (architecture-independent kernel code, e.g. IRQ-handling, process scheduler, etc.), `fs` (file

systems implementation), `init` (kernel initialization routines), `mm` (memory management), `sound` (sound subsystem), `block` (abstraction layer for disk access), `ipc` (inter-process communication code), `net` (network protocols), and `lib` (library functions, e.g., CRC and SHA-1 algorithms). During the configuration process all subsystems can be fine tuned, some like `sound` or `net` can be disabled as a whole, and others like `kernel` or `mm` cannot.

Device Drivers Device drivers enable the operating system to interact with a specific hardware device. This is the biggest subsystem in the Linux Kernel, which is due to the vast number of devices it supports. For example, USB devices, network interface cards, PCMCIA cards, and a lot more. During the configuration process, besides being enabled, customized, or disabled, a device driver can be set as a loadable kernel module. Device drivers configured as loadable modules can be dynamically added, that is, during runtime these device drivers can be loaded and unloaded on demand to the kernel memory space.

Config Options are responsible for enabling, parametrizing or disabling specific features on all levels of granularity. They define the variability of the project just like feature models in SPLs, therefore they can be classified as a problem-space asset. By assigning values to config options in the configuration process the desired architecture is set, subsystems and device drivers are selected. Moreover, compilation, linking and debug options can also be defined via config options.

The Linux kernel developers not only have to deal with the *specification* of this large amount of variability, but also have to *implement* it efficiently. That is because in the realm of system software—especially in operating systems—overhead imposed by undesired features is unacceptable. Since the operating system is responsible for managing the systems' resources, it should be implemented in a way that it optimizes the use of all available resources. The kernel community has developed tailor-made tools to meet these requirements.

2.2.2 The Kernel Configurator — KCONFIG

History In 2001 the kernel developers started to show dissatisfaction with the kernel configuration tool, back then known as Configuration Menu Language (CML1). With the growth of the kernel, the configuration process was getting very complicated. The tool was responsible for selecting the capabilities to be built into the kernel, handling dependencies and providing the user interface for feature selection. Moreover, it was comprised of a mixture of code written in Tcl/Tk scripts, Awk scripts, Perl and C, which made it hard to understand and to maintain [ker05].

In order to solve this problem, a Configuration Menu Language 2 (CML2) was proposed. It is a *mini-language* designed specifically for configuring kernels. A *ruleset* describing all the available options and their dependencies is translated into a *rule-base* that is read by the front-end in order to configure the kernel[Ray]. After more than two years of development, several *flame wars* on the mailing list, and many improvements over the previous system, the project was dropped and not accepted in the official kernel tree. This shows how restrictive and demanding the community is regarding new code being merged in the official tree. Nevertheless, the source code is still available and it is used as the configuration tool of other projects.

A couple of months after the discussions about the CML2 had finished, the *Kernel Configurator* (KCONFIG) was proposed to address the shortcomings of both CML1 and CML2. According to the KCONFIG author, the major advantages over CML2 are: (a) it is written in C code (CML2 is written in Python which makes a Python interpreter to be delivered with the kernel) (b) a tool for the automatic conversion of the CML1 configuration into the new one is included, (c) it is less complex then CML2, since it does not try to address issues like facilitating the configuration process for non-experts as the CML2 does.

As these three points were of great importance for the linux developers, after around one year of testing and improvements, the KCONFIG was accepted and merged in the kernel `2.5.45`.

Kconfig Language KCONFIG is basically comprised of a *parser* and a *dependency checker*, which are used as the back-end for the KCONFIG toolset. The features of the Linux kernel are defined as KCONFIG *entries* [3], the set of all entries constitutes the *configuration database*. This database plays the same role for the Linux kernel as features models do for SPLs. In other words, it defines the variability of the problem domain.

The kernel developers use the KCONFIG language to define new entries. Normally, entries that are related to a subdomain of the kernel (like an architecture or subsystem) are defined together in the same text file. The entries have their own dependencies and properties, that can be defined using the following constructs of the KCONFIG language:

type defines the type of an entry, which can be `boolean`, `tristate`, `string`, `hex` and `integer`. Boolean entries are those that can be only enabled and disabled, e.g., for the selection of a subsystem. Tristate entries are used for device drivers that can be compiled as a loadable kernel module. String, hex, and integer entries are used for kernel constants like buffer sizes, absolute memory addresses, and so on.

[3] The terms feature, config option and kconfig entry can be used interchangeably.

```
config FLATMEM_MANUAL
        bool
        prompt "Flat Memory"
        depends on !(ARCH_DISCONTIGMEM_ENABLE
                    || ARCH_SPARSEMEM_ENABLE)
                    || ARCH_FLATMEM_ENABLE
        help
          This option allows you to change some of the ways that
          Linux manages its memory internally.  Most users will
          only have one option here: FLATMEM.  This is normal
          and a correct option.
```

Figure 2.3: An entry defined in the KCONFIG language.

input prompt is the entry's visual name that is displayed to the user during the configuration process.

default values are assigned to the entry if no value was set by the user and the dependencies are met.

dependencies define the conditions that must be met so that the entry is selectable. Dependencies can be as simple as a single other entry, or as complex as an expression of several entries connected with the operators && (logical and), || (logical or) and ! (logical not).

reverse dependencies are used to force the selection of other entries when the entry where it is defined is selected.

numerical ranges limit the range of possible input values for `integer` and `hex` symbols.

help text defines the entry help text to be shown during configuration.

An example of a KCONFIG entry is shown in Figure 2.3. In this entry the feature `FLATMEM_MANUAL` is defined to be of type boolean, it will be shown during configuration to the user with the label "Flat Memory", and to be selectable the expression defined in the option `depends on` must evaluate to true (this operation is performed by the dependency checker). A help text to be shown during configuration is also provided.

Moreover, in order to provide a better organization of the entries in the configuration tree that is displayed to the user, the following constructs are also allowed:

menu entries defined between the keywords `menu` and `endmenu` are grouped together and displayed in a separate window. It may also have an attribute `prompt` to name the groups of entries.

choice constructs are defined with the keywords `choice` and `endchoice`. Only one entry of those defined between these keywords can be selected if its parent entry is also selected.

In the current Linux kernel version `3.2` there are 879 KCONFIG files that altogether correspond to more than 100,000 lines of text describing 11,000 KCONFIG entries. Each architecture has a main KCONFIG file and via an inclusion mechanism other KCONFIG files (for subsystem, drivers, and so on) can be recursively added. As a result, the Linux kernel does not have one large variability model, but one per architecture. Thereby, architecture-specific entries are isolated in specific models and architecture-independent entries are shared among many models.

2.2.3 Build System — `Kbuild`

KBUILD is the build system employed in the Linux kernel. It is implemented in the MAKE language and is responsible for driving the compilation process. In simple terms, its operation can be seen as follows: First, it reads the set of features that were selected by the user during the configuration process. Subsequently, a set of makefiles are processed and only the relevant source files or whole subdirectories are included in the compilation process according to the selected features. As an example, we show below the file `arch/ia64/mm/Makefile`:

```
1 obj-y := init.o fault.o tlb.o extable.o ioremap.o
2
3 obj-$(CONFIG_HUGETLB_PAGE) += hugetlbpage.o
4 obj-$(CONFIG_NUMA)          += numa.o
5 obj-$(CONFIG_DISCONTIGMEM) += discontig.o
6 obj-$(CONFIG_SPARSEMEM)    += discontig.o
7 obj-$(CONFIG_FLATMEM)      += contig.o
```

The first line contains a set of files that will be unconditionally compiled whenever this makefile is processed. This is because the filenames after the := symbol are simply added to the `obj-y` list, which is the list of files to be compiled. In the following lines, from 3 to 7, the files after the sign +=, can be added to three different lists depending on the values of the variables prefixed by `CONFIG_`. The values of these variables depend on the feature selection and can be `y`, `n`, or `m`, for an enabled feature, for a disabled feature, and for a feature defined to be compiled as a loadable module. This way, the three possible lists to which the filenames can be added are `obj-y` (files to be compiled in kernel), `obj-n` (files that will not be compiled), and `obj-m` (files to be compiled as loadable modules).

The makefile snippet shown above contains basic functionality of the powerful KBUILD system. Advanced KBUILD concepts include: Automatic generation of dependencies, mechanisms for cross-compilation, linking, postprocessing, etc. However, our goal here is not to study every detail of KBUILD but to highlight the connections between the KCONFIG entries (through the `CONFIG_` variables) and the build

system. That is, the technical solution behind the *mappings* from the problem space and the solution space.

2.2.4 Heterogeneous Variability Management

Another important source of variability in the Linux kernel is the source code. In Linux the source code is annotated with CPP directives to mark parts that have to be compiled only in specific circumstances, that is, only when specific features are selected. The details of the CPP language and usage is presented in the next chapter. Important to note is the heterogeneous characteristic of the variability management in the Linux kernel. As we have seen above, three different tools operate on the same set of features, namely, CPP, KCONFIG, and KBUILD. These operations are on the same syntactical and logical level. That is, the flag `CONFIG_SMP` represents the same feature across all tools. However, these tools operate independently from each other.

2.2.5 The Linux Kernel from the SPL Perspective

Reference techniques for SPLE [NC01, BKPS04, PBvdL05] are comprised of guidelines for analysis and design that allow industrial SPL manufacturers to keep the development controllable and efficient. The outcomes of these activities are *explicit* definitions of commonalities and variation points of the SPL on the requirements, software architecture, and implementation artifacts.

In contrast, the Linux Kernel development process does not employ a uniform domain engineering process. Commonalities and variation points often emerge *implicitly* from implementation necessities. Even more profound Kernel evolution issues like new features or software architectural changes do not undergo a controlled planning process. To quote the Linux kernel creator Linus Torvalds: *"Linux is evolution, not intelligent design"*. However, it does not mean a chaotic development process. On the contrary, the decisions are democratic and involve many developers and maintainers. The key difference to SPLE is that the variability is not taken into consideration beforehand. We consider the SPLE process to be a top-down approach, that is, at the highest level we have the features and their dependencies, and from them, the architecture, and variability realization are derived. In the Linux kernel, first the new features or extensions are implemented by a developer or a group of them, and only after that, the results are integrated to the variability model, sometimes even by a different stakeholder (e.g. a subsystem maintainer). As a result, we regard the Linux kernel development to be a bottom-up approach with respect to variability management.

This unorthodox development process is supported by very large manpower:

several thousands of volunteers (alongside with full-time paid engineers in contributing companies) stand by to implement new features, to review code changes, to do kernel-wide interface refactorings, or to support the release process by beta testing the kernel on their machines [Mor05, vG06]. Another difference to SPL projects in industrial environments is how the community deals with milestones and deadlines. There are no real release deadlines (although Linus Torvalds established a release cycle for the 2.6 kernel series), a release is *"done when it's done"*. This is of course a luxury that industrial SPL projects cannot afford.

A few years ago we posed the question: *"Is the Linux kernel a Software Product Line?"* [SSSPS07]. At that time we concluded that on one hand we could not classify the Linux kernel as an SPL because it is developed in a way that is very conflicting with the SPLE guidelines. On the other hand, we could consider it to be an SPL because it is a highly configurable software system, with a huge amount of variability, that allows users to customize specific kernel variants. Moreover, it contains an elegant separation of the problem and the solution space. Basically, the decision to consider or not the Linux kernel to be an SPL hinges on whether the *development process* or the *resulting software* is more important. We believe the latter is of most importance, therefore, in our opinion the Linux kernel is an SPL.

Revisiting this question today, we are even more convinced that the Linux kernel can be seen as an SPL mainly due to two reasons. First, the Linux kernel has served us very well for the implementation and evaluation of new SPLE techniques [STLSP10, TLSSP11]. Second, a series [LAL+10, SLB+10, SB10, BSL+10a, ZK10] of works have followed our initial efforts to use the kernel in the context of SPL research. Resulting in a number of new techniques, analyses and evaluations that have contributed to a better understanding and advance of SPLE.

2.3 Problem Overview and Goals

> "`#ifdef`'s sprinkled all over the place are neither an incentive for kernel developers to delve into the code nor are they suitable for long-term maintenance."
>
> Thomas Gleixner, ECRTS '10

> "The kernel is huge and bloated... and whenever we add a new feature, it only gets worse ."
>
> Linus Torvalds, LinuxCon'09

As detailed in the previous sections the *description* of variability tends to span over a diversity of artifacts in the code base. In *evolvable* systems it is difficult to impose one unique abstraction mechanism to describe variable parts of the software project. For example, variability in code (defined and used by developers) and variability in the configuration model (defined by developers/designers and used by end users) have different requirements. When choosing the variability mechanisms used to tailor source code, requirements like: separation of concerns, modularity, code tangling, composition overhead, etc., have a high influence on the design decisions. On the other hand, the variability implementation techniques for configuration models have completely different requirements that focus on the user's needs like attractive graphical interfaces, response time, error-resolution messages, model scalability, etc. Inevitably, in large projects there will be *variable* artifacts that have different requirements regarding how to describe such *variability*; hence, it is not only difficult—but many times totally undesirable—to describe the full variability with one single language or tool. Another disadvantage of centralizing the description of variability is the loss of flexibility regarding the integration with other projects as everything must be managed by the same tool.

Nevertheless, even though it is very often desired to have the flexibility to describe the variability of different artifacts with the most appropriate mechanisms, this flexibility comes with a price. The variability specified in different formats is closely related to each other, but is kept separately. As a consequence, there may be implicit constraints and dependencies, which are difficult to check as they are described in different formats and handled by different tools.

As described in Section 2.2.5, the Linux kernel can be seen as an SPL mainly due to its elegant separation of the *problem domain* and the *solution domain*. However, also as a result of our initial analysis, we have identified that due to its heterogeneous variability management, and lack of integration among its internal tools that deal with variability, it is likely that there are inconsistencies among the variability spread over its artifacts. Therefore, we clearly see the need to better integrate the variability descriptions that are spread over the different artifacts, and, also important, to offer reasoning operations (as detailed in Section 2.1.4) so that the inconsistencies can be automatically detected.

Regardless whether the decision to employ a heterogeneous variability management is motivated by design decisions or it is simply the result of software evolution, if this variability is not properly analyzed it will inevitably diverge with time. In this thesis we tackle this problem by:

- providing empirical evidence that description of variability in different formats leads to inconsistencies.
- providing a methodology to convert such different formats to a unified ab-

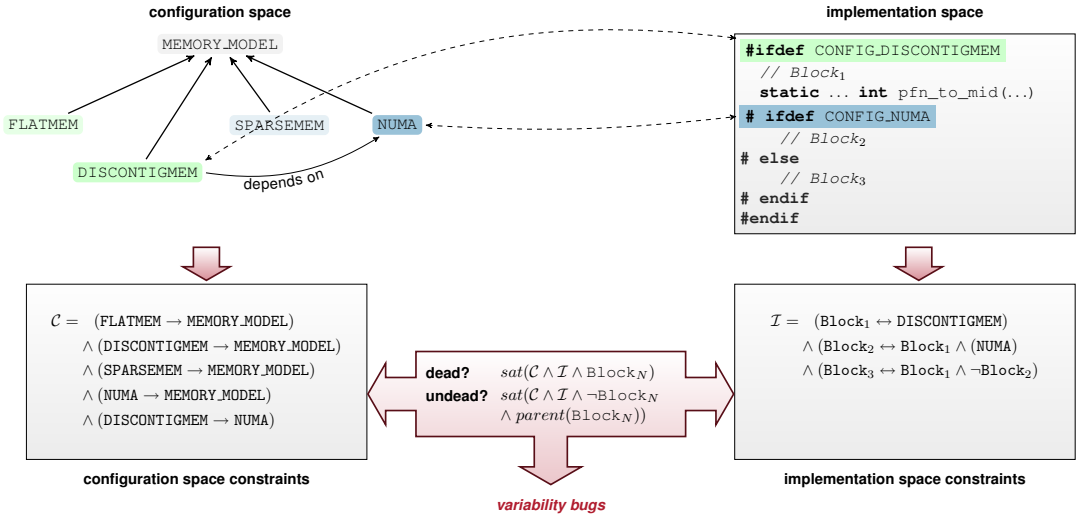


Figure 2.4: Our approach at a glance: The variability constraints defined by both spaces are extracted separately into propositional formulas, which are then examined against each other to find inconsistencies that we call *variability bugs*.

straction mechanism in order to perform reasoning operations that are able to reveal variability bugs.

- applying the concepts to real-world projects and make the corresponding adjustments to cope with realistic (and possibly very large) models.
- evaluating the developed tools and methods not only quantitatively but also qualitatively, that is, gathering feedback from developers responsible for the code where variability bugs have been found by our tools.

In summary, we concentrate on the development of methods and tools that enable the identification of variability bugs in software projects that have their variability spread over several artifacts in different formats. Special attention is given to the CPP, which is the *de facto* standard variability management tool employed in the development of system software. An overview of the approach that is formed by the contributions of this thesis is shown in Figure 2.4. In the upper part we see excerpts of actual artifacts of the Linux kernel, on the left side the configuration model, and in the right side a source file. We can see dependencies among them as they are built based on a common set of features (in this case the features DISCONTIGMEM and NUMA). The potential problem in this scenario is that the dependencies among these artifacts are only *implicit*, as the tools that handle them are independent of each other. As we mentioned above, our idea is to convert the variability of the different

artifacts to a common format so that we can apply reasoning operations to reveal inconsistencies. This is shown in the lower part of Figure 2.4, we have on the lower left and lower right side the constraints of the two different artifacts converted to boolean logic. Representing the constraints from different sources as boolean formulas allows us to perform reasoning operations as shown in the center of the figure. In this example we show only two models for the sake of simplicity, but our techniques enable us to combine as many models as needed in concrete scenarios.

Furthermore, state-of-the-art SPLE approaches have given little attention to the integration of CPP-based artifacts into techniques that handle variability reasoning. Also, desired properties like scalability (of model size), applicability (integration with existing build systems), effectiveness (will variability bugs really be found), impose further challenges that have to be addressed since we focus on real-world large-scale projects. In this context, the development of methods and tools to efficiently identify and avoid variability bugs and the identification of variability bugs in real systems form the aspirations of this thesis. Consequently, by achieving our postulated goals, we expect to advance the state of the art in the field of variability management for system software.

The title of this thesis combines a subject, *variability bugs*, and a context, *system software*. The subject simply denominates the type of software problem that we address in this thesis by providing techniques and tools to reveal and avoid them. We decided to stress the context, system software, mainly due to two reasons: (1) among the techniques presented here, the integration of CPP-based artifacts into the variability management process plays a central role, and the CPP is the variability realization technique of choice for the majority of system software like operating systems, databases, middleware, etc. (2) The case studies presented in this thesis were carried out exclusively on open-source system software, this decision was motivated by the challenges imposed by this domain (see Section 2.1.1).

"Simplicity is prerequisite for reliability."

Edsger W. Dijkstra

3 Source Code Variability Models

In this chapter we introduce the concept of *source code variability models*, which are variability models automatically extracted from annotated source files. These models allow a better integration of the source code artifacts into the variability management process. We employ propositional logic as abstraction means, as a result, we are able to perform several reasoning operations to reveal inconsistencies. *This chapter shares material with the GPCE'10 paper 'Efficient Extraction and Analysis of Preprocessor-Based Variability'[STLSP10].*

3.1 Introduction

The CPP is the tool of choice for a diversity of domains. Many [SC92, KS94, mF95, EBN02, Loh09, Käs10] have studied its characteristics. Although it is debatable what are its strong and weak attributes, its success in terms of popularity and adoption is unquestionable. Regardless of its technical merit or impact on software projects, the CPP is present in thousands[1] of important projects like the Linux kernel, the Apache web server or the Mplayer media player. For this reason any improvement on the usability of the CPP can have a direct impact on a large base of users and developers. Moreover, as we discussed in Section 2.3, and others elsewhere [KATS11, KGO11], the lack of integration of the CPP into variability management is still a research problem that needs further investigation. In this context, we aim at (1) studying its usage for variability implementation, (2) understanding how to better integrate it to the variability management process (3) providing techniques and tool support to reveal variability problems, and (4) improving the overall usability of the CPP for highly-configurable software projects.

3.2 The C Preprocessor as a Variability Realization Technique

Several researchers have studied the different options [GA01, MO04, SvGB06] to realize variability in the source code, that is, how to associate specific parts of the source code to features of the system. Prominent examples are aspect-oriented programming, feature-oriented programming, aggregation, delegation, inheritance, parametrization, meta-programming, etc. The CPP is one of the well-known mechanisms to realize variability, and its use for this purpose [JB02, Loh09, Käs10, LAL+10] as well its general use as a macro processing tool [SC92, KS94, mF95, EBN02] is a topic of academic research. Although its advantages and disadvantages are still a topic of hot debates, we simply accept the fact that it is broadly used in practice and it still is the *de facto* standard for domains such as operating systems and embedded systems. For this reason, we pay special attention to CPP's capabilities for variability realization, and usage patterns in established and successful CPP-based software product lines.

In this context, LIEBIG et al. [LAL+10] studied forty open-source, CPP-base software product lines. The authors analyzed millions of lines of code in popular open source projects in order to investigate CPP usage patterns. Using several metrics

[1] According to the TIOBE index (`http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`) as of this writing the programming languages `C`, `C++` and `Objective-C` are among the five most popular programming languages, in first, third and fourth respectively.

they observed that the CPP's mechanisms are frequently used to implement optional and incremental code. Also, they found that these extensions are primarily of coarse granularity and their complexity is not related to the project size.

KÄSTNER [Käs10] classifies the CPP under the more general concept of *annotative* approaches for variability realization. According to him these approaches are implemented by "annotating code fragments with features in a code base". Variants are generated by removing annotated code fragments, which is known as *negative variability*. Still according to him, such approaches have limitations in the following areas: modularity, traceability, language integration and error detection, and stand out in areas like: fine granularity, simplicity, uniformity and adoption. Similarly, LOHMANN [Loh09] classifies the CPP as a *decompositional* approach. He argues that the CPP is different from other variability realization techniques in two respects. First, the CPP has no relation to the type system of the underlying language, which enables fine-grained annotations. Second, as a decompositional approach, the CPP-based source files represent the union of all possible variants they can generate. The unnecessary parts are filtered out as required. He concludes that the decoupling of macro language and type system makes the approach overhead free, as no abstraction of the programming language takes part in the customization process and, therefore, it is attractive to system software developers.

We have also contributed to the analysis of the CPP in the context of variability management [STLSP10, TLSSP11]. We argue that although the CPP has been criticized many times for leading to unmaintainable code, there must be a reason for its long tradition as a tool of choice to implement variability in the realm of C and C++. Evidently, its advantages either outweigh its disadvantages, or its disadvantages can be circumvented. We believe that the former is the case. For example, in the development of the Linux kernel, the usage of the CPP is regulated by the project's *coding guidelines*. Such guidelines establish which constructs and usage patterns are allowed when implementing variability with the CPP. This is one way to avoid language abuse, as code which does not conform to the guidelines is not accepted into the mainline. However, this is not enough. In our opinion, the guidelines should be supported by appropriate tooling (as discussed in Section 2.3). In the following we present the formalism that will support the creation of tools to better integrate the CPP into the variability management process and, as a result, enable the automatic detection of variability problems.

3.2.1 The C Preprocessor Language

The C preprocessor (CPP) is a macro processor that is employed to transform the source code prior to compilation. It works on lexical level, that is, it transforms an input character sequence into an output character sequence. It is called a macro

processor because it allows the definition of *macros*, which are compact abbreviations for long constructs [cpp]. Although it is mainly used by C and C++ projects, the CPP is actually language independent and can be used with any programming language or text file.

The CPP is controlled by *directives* that form its language and can be used to perform the following operations:

- Inclusion of header files. In C and C++ programs it is common practice to separate parts of the program in reusable units that contain forward declarations, the so called header files. The CPP replaces a file declaration with its contents. Directive: `#include`.

- Macro expansion. A macro, representing an arbitrary code fragment, can be defined and its usage in the code is replaced by its definition throughout the code by the CPP. Directive: `#define`.

- Conditional compilation. Specific parts of the code can be included or excluded to the CPP's output according to various conditions. Such conditions can be built by using single macros or a combination of them. Directives: `#ifdef`, `#if`, `#elif`, `#else`, `#endif`.

- Line control. Mechanisms to inform the compiler about line numbers when files are combined together during preprocessing. Directive: `#line`.

- Diagnostics. Mechanisms to issue error or warnings during preprocessing. Directives: `#error`, `#warning`.

These operations are triggered by the usage of preprocessing directives, which are lines in the source code that start with the character '`#`' and is followed by the directive name.

To illustrate the use of CPP directives to implement variability, in Figure 3.1 we show an excerpt of the file `linux/kernel/sched.h`. On lines 1–5 we see examples of the file inclusion, that is, the files listed after the directive will have their content inserted on the place where they are declared. On lines 14–16 we see the definition of macros that can be used in subsequent lines. From line 21 to 51 a `struct` is defined, it contains 6 fields that will be always present (unconditionally compiled), they are defined on lines 22–23 and 38–42. The remaining fields defined inside this `struct` are annotated with conditional-compilation directives. They will be included in the output only when specific macros (or combinations thereof) are defined. For example, the fields defined in lines 33–36 will be included only when a macro with the name `CONFIG_SMP` is defined. Note that macros can have a name, parameters and

```
#include <linux/sched.h>
#include <linux/mutex.h>
#include <linux/spinlock.h>
#include <linux/stop_machine.h>
#include "cpupri.h"

extern __read_mostly int scheduler_running;

/*
 * Convert user-nice values [ -20 ... 0 ... 19 ]
 * to static priority [ MAX_RT_PRIO..MAX_PRIO-1 ],
 * and back.
 */
#define NICE_TO_PRIO(nice)      (MAX_RT_PRIO + (nice) + 20)
#define PRIO_TO_NICE(prio)      ((prio) - MAX_RT_PRIO - 20)
#define TASK_NICE(p)            PRIO_TO_NICE((p)->static_prio)

[...]

/* Real-Time classes' related field in a runqueue: */
struct rt_rq {
        struct rt_prio_array active;
        unsigned long rt_nr_running;
#if defined CONFIG_SMP || defined CONFIG_RT_GROUP_SCHED
        struct {
                int curr; /* highest queued rt task prio */
#ifdef CONFIG_SMP
                int next; /* next highest */
#endif
        } highest_prio;
#endif
#ifdef CONFIG_SMP
        unsigned long rt_nr_migratory;
        unsigned long rt_nr_total;
        int overloaded;
        struct plist_head pushable_tasks;
#endif
        int rt_throttled;
        u64 rt_time;
        u64 rt_runtime;
        /* Nests inside the rq lock: */
        raw_spinlock_t rt_runtime_lock;

#ifdef CONFIG_RT_GROUP_SCHED
        unsigned long rt_nr_boosted;

        struct rq *rq;
        struct list_head leaf_rt_rq_list;
        struct task_group *tg;
#endif
};
```

Figure 3.1: Excerpt of the file `linux/kernel/sched.h` showing several uses of different CPP directives

body (like the ones in lines 14–16), just a name and body, or just a name. Macros that have only a name are also called compilation flags.

What we see in Figure 3.1 is just a small excerpt of the original file. This file has actually 9,371 lines of code and makes use of 515 CPP directives. By issuing the command `cpp -include config.h sched.c` the CPP will preprocess the file and output the result that can be used for compilation. However, the option `-include config.h` will include the contents of the file `config.h` at the first line of `sched.c`. This is the mechanism used by the Linux kernel to inform the CPP what are the features that were selected during configuration. An excerpt of such a configuration file in shown in Figure 3.2. As we can see the selected features are defined as CPP macros. Nevertheless, the CPP is not able to give answers to questions like:

- How many different outputs can be generated from a file?
- Are all conditional blocks included in at least one of the possible outputs?
- Is there any output where a specific conditional block is not selected?

```
1  /*
2   *
3   * Automatically generated file; DO NOT EDIT.
4   * Linux/x86_64 3.3.0-rc2 Kernel Configuration
5   *
6   */
7  [...]
8
9  #define CONFIG_X86_64_SMP 1
10 #define CONFIG_PM_SLEEP_SMP 1
11 #define CONFIG_USE_GENERIC_SMP_HELPERS 1
12 #define CONFIG_SMP 1
13 #define CONFIG_HAVE_TEXT_POKE_SMP 1
14
15 [...]
```

Figure 3.2: Excerpt of a configuration file from the Linux kernel showing the definition of features that were enabled during configuration.

These are questions that can normally be answered with the reasoning techniques presented in Section 2.1.4. Moreover, answers for such questions require information from other tools. For example, the constraints between features defined in the KCONFIG model have also an impact on the conditional blocks that make use

of these features. In the following we discuss the basic characteristics of conditional compilation focusing on how to answer the questions posed above, and also, how to integrate the CPP with other tools like KCONFIG.

3.2.2 Quantifying Variability

Our idea is to use boolean functions to answer this type of questions we presented in the previous section. Before we give an example of such a boolean function, it is important to understand the components of a CPP-based source file. To this end, we inspect a few very basic examples of CPP usage.

The following listing shows the most trivial example of a conditional block:

```
1 #ifdef CONFIG_SMP
2 //block 1
3 #endif
```

The number of variants that can be composed from this code snippet is two: Either the item `CONFIG_SMP` is set, then `block 1` is selected, or if `CONFIG_SMP` is not set, then `block 1` is skipped. The next listing shows alternative blocks:

```
1 #ifdef CONFIG_SMP
2 //block 1
3 #else
4 //block 2
5 #endif
```

Here, we have again in total two variants that can be composed. Depending on the definition of `CONFIG_SMP`, either `block 1` or `block 2` is selected. Another use of if-groups can be seen in the following listing:

```
1 #ifdef CONFIG_SMP
2 //block 1
3 #elif defined CONFIG_APIC
4 //block 2
5 #endif
```

In this example there are three variants that can be composed: If both items are unset, then none of the blocks are selected. `Block 2` is selected only if `CONFIG_SMP` is unset and `CONFIG_APIC` is set. If both flags are set, then `block 1` gets precedence.

Important for the understanding of our approach is the concept of code *block*. It represents the snippet of code that is controlled by conditional compilation. The variability of a `CPP`-based compilation unit is basically the set of all possible block combinations that the `CPP` can generate for the compilation unit.

We aim at modeling this variability with a boolean formula that contains logic variables representing both the CPP flags used in #if expressions and also the code blocks. In anticipation of our approach for the automatic generation of these formulas, we present a simple example of such a formula that mimics the CPP semantics. Consider the last code listing presented above. In this listing we see the use of two CPP flags: CONFIG_SMP and CONFIG_APIC. They are used to define two[2] code blocks. To understand how this boolean variables can model the CPP semantics lets take a look at the truth table shown in Table 3.1. The first two columns show all possible ($2^2 = 4$, the total of lines in the table) combinations of what is the *input* that is given to the CPP, that is, how the CPP flags can possibly be set. The following two columns show what is the *output* of the CPP, that is, it shows whether, given the input flags, the respective blocks will or will not be selected by the CPP. The value 1 represents the block selection and 0 represents a skipped block. At each line, the last column indicates whether the values in the previous columns correspond to the CPP semantics. For example, in the first line none of the flags are set and none of the blocks are selected, this is exactly the expected behavior, that is why we have 1 in the last column of the first line. In the second line we have the flag CONFIG_ACPI set and the *Block 2* as selected, this also represents the correct behavior. The following lines also reflect the correct behavior, this is why in all lines we have 1 in the last column.

CONFIG_SMP	CONFIG_ACPI	*Block 1*	*Block 2*	f_{CPP}
0	0	0	0	1
0	1	0	1	1
1	0	1	0	1
1	1	1	0	1

Table 3.1: A truth table representing a expected CPP behavior.

However, the full truth table considering the two CPP flags and the two code blocks should contain in total 16 entries ($2^4 = 16$). The excerpt of this full true table shown in Table 3.1 contains only those combinations that correspond to the expected behavior of the CPP. For any other combination of values in the first 4 columns we had to mark the last column as 0, as it would not correctly represent the CPP semantics. Consider the boolean function $f(\texttt{CONFIG\_SMP}, \texttt{CONFIG\_ACPI}, \textit{Block1}, \textit{Block2})$, by assigning values to these four inputs the function f has to output true (1) for the inputs shown in Table 3.1, and false (0) for any other combination. Just to illustrate: $f(0, 0, 1, 0) = 0$, because this assignment means that none of the flags are

[2]We could also consider the *possible* code blocks above the definition of the first #ifdef and below the #endif, but for simplicity's sake they are disregarded in this example.

set and *Block 1* is selected. This is not correct, hence the function returns false. The boolean function that corresponds to the expected CPP behavior for this example is defined as follows: $f(\texttt{CONFIG\_SMP}, \texttt{CONFIG\_ACPI}, \mathit{Block1}, \mathit{Block2}) \stackrel{\text{def}}{=} (\mathit{Block\ 1} \leftrightarrow \texttt{CONFIG\_SMP}) \wedge (\mathit{Block\ 2} \leftrightarrow (\texttt{CONFIG\_ACPI} \wedge \neg \mathit{Block\ 1}))$.

In the following sections we show how to automatically generate this formula given any CPP source file. We use some formalisms so that our technique can be re-implemented by others.

3.3 Generating Variability Models from CPP-Based Artifacts

In this section we describe our technique that can automatically extract the variability model of any file annotated with CPP code. Such a generated model must precisely describe the variability specified by the CPP commands. As a means of abstraction we employ propositional logic. Several authors [Bat05, MWC09, SLB+10] have also employed propositional logic to describe the variable artifacts of software product lines. Currently, propositional logic can be seen as the *de facto* standard to implement reasoning operations that handle variability in software product lines. Although alternatives have been proposed as well [BRCT05, JK07].

After presenting a few basic definitions in the following section, we introduce the algorithm to convert annotated files to variability models described as propositional formulas. The algorithm handles the CPP directives `#ifdef, #if, #elif, #else`. This was motivated by the variability management used in the Linux kernel where configuration-related CPP flags should not be changed in code by macro redefinitions with the directive `#define`.

```
1  #if (defined A || defined B || defined C)
2  //block 1
3  # if defined A
4    //block 2
5  # elif defined B
6    //block 3
7  # else
8    //block 4
9  # endif
10 #endif
```

Listing 3.1: Example of Conditional Blocks

3.3.1 Basic Definitions

Definition 1 *[Configuration] Given n boolean configuration flags found in a compilation unit u, a **configuration** is the vector $\vec{f} = f_1, \ldots, f_n$, where f_i is a boolean variable representing the assignment of the i-th configuration flag.*

Configuration flags control the selection of conditional blocks in a compilation unit. Each member of the vector $\vec{f}$ represents the flag assignments as seen by `CPP`. For example, the file shown in Listing 3.1 makes use of three different flags; therefore its configuration vector is defined as $\vec{f} = \{A, B, C\}$. If this file is compiled with only the flag B set, according to our definition, the configuration vector is set as $\vec{f} = \{false, true, false\}$.

Definition 2 *[Block Selection] Given a compilation unit u with m conditional blocks, the **block selection** is the vector $\vec{b_u} = b_1, \ldots, b_m$, where b_i is a boolean variable that represents the presence of the i-th conditional block.*

Each variable in this vector represents a conditional block in the compilation unit u. The block selection represents the selection as done by the `CPP`; that is, the members of the vector $\vec{b_u}$ that represent selected blocks are set to true and members representing skipped blocks are set to false. For example, the block selection vector of the file shown in Listing 3.1 is defined as $\vec{b_u} = \{b_1, b_2, b_3, b_4\}$. If this file is compiled with only flags A and B set, that is, using the configuration vector $\vec{f} = \{true, true, false\}$, the resulting preprocessed file will contain blocks b_1 and b_2. Consequently, the block configuration vector will be $\vec{b_u} = \{true, true, false, false\}$

Using Definition 1 and Definition 2, the process of applying a set of configuration flags $\vec{f}$ to a compilation unit u that contains m conditional blocks $\vec{b_u}$ can be expressed by the following function (which is compilation unit dependent, therefore, the index u) $\mathcal{P}_u$:

$$\mathcal{P}_u(\vec{f}) \mapsto \vec{b_u} \tag{3.1}$$

This function represents the *mapping* from a given *configuration* $\vec{f}$ to a specific *block selection* $\vec{b_u}$ if preprocessing is performed by the `CPP` tool. Since the basic nature of propositional formulas is to work on binary decisions, our approach does not attempt to calculate the function $\mathcal{P}_u$ directly. Instead, a helper function $\mathcal{C}_u$ that *checks* the behavior of `CPP` for a given configuration is built using propositional logic. Our approach allows deducing the variability indirectly with the following checker function:

Definition 3 *[Checker Function] Given a function $\mathcal{P}_u$ that represents the conditional compilation semantics of the* `CPP` *language, a configuration vector $\vec{f}$ and a compilation unit u with the block selection vector $\vec{b_u}$, the* ***checker function*** *$\mathcal{C}_u$ is defined as:*

$$\mathcal{C}_u(\vec{f}, \vec{b_u}) \rightarrow \begin{cases} \text{true} & \iff \quad \mathcal{P}_u(\vec{f}) = \vec{b_u} \\ \text{false} & \iff \quad \mathcal{P}_u(\vec{f}) \neq \vec{b_u} \end{cases} \tag{3.2}$$

In order to extract the *variability model* from the compilation unit u, the function $\mathcal{C}_u$ must be constructed. According to the specification of the `CPP` language there are three preconditions for the inclusion of a conditional block during preprocessing:

1. The expression that controls conditional inclusion must evaluate to true, otherwise the block is completely skipped [Int05].

2. A nested block can be selected if and only if its parent is also selected [Int05].

3. For if-groups (groups that contain the directive `#elif` or `#else`), the blocks are evaluated in declaration order. The first that evaluates to true is selected, all others are skipped. The `#else` conditional is selected when none of the predecessor blocks of the if-group evaluates to true [Int05].

These three preconditions control the *presence condition* of conditional blocks. Together, they are *necessary* and *sufficient*, which means that if they are valid for a specific block, this block will necessarily be selected by the `CPP`. They can be directly translated to the following helper functions:

***expression*(b_i)** Given a block b_i, the function *expression*(b_i) returns the logical expression as specified in the block declaration. Example: For the first block in Listing 3.1, the function *expression*(b_1) returns: $A \vee B \vee C$.

***parent*(b_i)** Given a block b_i, the function *parent*(b_i) returns the logical variable that represents the selection of its parent. If the block is not nested in any other block, then the result is always *true*. Example: For the third block in Listing 3.1, the function returns: b_1.

***noPredecessors*(b_i)** Given a block b_i, the function *noPredecessors*(b_i) returns the negation of the disjunction of all its predecessors (logical variables representing blocks) in an if-group. Example: For the fourth block of Listing 3.1 (*noPredecessors*(b_4)), the function returns: $\neg(b_2 \vee b_3)$.

Note that the helper functions do not return boolean values, they actually return[3] logic expressions containing logic symbols and operators. Using these functions, the *presence condition* for conditional blocks is constructed as follows:

Definition 4 *[Presence Condition] A conditional block b_i is selected by* `CPP` *if and only if the following conjunction holds:*

$$\mathcal{PC}(b_i) = expression(b_i) \wedge noPredecessors(b_i) \wedge parent(b_i) \tag{3.3}$$

The presence condition $\mathcal{PC}(b_i)$ allows us to build a boolean formula that can be used to check if a specific block is to be selected for a configuration $\vec{f}$. By combining all presence conditions of a compilation unit we can build the checker function as follows:

$$\mathcal{C}_u(\vec{f}, \vec{b_u}) = \bigwedge_{i=1..m} b_i \leftrightarrow \mathcal{PC}(b_i) \tag{3.4}$$

The checker function returns a boolean formula containing $n + m$ variables, where n is the size of the vector $\vec{f}$ and m is the size of the vector $\vec{b_u}$. The function is simply the conjunction of the presence condition of all blocks of a specific compilation unit. It is important to note that the vector $\vec{f}$ does not explicitly appear on the right hand side of the function because it will appear implicitly as a result from function $expression(b_i)$ when calculating the presence condition of a block. Also important, is the biimplication between a block and its presence condition. When a block has its presence met, it is not only *allowed* to be enabled, but it will *necessarily* be enabled by the `CPP`, it means that not only the block implies its presence condition, but the presence condition also implies the block, therefore the biimplication.

This function is satisfiable for the variable assignments that correspond to valid behaviors of the `CPP`. The checker function is able to verify if a specific block selection $\vec{b_u}$ represents the resulting preprocessing when given a configuration $\vec{f}$ and a compilation unit u. Consequently, the term *variability* is defined as follows:

Definition 5 *[Variability] Given a compilation unit u and the checker function $\mathcal{C}_u$, the* ***variability*** *$\mathcal{V}$ of a compilation unit u is the set of all block selections $\vec{b_u}$ for which there exists a selection $\vec{f}$ such that $\mathcal{C}_u$ is satisfied:*

$$\mathcal{V} = \{\vec{b_u} \mid \exists \vec{f} : \mathcal{C}_u(\vec{f}, \vec{b_u})\} \tag{3.5}$$

[3]For some function calls, the returned expression is trivial. For example, the call of *parent()* for a top-level block, *expression()* for an else block, or *noPredecessors()* for a block that does not belong to an if-group. In all cases, we can return the constant `true` and (optionally) avoid its inclusion in the resulting presence condition.

3.3.2 Calculating Variability with the *Checker Function*

The *variability* $\mathcal{V}$ (Formula (5)) is the set of vectors $\vec{b_u}$ that can be mapped by the function $\mathcal{P}_u(\vec{f})$ (Formula (1)) with at least one *input configuration* $\vec{f}$. This set can be calculated by generating the checker function $\mathcal{C}_u$ (Formula (4)), and solving the satisfiability problem that follows from Formula (2). The complexity to generate the checker function scales linearly $O(n)$ with the number n of conditional blocks. *Solving* the resulting formulas of course remains NP-complete.

The result of this method is a propositional formula that represents the implementation variability. It serves as a building block for further reasoning techniques. The formula that we generate is not a visual or editable model. It is a logical representation of the variability as described by the `CPP` annotations. It is similar to other approaches [BRCT05, MWCC08, CHH09, CP06] that translate visual models, e.g., feature models, into boolean formulas.

A Concrete Example

In order to explain how the construction of the *checker function* works in practice, we show the results for a non-trivial example. Additional examples are discussed in detail elsewhere [STL10]. In order to facilitate the comprehension we introduce the following annotations:

- The symbol ↻ at the end of each line specifies in which iteration the presence condition of the corresponding conditional block is generated.
- Each clause has been marked with an overbrace ($\overbrace{\ \ }^{\blacktriangleright f()}$) in order to indicate which helper function provided each part of the resulting presence condition.

The example shown in Listing 3.2 uses all `CPP` directives that are relevant for conditional compilation. The complexity of the example is typical for real world code. From this source code, we construct the *checker function* as shown in Figure 3.3.

The first iteration generates the presence condition for the top level block b_1 (line 1). As this block has no predecessors or a parent, the resulting presence condition is the biimplication between the block and the formula returned by the helper function *expression(b_1)*.

In the second iteration the presence condition for the block b_2 (line 2) is generated. The presence condition for this block depends on its expression, provided by *expression(b_2)* and on its parent, provided by *parent(b_2)*. The resulting presence condition is the conjunction of these formulas biimplied with the block itself.

For block b_3 (line 3) the three helper functions contribute to its presence condition. Because this block is a successor of block b_2 and must not be considered in case

$$\mathcal{C}_u(\{A,B,C\},\{b_1,\ldots,b_7\}) = \bigwedge_{i=1..7} b_i \leftrightarrow \mathcal{PC}(b_i) =$$

$$\bigwedge_{i=1..7} b_i \leftrightarrow expression(b_i) \wedge noPredecessors(b_i) \wedge parent(b_i) =$$

$$\Big(b_1 \leftrightarrow (\ \overbrace{A \vee B \vee C}^{\blacktriangleright expression(b_1)}\ )\Big) \qquad (\circlearrowleft\ 1)$$

$$\wedge \quad \Big(b_2 \leftrightarrow (\ \overbrace{(A \wedge B)}^{\blacktriangleright expression(b_2)} \ \wedge\ \overbrace{(b_1)}^{\blacktriangleright parent(b_2)}\ )\Big) \qquad (\circlearrowleft\ 2)$$

$$\wedge \quad \Big(b_3 \leftrightarrow (\ \overbrace{(A \wedge C)}^{\blacktriangleright expression(b_3)} \ \wedge\ \overbrace{\neg(b_2)}^{\blacktriangleright noPredecessors(b_3)} \ \wedge\ \overbrace{(b_1)}^{\blacktriangleright parent(b_3)}\ )\Big) \qquad (\circlearrowleft\ 3)$$

$$\wedge \quad \Big(b_4 \leftrightarrow (\ \overbrace{(B\ \wedge\ C)}^{\blacktriangleright expression(b_4)} \ \wedge\ \overbrace{\neg(b_2 \vee b_3)}^{\blacktriangleright noPredecessors(b_4)} \ \wedge\ \overbrace{(b_1)}^{\blacktriangleright parent(b_4)}\ )\Big) \qquad (\circlearrowleft\ 4)$$

$$\wedge \quad \Big(b_5 \leftrightarrow (\ \overbrace{(C)}^{\blacktriangleright expression(b_5)} \ \wedge\ \overbrace{(b_4)}^{\blacktriangleright parent(b_5)}\ )\Big) \qquad (\circlearrowleft\ 5)$$

$$\wedge \quad \Big(b_6 \leftrightarrow (\ \overbrace{\neg(b_5)}^{\blacktriangleright noPredecessors(b_6)} \ \wedge\ \overbrace{(b_4)}^{\blacktriangleright parent(b_6)}\ )\Big) \qquad (\circlearrowleft\ 6)$$

$$\wedge \quad \Big(b_7 \leftrightarrow (\ \overbrace{\neg(b_2 \vee b_3 \vee b_4)}^{\blacktriangleright noPredecessors(b_7)} \ \wedge\ \overbrace{(b_1)}^{\blacktriangleright parent(b_7)}\ )\Big) \qquad (\circlearrowleft\ 7)$$

Figure 3.3: Checker function for Listing 3.2.

```
1 #if defined A || defined  B || defined C
2 # if defined A && defined B
3 # elif defined A && defined C
4 # elif defined B && defined C
5 #   ifdef C
6 #   else
7 #   endif
8 # else
9 # endif
10 #endif
```

Listing 3.2: Source Code Example with all language features for Conditional Compilation.

the block b_2 is selected, the formula returned by *noPredecessors(b_3)* ensures that b_3 can only be selected if b_2 is not.

Blocks b_5 (line 5) and b_6 (line 6) form a group of blocks nested inside block b_4 (lines 4-8). Therefore, iteration 5 and 6 generate presence conditions using formulas provided by ►*parent()*, ►*expression()* and ►*noPredecessors()* for both blocks.

Block b_7 (line 6) is treated in the last iteration. Since this is an `#else` block without an expression of its own, only the helper functions ►*parent()*, and ►*noPredecessors()* are generated.

The following solutions satisfy this *checker function*:

$$\Big\{\{\emptyset,\emptyset\},\{C,b_1,b_7\},\{B,b_1,b_7\},\\ \{B,C,b_1,b_4,b_5\},\{A,b_1,b_7\},\{A,C,b_1,b_3\},\\ \{A,B,b_1,b_2\},\{A,B,C,b_1,b_2\}\Big\}$$

Note that the tuples are in the form $(\vec{f},\vec{b})$, but for a representation that is easy to read we show only the vector members that are set to *true*, therefore, the empty set $\emptyset$ represents a vector where all members are set to *false*. From these solutions, we identify the following set of block combinations:

$$\mathcal{V} = \Big\{\{b_1,b_7\},\{b_1,b_4,b_5\},\{b_1,b_7\},\{b_1,b_3,\}\{b_1,b_2\}\Big\} \tag{3.6}$$

The set $\mathcal{V}$ corresponds to all possible configurations (different block selections) that the `CPP` can generate for the file shown in Listing 3.2. This is possible to be calculated due to the characteristics of the boolean formula shown in Figure 3.3, which is built taking into consideration the structure of the input file and the semantics of the `CPP` language. As a result, it is a compact boolean formula that mimics the `CPP` for a specific source file.

3.4 Reasoning on Source Code Variability Models with the *Checker* Function

The formula defined in Equation 3.4 represents the *implementation variability* model of a compilation unit. Based on this model, similar reasoning techniques already proposed for feature models [BRCT05, TBK09] can be applied. These operations are built upon typical functions provided by SAT solvers: (1) *sat_count()* calculates the number of solutions for a given boolean formula, (2) *all_sat()* returns all valid variable assignments that satisfy a given boolean formula, (3) *SAT()* checks if a given boolean formula is satisfiable. Using these functions we discuss the following reasoning operations:

Number of variants: using the formula $\mathcal{C}_u$ built by our approach, we are able to calculate the number of all possible different configurations a compilation unit can be translated to as *sat_count*($\mathcal{C}_u$).

Calculating all variants: using $\mathcal{C}_u$ we can also calculate all valid variable assignments as *all_sat*($\mathcal{C}_u$). Using this result, we can classify which assignments lead to the same configuration, and, as a consequence, the unique set of valid block configurations.

Validation: using $\mathcal{C}_u$, we can also check for internal consistency. That is, for each block of $\mathcal{C}_u$ we can evaluate whether it is selectable by at least one valid configuration. For each block this can be executed as *SAT*($\mathcal{C}_u \wedge b_i$). External consistency (e.g., implementation model versus feature model) can also be examined. In this case each block has to be selectable in at least one valid configuration of the feature model. Using *FM* as the boolean formula representing a feature model, we have to calculate *SAT*($\mathcal{C}_u \wedge b_i \wedge FM$) for each block.

Reasoning about edits: the algorithms for reasoning about edits to feature models presented by THUM [TBK09] can also be applied to the formula $\mathcal{C}_u$. These algorithms are used to detect to which extent a refactoring on a feature model (in our case on the `CPP` directives) impacts the model variability.

Filtering and partial configurations: reducing the number of variation points is an effective means to facilitate code comprehension. By pre-configuring a subset of the available configuration flags (assignments to some variables of $\vec{f}$) in order to constrain $\mathcal{C}_u$, both simplified `#ifdef` expressions of conditional blocks as well as unselectable blocks can be queried and used to provide such a *partial* view

On the basis of the reasoning operations described above together with the resulting variability model denoted as $\mathcal{C}_u$ we are able to answer the questions we posed in Section 3.2.1

3.5 Related Work

Analyzing large scale software projects with transformation systems are related to our approach; DMS [Bax02] proposed by BAXTER is probably the most renowned one. In the context of this framework, an approach has been published [BM01] that aims at simplifying CPP statements by detecting dead code. Unlike our approach, this work uses concrete configurations to evaluate the expressions partially [JGS93]. In contrast to that, our work does not require concrete values for CPP identifiers, but produces a logical model in the form of a propositional formula that can either be evaluated directly or can be combined with further constraints like the ones extracted from the feature model. We believe that our approach could be used to further extend the DMS framework.

Analyzing conditional compilation with symbolic execution has been proposed by HU et al. [HMDL00]. This approach maps conditional compilation to execution steps: Inclusion of headers map to *calls*, alternative blocks to branches in a control flow graph (CFG), which is then processed with traditional symbolic execution techniques. LATTENDRESSE [Lat04] improves this technique by using *rewrite systems* in order to find presence conditions for every line of code. Similarly to our approach the presence conditions of all conditional blocks are calculated during the process as well. However, symbolic evaluation does not actually calculate the variability as per Definition 5, which is required for the reasoning techniques that we presented in Section 3.4. Although it would be possible to use the presence conditions obtained by symbolic execution and to construct the checker function (Formula 3.4), we present a lightweight approach that does scale, as the algorithm for generating the boolean formula grows linearly with respect to the number of blocks.

Other approaches extend the grammar of the C/C++ parser with support for CPP directives. BADROS et al. [BN00] propose the PCp^3 framework to integrate *hooks* into the parser in order to perform many common static code analyses. However, they focus on the mapping between unprocessed and preprocessed code, whereas our work aims at mapping towards higher level models. GARRIDO [Gar05] extends the C/C++ parser with the concept of conditional abstract syntax trees, which enables preprocessor-aware refactorings on CPP-based source files. We believe that this work can be combined with our approach to further support software engineering tools.

The evaluation of variability from models is also related to our work. CZARNECKI et al. [CW07] present an approach to transform logic formulas into *feature models*. This technique could be combined with ours in order to automatically generate visual models from the boolean formula that is calculated by our algorithm. BENAVIDES et al. [BRCT05] present several techniques for *reasoning* on feature mod-

els after transforming them into boolean formulas. We have shown how the same kind of reasoning can be applied to the resulting implementation variability model of our approach. SHE et al. [SLB+10] present an approach to convert the Linux Kconfig model into feature models. On the one hand, complementary to our work, it could be used in combination with our algorithm to provide the crosscheckings. On the other hand, contrary to our work, the source code files—consequently the CPP information—are not taken into consideration.

3.6 Summary

In this chapter we presented a technique to transform source code annotations in the form of CPP directives into a propositional formula that represents the full variability of a source file. This is the first building block for the construction of a reasoning framework that is able to reveal variability bugs. Using solely the formula describing the source code variability we are now able to perform reasonings that might reveal internal inconsistencies. Moreover, we are also able to combine this formula with the formulas from other variability sources to reveal inconsistencies across different artifacts. In the next chapters we will introduce the concepts and techniques to combine the propositional formulas from different variability models, and also, how to perform reasonings targeted to the detection of variability bugs.

"The kernel is huge and bloated... and whenever we add a new feature, it only gets worse ."

Linus Torvalds, LinuxCon'09

4 Slicing Variability Models

In this chapter we introduce the concept of *slicing variability models*. A diversity of reasoning operations can be performed in variability models represented as boolean formulas. In real-world large-scale software product lines these logic formulas may contain up to several thousands of variables. Due to the sheer size of these formulas the use of reasoning techniques in such large projects may be impracticable. By slicing models we are able to generate sub-models from an original model containing only the relevant parts for a specific reasoning operation. By doing so we can significantly reduce the model size. We present techniques to build variability models as sets of presence conditions that can be used with our algorithm for slicing models. *This chapter shares material with the EuroSys'11 paper 'Feature Consistency in Compile-Time–Configurable System Software'[TLSSP11].*

4.1 The Role of Configuration Variability Models

The scope of a software product line is defined by the set of features and their relations, which form the set of all possible products. Normally, this information is captured during domain analysis and documented and formalized in form of *variability models*. There are several kinds of variability models: feature models, decision models, and also, project-tailored models like KBUILD for the Linux kernel or the Component Definition Language (CDL) of the eCos operating system, among others.

These models describe the user-visible variability completely. That is, the variability accessible by the user (responsible for configuring a product) during the configuration process. For this reason, tool support for the variability modeling approaches cited above offer graphical user interfaces for the derivation of variants. Such tools normally guide the user through the selection of features impeding the derivation of incorrect configurations, that is, the output of these tools are specifications (variant/product configurations) that are conform to the variability model. Although these models can be used in different phases (domain engineering or application engineering) of the software lifefcycle, after the implementation phase, they will always be used for the configuration of specific variants. For this reason we categorize these models as *configuration variability models*.

However, as we introduced in Section 2.1.2, variability management is not only part of the configuration stage, it is pervasive to all phases of software product line engineering. For this reason, as decisions about variation points are taken, they have to be documented or implemented in the corresponding artifacts. For example, when a new source file is added to the project, the build system has to be extended to include not only the command required to build and link this file, but also the conditions (possibly the presence of a feature or a set of them) under which this file has to be compiled. Another example is the extension of a source file to include the functionality of a new feature. The new code snippet can be annotated with preprocessor directives so that it will be included only when the feature it implements is selected. The set of annotations in a source file implicitly, as explained in detail in Chapter 3, forms its variability model. In both examples, the described efforts that set conditions for the inclusion of variation points, are, in our opinion, part of the variability management process. Moreover, the rules that control this kind of variation points are classified as *implicit variability models*.

For our purposes it does not matter if a model is explicit or implicit. Our goal is to check for inconsistencies among several variability models. Typically, reasoning operations (both in single and in multiple) variability models are performed after such models have been converted to some common abstraction format. Examples are,

propositional formulas [Bat05], constraint satisfaction problems [BRCT05], binary decision diagrams [MWCC08], higher-order logic [JK07], among others. As a result, the original format of variability models is not as important as the final format in which the reasoning operation will be performed. However, the key issue is to handle the size of these models. Specifically in our case, where we aim at combining all models of a project to perform targeted reasoning operations, and, consequently, as each model may contain several thousands of variation points and their constraints, to handle the size of the combined model becomes the key challenge.

4.2 Scalability Issues of Variability Models

Explicit Variability Models Many authors have studied the scalability issues of configuration variability models. BENAVIDES [BRCT05] is one of the first authors to study in detail automated reasoning in feature models. His tool, the FAMA (FeAture Model Analyzer) [BSTRC07], is a framework for automated reasoning. It integrates several logic representations and off-the-shelf solvers, namely, BDDs, CSPs and SAT solvers. FAMA combines all these representations in order to achieve the best performance for the different reasoning operations it offers. The framework is able to automatically select the most efficient solver according to the reasoning operation chosen by the user. For example, if the user requests the reasoning *list all possible products*, the BDD solver will be selected as this is the most appropriate solver for this particular reasoning. On the other hand, if the user selects the operation *valid model*, which checks for the validity of the feature model, the SAT solver will be the better choice. Although the authors argue that the different solvers have their advantages and disadvantages with respect to the different reasoning operations, there is no silver bullet. That is, even though BDDs is the best solver for counting the number of valid products in a feature model, the size of BDDs can grow exponentially depending on the variable ordering, and finding the optimal ordering is a NP-complete problem. As a result, even the most appropriate solver may fail if the input model is too large.

MENDONÇA [Men09] has also studied the reasoning in feature models with both BDDs and SAT solvers. In his PhD thesis he explores the structural properties of feature models, like impact of mandatory features, child-parent relations, etc., in order to improve BDD minimization. Based on this, he introduces two heuristics to improve ordering of BDD variables. The evaluation of these heuristics has shown that their application leads to a considerable improvement compared to ordering heuristics originating from other domains, as for example circuit analysis. When applied to both real and automatically generated feature models, the new heuristics were able to reduce BDD sizes up to 10 times. As a result, he was able to build

feature models, as BDDs, containing up to 2000 features. He also introduces the concept of FTRS (Feature Tree Reasoning System) in order to improve feature-model reasoning with SAT solvers. The basic idea is to use a tree structure (conforming to a given meta-model) instead of converting the feature model to a propositional formula. The FTRS system is able to check for satisfiability and count the number of solutions of a feature tree in constant and linear time (w.r.t. to the tree size). Based on this, the reasoning *number of solutions* worked on models with up to 150 features, in contrast to BDDs, that for this type of reasoning operation could only handle models containing up to 30 features. Additionally, MENDONÇA also empirically shows that for reasonings that depend only on satisfiability checks, standard SAT solvers can handle feature models with up to 10000 features.

JANOTA [Jan10] studies feature-model–based configuration processes. According to him, a configuration process is an interactive step-by-step process where the user gradually contrains the problem space. To support this process, JANOTA builds a *configurator* that gets as input the feature model and the decisions (feature selections) as propositional formulas and guides the user by discouraging decisions inconsistent with the model and suggesting decisions that are necessary to make the selection valid. He also optimizes the configuration process by reducing model size. To this end, he proposes an approach to eliminate dispensable variables by enumerating minimal models. Moreover, his evaluation showed that the proposed variable elimination algorithm worked (response time below 1 second) for most of the test data, but for one specific complex feature model the algorithm aborted in 22% of the tests. However, in the cases where the computation succeeded, around 80% of the unbounded variables were dispensable.

The approaches presented above have tackled the scalability problems of reasoning on feature modeling in general. Other techniques have approached similar challenges in the realm of other configuration variability models. The most prominent example is the variability model of the Linux kernel which, as explained in Section 2.2.2, is implemented with the tool KCONFIG. In this context, some authors [LSB+10, SLB+10, BSL+10b, DTSPLb] have studied the characteristics of Linux' actual configuration model. That is, the analysis of the variability model described in the source files of the Linux kernel in order to reveal the number of features, their types, types of relations, and also, how these (and many others) metrics have evolved during the different releases. Others [ZK10, TSD+12], focused on the semantics of the KCONFIG language in order to convert it to propositional formulas (in the same fashion as done with feature models). In a comprehensive study, ZENGLER et al. [ZK10] present a formalization of the KCONFIG constructs in form of propositional logic. The introduced algorithms were implemented and analyzed in the Linux kernel version 2.6.33. For the architecture `x86` the translation results in 13,403

propositional symbols distributed in $250,430$ logic clauses. TARTLER et al. [TSD+12] present a similar set of algorithms to translate KCONFIG artifacts into logical formulas. For the `x86` architecture on the Linux kernel version 3.0 their translation resulted in a formula with $12{,}217$ propositional symbols and $202,809$ clauses.

As we have seen above, several approaches reach the limits of logic solvers even for operations involving formulas that contain only a few thousands of clauses [Men09, Jan10]. We have also seen that approaches aimed at extracting boolean formulas from real-world models, for example the configuration model of the Linux kernel KCONFIG, yield formulas containing hundreds of thousands of clauses [ZK10, TSD+12]. One of the goals of this thesis is to use such very large models in reasoning operations backed by logic solvers. Clearly, it is a challenge to avoid the intractability of reasoning operations that are built with formulas of this dimension.

Implicit Variability Models The approaches explained above were developed and assessed using feature models and the KCONFIG toolchain, which are explicit variability models. However, researchers have also studied similar issues, like translation to propositional logic, reasoning scalability, etc., on implicit variability models. For example, the extraction and translation to propositional logic of the variability description from the build system (KBUILD) of the Linux kernel has been investigated [BSL+10a, NH11, NH12, DTSPLa]. All these surveys were developed independently, and employ different strategies to extract the formulas. Nevertheless, they delivered comparable results. The approach from DIETRICH et al. [DTSPLa] is the most recent, and has improved on the aspect robustness as it works in all versions of Linux, in contrast to the other approaches [BSL+10a, NH11, NH12] that work only on specific versions. Their tools are able to automatically extract presence conditions for more than 93% of all source files of the Linux kernel. For the architecture `x86` on the Linux kernel version 3.2, they extracted $11{,}147$ boolean formulas, where their conjunction forms the variability model of the KBUILD system.

In previous work [STLSP10] we have also presented a technique to extract constraints from implicit variability models. We showed, using a technique that was further extended in Chapter 3, how to extract variability models from source files annotated with CPP directives. In that work our tools extracted constraints for $25{,}844$ CPP blocks for the Linux version 2.6.33 code base.

The size of the formulas extracted from implicit variability models also impose challenges to the techniques that aim at using them in reasoning operations.

Problem Statement Reasoning with propositional logic and solvers like SAT and BDDs brings several benefits like established theories, availability of mature tooling, etc. However, as we discussed above, all approaches presented have reached

certain limits where the solvers could not provide an answer in an acceptable time frame. For this reason the biggest concern when generating formulas to implement a desired reasoning operation is model size. This is particularly important in the context of this thesis because we aim at combining several models in order to reveal inconsistencies. However, as we have seen above, even in single models, reasoning operations may fail due to model size.

As an example, consider a hypothetical reasoning operation to check for the inconsistencies in source files of the Linux kernel. In order to build a boolean formula that describes this problem one would have to consider the constraints from KCONFIG, KBUILD and the source file in question. It can be described as $\Phi = \theta \wedge \psi \wedge \mathrm{PC}(B_i)$. Where θ represents the constraints from the KBUILD model (which as shown above may contain more than $200,000$ clauses), ψ is the model extracted from the KBUILD system (which as shown above may contain more than $11,000$ boolean formulas), and $\mathrm{PC}(B_i)$ is the condition under which the CPP block of interest will be compiled. It is easy to see that after the combination of several models from real-world projects, the final reasoning formula will contain an overwhelming number of propositional symbols and clauses, which will inevitably lead to intractability or unacceptable response times from solvers.

To address this problem we propose in the following an algorithm to extract a subset of the constraints from a specific model that are of interest for a specific reasoning operation. Our goal is to use this algorithm to rewrite the formula $\Phi \stackrel{\text{def}}{=} \theta \wedge \psi \wedge \mathrm{PC}(B_i)$ to $\Phi' \stackrel{\text{def}}{=} \theta' \wedge \psi' \wedge \mathrm{PC}(B_i)$ where θ' is a formula containing a subset of the constraints in θ and ψ' is a formula containing a subset of the constraints in ψ, and $\mathrm{SAT}(\Psi) \equiv \mathrm{SAT}(\Psi')$, that is, the equisatisfiability for this specific reasoning (validity of the CPP block $\mathrm{PC}(B_i)$) is maintained.

In the next section we introduce a technique to prepare the model constraints to be used in our algorithm, which we call *model slicing* and, subsequently, we describe the algorithm and illustrate it with examples.

4.3 Slicing Models

One of our main goals in the identification of variability bugs is to check if a given combination of variation points—even from different models—are valid together. Clearly, this information from different models must be converted to a common representation so that the desired reasoning operation can be applied. To illustrate, we want to know if a set of features from the feature model, a set of source files (controlled by the build system) and a set of conditional blocks from these source files form a valid combination of variation points. This validity is determined by the constraints of the models being combined. Such a combination will be valid if there is

a valid interpretation that satisfies all models involved. Normally, all these models, and the corresponding formulas, are build on the same set of symbols, namely, the features described in the configuration variability model. Therefore, the typical reasoning operations are simply a conjunction of a set of constraints from different models over the same set of propositional symbols plus the constraint (or constraints) that define the reasoning operation of interest. Intuitively, this kind of reasoning checks for all variation points being combined if there are no contradictions among them. The constraints that regulate the inclusion of individual variation points are called *presence conditions*.

However, the presence conditions of variation points are mixed together in the formula that describes the whole model they belong to. Recall the conversion from feature models to propositional formulas we introduced in Section 2.1.4, in which the constraints of the different features are all together in one formula, and if only the constraints of a few features are of interest, the whole formula has to be used. This is sub-optimal, as formula size is one of the key issues that lead to problem intractability and poor performance of solvers.

To address these issues this chapter introduces the concept of model slicing that allows specific parts of a model to be used for specific reasonings. Thereby, we are able to reduce model size, and, consequently, to improve the overall reasoning process for the detection of variability bugs.

The first step towards the slicing algorithm is the preparation of models that will be used as input. Such models have to conform to a given structure. In the following section we specify the concept of presence conditions, which is a format for variability models that enables the use of the slicing algorithm.

4.3.1 Presence Conditions

Presence conditions define the constraints that a variation point has to meet to be included in a valid variant. For example, in a feature model a feature can only be included in a valid variant if its parent is also included. Naturally, each of the different feature types has different constraints.

We define a presence condition as follows:

$$v \rightarrow \mathrm{PC}(v) \tag{4.1}$$

A variation point v implies its presence condition $\mathrm{PC}(v)$. Note that v must be a single propositional symbol (e.g. v_1, v_2, v_3, etc.) and $\mathrm{PC}(v)$ can be any propositional formula (e.g. v_1, $v_1 \wedge v_2$, $v_3 \vee \neg v_4$, etc.). For the inclusion of v into a valid variant, the constraint described as $\mathrm{PC}(v)$ must evaluate to true. Recall that this formula is equivalent to $(\neg v \vee \mathrm{PC}(v))$, it is easy to see that if the variation point is set to false

($v \mapsto \bot$), the formula evaluates to true, but if v is true, the formula will evaluate to true only if $\mathrm{PC}(v)$ is also true ($\mathrm{PC}(v) \mapsto \top$).

Variation points are the building blocks of variability models. We regard variability models as sets of variation points and their interdependencies. So, we can use our definition from Equation 4.1 to build formulas that describe a whole variability model. One can achieve that by building a propositional formula that conjuncts the presence conditions of all variation points that belong to the variability model:

$$v_1 \rightarrow \mathrm{PC}(v_1) \wedge v_2 \rightarrow \mathrm{PC}(v_2) \wedge \ldots \wedge v_n \rightarrow \mathrm{PC}(v_n) \tag{4.2}$$

The typical transformation rules to convert variability models (feature models, KCONFIG, KBUILD, etc.) into propositional formulas do not always yield formulas in this format. However, using propositional calculus we can transform such formulas into the format shown in Equation 4.2, which is a prerequisite for the application of the slicing algorithm since it expects as input variability models with this structure.

Feature Model Presence Condition

In this section we recall the mappings from feature trees to propositional formulas that we introduced in Section 2.1.4. Such mappings are broadly accepted and have been used by several researchers. Here we will show how to equivalently transform them into the presence condition format that we presented in the previous section.

Optional Features Given an optional feature F and its parent P the boolean formula is simply $F \rightarrow P$. Note that we have an implication of two propositional symbols. It is already in the format of Equation 4.1, where $v = F$ and $\mathrm{PC}(v) = P$, and nothing has to be done.

Mandatory Features Given a mandatory feature M and its parent P, we get the formula $M \leftrightarrow P$. This is the typical formula that is used in the literature to describe mandatory features. It is not compatible with our definition of presence condition because it uses a bi-implication. Therefore, we have to transform it to the format of Equation 4.1 where a propositional symbol implies a formula. Recall that $(M \leftrightarrow P) \equiv (M \rightarrow P) \wedge (P \rightarrow M)$. So, we split the bi-implication into two implications and add one of the implicants to the presence condition of M and the other to the presence condition of P. As the model is the conjunction of all presence conditions, both implicants will be conjugated in the final model. As a result, the presence condition for mandatory features is the implication to its parent. At first, it may seem inaccurate for the reader familiar with the typical bi-implication construct, since mandatory and optional features have the same kind of presence condition (an implication to the parent). However, in the case of mandatory features, the parent also implies the child.

Alternative Feature Groups Given n features in an alternative group exactly one feature must be selected if the parent is selected. So, given three features f_1, f_2 and f_3 and the parent P, the typical formula describing this group is

$$P \leftrightarrow (f_1 \wedge \neg f_2 \wedge \neg f_3) \vee (\neg f_1 \wedge f_2 \wedge \neg f_3) \vee (\neg f_1 \wedge \neg f_2 \wedge f_3)$$

Again, to transform this formula to our desired format, we have just to break the bi-implication so that the parent implies the disjunction of its children, and each child implies the parent and the negation of all of its siblings:

$$\begin{aligned}
&(P \rightarrow f_1 \vee f_2 \vee f_3) && \wedge \\
&(f_1 \rightarrow P \wedge \neg f_2 \wedge \neg f_3) && \wedge \\
&(f_2 \rightarrow P \wedge \neg f_1 \wedge \neg f_3) && \wedge \\
&(f_3 \rightarrow P \wedge \neg f_1 \wedge \neg f_2)
\end{aligned}$$

This is very similar to the transformation we performed on mandatory features. The bi-implication is split and the different implicants are added to the corresponding presence conditions of the features involved so that when the presence conditions are all conjuncted, we have a formula equivalent to the original one with the bi-implication.

Or-Feature Groups Given n features in an or-group at least one must be selected if the parent is selected. For a group with three features f_1, f_2 and f_3 and the parent P we get the formula:

$$P \leftrightarrow (f_1 \vee f_2 \vee f_3)$$

Splitting the bi-implication we get:

$$\begin{aligned}
&(P \rightarrow f_1 \vee f_2 \vee f_3) && \wedge \\
&(f_1 \rightarrow P) && \wedge \\
&(f_2 \rightarrow P) && \wedge \\
&(f_3 \rightarrow P)
\end{aligned}$$

Again, we have a formula as a conjunction of presence conditions that are in the format of Equation 4.1.

Cross-tree Constrains Also common in feature models are the constraints among features that do not have a direct dependency. That is, one feature may **require** or **exclude** any other feature of the model. Such constraints are normally described as $F_1 \rightarrow F_2$ (for F_1 requires F_2) and $F_1 \rightarrow \neg F_2$ (for F_1 excludes F_2). Such constraints are already in our desired format and no transformation is required.

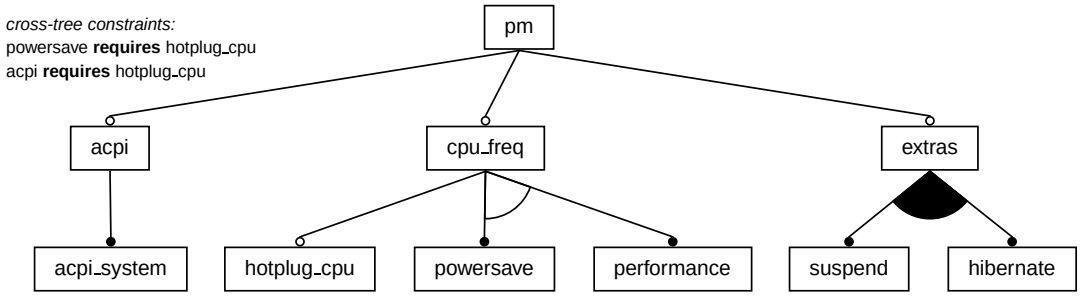


Figure 4.1: Power management features of the Linux kernel. This feature model was extracted and adapted from KCONFIG. A similar version of this model has also been used by SHE et al. [SLB+11]

Example – Feature Models and Presence Conditions

To illustrate how we handle propositional formulas from feature models using presence conditions we introduce an example. In Figure 4.1 we show a feature model with 10 features (including all 4 types) and 2 cross-tree constraints. Using the typical transformation from feature diagrams to propositional formulas we get the following formula for this feature model:

$$
\begin{aligned}
\mathcal{M} = {} & (\texttt{acpi} \rightarrow \texttt{pm} \wedge \texttt{hotplug\_cpu}) \wedge \\
& (\texttt{acpi\_system} \leftrightarrow \texttt{acpi}) \wedge \\
& (\texttt{cpu\_system} \rightarrow \texttt{pm}) \wedge \\
& (\texttt{hotplug\_cpu} \rightarrow \texttt{cpu\_freq}) \wedge \\
& (\texttt{cpu\_freq} \rightarrow \texttt{pm}) \wedge \\
& (\texttt{powersave} \vee \texttt{performance} \leftrightarrow \texttt{cpu\_freq}) \wedge \\
& \neg(\texttt{powersave} \wedge \texttt{performance}) \wedge \\
& (\texttt{extras} \rightarrow \texttt{pm} \wedge (\texttt{suspend} \vee \texttt{hibernate})
\end{aligned}
$$

The propositional formula defined as $\mathcal{M}$ represents the semantics of the feature model shown in Figure 4.1. Several kinds of reasoning operations can be performed with this formula. For example, in a configurator tool, where the user selects the features of interest, the tool checks if the selection is valid. The first user action could be the selection of the feature `acpi`. To check if this partial selection is valid the tool has to perform the reasoning operation $\mathrm{SAT}(\mathcal{M} \wedge \texttt{acpi})$. In this case, as the selection conforms to the model, the solver would respond that the query is satisfiable. The issue with the model $\mathcal{M}$ is that the constraints of different features are mixed

$\mathrm{PC}(v)$	Formula
$\mathrm{PC}(\texttt{pm})$	$\top$
$\mathrm{PC}(\texttt{acpi})$	$\texttt{pm} \wedge \texttt{hotplug\_cpu}$
$\mathrm{PC}(\texttt{acpi\_system})$	$\texttt{acpi}$
$\mathrm{PC}(\texttt{cpu\_freq})$	$\texttt{pm} \wedge (\texttt{powersave} \vee \texttt{performance})$
$\mathrm{PC}(\texttt{hotplug\_cpu})$	$\texttt{cpu\_freq}$
$\mathrm{PC}(\texttt{powersave})$	$\texttt{cpu\_freq} \wedge \neg\texttt{performance}$
$\mathrm{PC}(\texttt{performance})$	$\texttt{cpu\_freq} \wedge \neg\texttt{powersave} \wedge \texttt{hotplug\_cpu}$
$\mathrm{PC}(\texttt{extras})$	$\texttt{pm} \wedge (\texttt{suspend} \vee \texttt{hibernate})$
$\mathrm{PC}(\texttt{suspend})$	$\texttt{extras}$
$\mathrm{PC}(\texttt{hibernate})$	$\texttt{extras}$

Table 4.1: List of presence conditions for all features of the model shown on Figure 4.1.

together. As a result, regardless of the reasoning operation being perfomed, it has always to be used as a whole.

Using our concept of presence conditions we define the constraints of each variation point (in this case the features of the model) separately so that they can be combined (when necessary) to form a model that is equivalent to $\mathcal{M}$. The presence conditions for the features of the feature model shown in Figure 4.1 are described in Table 4.1. These presence conditions were derived using the rules we introduced in the previous section. Recall that using this type of presence conditions we can build a model equivalent to $\mathcal{M}$ by combining the presence conditions: $\bigwedge_{v \in V} (v \rightarrow \mathrm{PC}(v))$ (see Equation 4.2), where V is the set of all features of the model. Combining this formula with the presence conditions of Table 4.1 we get:

$$
\begin{aligned}
\mathcal{M}' = {} & (\texttt{pm} \rightarrow \top) \wedge \\
& (\texttt{acpi} \rightarrow \texttt{pm} \wedge \texttt{hotplug\_cpu}) \wedge \\
& (\texttt{acpi\_system} \rightarrow \texttt{acpi}) \wedge \\
& (\texttt{cpu\_freq} \rightarrow \texttt{pm} \wedge (\texttt{powersave} \vee \texttt{performance}) \wedge \\
& (\texttt{hotplug\_cpu} \rightarrow \texttt{cpu\_freq}) \wedge \\
& (\texttt{powersave} \rightarrow \texttt{cpu\_freq} \wedge \neg\texttt{performance}) \wedge \\
& (\texttt{performance} \rightarrow \texttt{cpu\_freq} \wedge \neg\texttt{powersave} \wedge \texttt{hotplug\_cpu}) \wedge \\
& (\texttt{extras} \rightarrow \texttt{pm} \wedge (\texttt{suspend} \vee \texttt{hibernate}) \wedge \\
& (\texttt{suspend} \rightarrow \texttt{extras}) \wedge \\
& (\texttt{hibernate} \rightarrow \texttt{extras})
\end{aligned}
$$

The formulas $\mathcal{M}$ and $\mathcal{M}'$ are equivalent. By employing propositional calculus (as shown in Section 4.3.1) we can derive one from the other. As a result, the reasoning operation $\mathrm{SAT}(\mathcal{M}' \wedge \texttt{acpi})$ has the same result as discussed above. For any reasoning operation the use of $\mathcal{M}$ or $\mathcal{M}'$ makes no difference. At this point, we are not able to see the advantages of building presence conditions individually for every variation point. However, building variability models from these presence conditions is just the initial building block of our approach that aims at generating smaller sub-models that are equivalent to the complete model for specific reasoning operations.

Using our definition of presence condition for individual presence conditions allows us to handle variability models as a *database* of constraints. This database can then be queried to yield the constraints of single variation points. One example is given in Table 4.1, the query operation gets as input the variation points, the left column in the table, and the presence condition is returned, the right column in the table.

4.3.2 Discussion

In the previous sections we introduced the initial concepts that are required for the understanding of our model slicing approach. We showed how to use simple propositional calculus transformations to generate propositional formulas that match our definition of presence conditions. This specific format of presence conditions is particularly interesting because it—together with the slicing algorithm—enables the use of only a subset of constraints instead of always using a formula representing the full model.

As a consequence, besides having the variability models of a project converted to a common representation (e.g. propositional logic), it is also beneficial to use the same type of presence conditions for all models. In doing so, one can take advantage of the slicing algorithm when performing reasonings among many models. As a result, reasoning operations will no more be as big as the sum of the size of all models involved. They will rather have a size matching their complexity.

Fortunately, the transformations in the formulas to organize feature models as sets of presence conditions can be done with other variability models as well. That is, models derived from KCONFIG, KBUILD or CPP-based source files, can also be transformed into sets of presence conditions [TLSSP11].

As we explained in Section 4.2 many reasoning operations are unfeasible because of model size. Therefore, we aim at reducing model size to improve scalability and performance of reasoning operations. In the following we introduce the algorithm that can derive the dependencies of a given initial set of variation points.

Require: variability model: $\mathcal{M}$
Require: initial set of variation points: $\mathcal{S}$, where $\mathcal{S} \subset \mathcal{M}$
1: $\mathcal{R} = \mathcal{S}$
2: **while** $\mathcal{S} \neq \emptyset$ **do**
3: $\quad \text{item} = \mathcal{S}.\,\text{get}()$
4: $\quad \mathcal{PC} = \text{presenceCondition}(\text{item}, \mathcal{M})$
5: $\quad$ **for all** i such that $i \in \mathcal{PC}$ **do**
6: $\quad\quad$ **if** $i \notin \mathcal{R}$ **then**
7: $\quad\quad\quad \mathcal{S}.\,\text{insert}(i)$
8: $\quad\quad\quad \mathcal{R}.\,\text{insert}(i)$
9: $\quad\quad$ **end if**
10: $\quad$ **end for**
11: **end while**
12: **return** $\mathcal{R}$

Figure 4.2: Algorithm for model slicing

4.3.3 The Slicing Algorithm

In simple terms, the slicing algorithm gets as input a set of variation points and outputs another set that is larger or equal to its input. Basically, the algorithm maps the input to the output by calculating the direct and indirect dependencies of the presence conditions from the initial set.

For its operation the algorithm relies on variability models built with presence conditions as we introduced in Section 4.3.1 and on the function $\text{presenceCondition}()$ that is used to query for presence conditions in given models. The algorithm handles variability models as a *database* of presence conditions (see Table 4.1) that can be queried through the function $\text{presenceCondition}(\text{item}, \mathcal{M})$, where item is a variation point and $\mathcal{M}$ is the model. The function looks up the presence condition of item in $\mathcal{M}$ and returns it.

The algorithm is depicted in Figure 4.2. The inputs for the algorithm are the variability model $\mathcal{M}$ and a set of variation points $\mathcal{S}$ that are part of $\mathcal{M}$. The set $\mathcal{R}$ that will be returned is initialized (Line 1) with the given set $\mathcal{S}$. Then, the algorithm iterates (Lines 2–11) through the elements of $\mathcal{S}$. At each iteration one element from $\mathcal{S}$ is taken (Line 3), and its presence condition from $\mathcal{M}$ is fetched (Line 4). As we have seen in the previous section, the presence condition of a variation point is a boolean formula where the propositional symbols represent other variation points of the model. So, the algorithm iterates (Lines 5–10) over the set of symbols (i) of the presence condition ($\mathcal{PC}$) from the item ($item$) being processed. These symbols, which are variation points, are added to $\mathcal{S}$ (Line 7) and $\mathcal{R}$ (Line 8) if they do not

already belong to $\mathcal{R}$ (Line 6). Finally, the algorithm returns the set $\mathcal{R}$ that contains the initial set of variation points extended by the direct and indirect dependencies derived from it.

A variation point has a direct dependency to all variation points that are referenced in its presence condition. For example, referring to Table 4.1 we can see that the variation point `hotplug_cpu` has a direct dependency to `cpu_freq`. Indirect (or transitive) dependencies are derived by recursively chaining direct dependencies. For example, the variation point `hotplug_cpu` has an indirect dependency to `pm`. This is because `pm` is one of the direct dependencies of `cpu_freq`, that, in turn, is a direct dependency of `hotplug_cpu` .

Important for the understanding of the algorithm is the fact it iterates through the initial set $\mathcal{S}$ (Lines 2–11) and ends when this set is empty (Line 2). In each iteration one variation point is removed of the set, however, as new dependencies (variation points not yet processed) are found (Line 7), they are also added to the set $\mathcal{S}$. The algorithm returns all variation points that were initially given as a parameter, plus the variation points that were found in presence conditions.

Clearly, it is advantageous to use the slicing algorithm when the returned set contains less variation points than the model they belong to. The algorithm has the following properties:

Best case In the best case the algorithm will return a set that is equal to the initial set passed as a parameter. This way the user knows about the initial set of variation points that either their dependencies are self-contained or that they have no dependencies. Either way, the user knows what are the presence conditions that must be taken into consideration for a specific reasoning operation.

Worst case In the worst case the returned set contains all variation points of the model.

Complexity To understand the worst case complexity imagine a model where the presence condition of every variation point references every other variation point of the model. In this case the algorithm would have the complexity $\mathcal{O}(n^2)$ where n is the number of variation points in the model. However, as we will see in detail in the next chapter, this kind of variability models are rare or even inexistent. Typical real-wold, large-scale models have a structure that allow the slicing algorithm to always run with a complexity lower than the $\mathcal{O}(n^2)$ upper bound.

The goal for the user of this algorithm is to get the set of presence conditions that are required for building a desired reasoning operation A typical usage scenario is as follows:

1. The user sets the variation points he wants to combine in a reasoning operation, for example, v_1 and v_2 from model $\mathcal{M}$.

2. With these definitions he can call the algorithm: $\mathcal{R} = \text{slice}(\{v_1, v_2\}, \mathcal{M})$.

3. Using the result $\mathcal{R}$ that contains a subset of variation points of $\mathcal{M}$ the reasoning operation that motivated the use of the slicing function can be built. Lets say that the reasoning operation is the test if $\mathcal{M}$ allows a variant without v_1 and v_2. This reasoning can be written as $(\neg v_1 \wedge \neg v_2 \wedge \bigwedge_{v \in \mathcal{M}} \text{PC}(v))$. However, instead of using the whole formula $\mathcal{M}$, one can use the result from the slicing algorithm: $(\neg v_1 \wedge \neg v_2 \wedge \bigwedge_{v \in \mathcal{R}} \text{PC}(v))$, that is, the reasoning query conjuncted with the presence conditions from the variation points returned by the slicing algorithm.

In the following we introduce a specialization of this general scenario with a concrete example.

4.3.4 Example – Reasoning with Sliced Models

In Table 4.2, we show the slicing algorithm applied to the features of the power management feature model (shown in Figure 4.1). The sets $\mathcal{R}_1$ to $\mathcal{R}_9$ correspond to the results of the algorithm call for each feature individually. Note that the smallest *slices* have cardinality 4, they were obtained applying the algorithm to the features `cpu_freq`, `powersave`, `extra`, `suspend` and `hibernate`. The feature `acpi_system` provided a slice with cardinality 8, which was the largest.

As expected, the features with less dependencies in the model yield a smaller slice, and the features with more dependencies, a larger slice. It is also important to note that the results shown in Table 4.2 are for slices of single features. However, we can extrapolate the results for combinations of features. This is possible because applying the algorithm to a group of features is equal to applying it to each feature individually and making the union of the results. For example, the call $\text{slice}(\{\texttt{apci\_system}, \texttt{extras}\}, \mathcal{M})$ results in a set with all features of the model, as it can be inspected with the data from Table 4.2, the result of $\mathcal{R}_2 \cup \mathcal{R}_7$ is the set of all features.

Of course, the slicing algorithm is beneficial when the size of the slices is considerably smaller than the model size. However, the example shown in Table 4.2 does not aim at quantifying the real benefits of the slicing algorithms, it is rather an illustrative example which leads to an improved understanding of the algorithm.

$\mathcal{R} = \text{slice}(v)$
$\mathcal{R}_1 = \text{slice}(\{\texttt{acpi}\}, \mathcal{M})$
$\mathcal{R}_1 = \{\text{acpi}, \text{pm}, \text{hotplug\_cpu}, \text{cpu\_freq}, \text{powersave}, \text{performance}\}$
$\mathcal{R}_2 = \text{slice}(\{\texttt{acpi\_system}\}, \mathcal{M})$
$\mathcal{R}_2 = \{\text{acpi\_system}, \text{acpi}, \text{pm}, \text{hotplug\_cpu}, \text{cpu\_freq}, \text{powersave}, \text{performance}\}$
$\mathcal{R}_3 = \text{slice}(\{\texttt{cpu\_freq}\}, \mathcal{M})$
$\mathcal{R}_3 = \{\text{cpu\_freq}, \text{pm}, \text{porwersave}, \text{performance}\}$
$\mathcal{R}_4 = \text{slice}(\{\texttt{hotplug\_cpu}\}, \mathcal{M})$
$\mathcal{R}_4 = \{\text{hotplug\_cpu}, \text{pm}, \text{powersave}, \text{performance}\}$
$\mathcal{R}_5 = \text{slice}(\{\texttt{powersave}\}, \mathcal{M})$
$\mathcal{R}_5 = \{\text{powersave}, \text{cpu\_freq}, \text{performance}, \text{pm}\}$
$\mathcal{R}_6 = \text{slice}(\{\texttt{performance}\}, \mathcal{M})$
$\mathcal{R}_6 = \{\text{performance}, \text{cpu\_freq}, \text{powersave}\}$
$\mathcal{R}_7 = \text{slice}(\{\texttt{extras}\}, \mathcal{M})$
$\mathcal{R}_7 = \{\text{extras}, \text{pm}, \text{suspend}, \text{hibernate}\}$
$\mathcal{R}_8 = \text{slice}(\{\texttt{suspend}\}, \mathcal{M})$
$\mathcal{R}_8 = \{\text{suspend}, \text{extras}, \text{pm}, \text{hibernate}\}$
$\mathcal{R}_9 = \text{slice}(\{\texttt{hibernate}\}, \mathcal{M})$
$\mathcal{R}_9 = \{\text{hibernate}, \text{extras}, \text{pm}, \text{suspend}\}$

Table 4.2: Results of the slicing algorithm applied to all features of the feature model shown in Figure 4.1.

As we will show in the next chapter, and also studied by others in detail elsewhere [Lot09, SLB+10, LSB+10], real-world configuration models tend to be shallow and contain many branches that are independent from each other. In these scenarios the slicing algorithm performs best.

Our goal here is to show how to build up a reasoning operation using slices. We use the code from the Linux kernel shown in the Listing 4.1 to envision the following scenarios:

- Consider a configuration tool that displays the configuration variability model so that the user can derive a variant. At each user action the tool has to decide if the partial configuration is valid or not. If not, the user should be warned that his decision is inconsistent with the model. Using the feature model from Figure 4.1 and the slices shown in Table 4.2 it is easy to see that the user could make several partial selections that would end up in a reasoning operation containing only a few features instead of the whole model. The configuration of complete branches can be performed and validated using only the presence conditions of a few features. For example, the slices for the features under the branches `extras` and `cpu_freq` are self-contained, that is, they do not reference features from other branches. As a result, the formulas for checking the validity of any possible partial configuration under `extras` (or `cpu_freq`) require only the presence condition of 4 features. On the other hand, the branch under the feature `acpi` has dependencies on the branch `cpu_freq`, as a result, the slices are larger. So, if the user's first choice is the selection of feature `acpi`, the configuration tool will generate a validity test containing the presence condition of 8 features. It is unavoidable to have reasoning operations at certain points in the configuration process that will require all features of the model. Nevertheless, several steps in between can benefit from slicing, which might improve, for example, the responsiveness of the tool.

- In Listing 4.1 we show an excerpt of the file `kernel/sched.c` from the Linux kernel. Consider the scenario where the developer responsible for this file decides to extend it, specifically the function `migration_call` (Line 10) to support the feature `cpu_hotplug`. After inspecting this function the developer decides to insert the new code to the function (Lines 16–23). The code between the CPP directives `#ifdef CONFIG_CPU_HOTPLUG` and `#endif` is specific to the feature `cpu_hotplug`. Note that for the decision to include the feature-specific code at this location the developer has to deal with the information of different variability models. First, the source file itself. As we can see in the listing, the file in question is a large file with more than 9000 lines. The code we show in the listing might be nested in other CPP directives, making this snippet

dependent on other features. This is exactly the case in this file, where 218 lines above the snippet shown here we found the directive `#ifdef CONFIG_SMP`. Typically, researchers [Käs10] consider one page of source code as a block of 50 lines of code. This means that the developer would have to inspect more than 4 pages of code to find out this dependency. Another important variability model that might influence the new code is the build system. The developer has also to consider the conditions under which the file will be compiled. The case for the file `kernel/sched.c` is simple since it is unconditionally compiled in every variant. Finally, the last variability model that can influence the code in our example is the configuration variability model, in the case of Linux, the KCONFIG. These dependencies are represented here in a simplified version as the feature model shown in Figure 4.1. To build a reasoning operation that checks for the conformity of the new code with variability models of the project we can perform the following steps:

(1) build a variability model of the source file `kernel/sched.c` using the checker function we introduced in the previous chapter. The resulting checker function for this file is a boolean formula containing the conjunction of presence conditions of more than 300 conditional blocks over 30 features.

(2) apply the slicing algorithm to the generated checker function using as initial set of variation points the block of code of interest (Line 16), which results in a formula containing 2 blocks over 2 features (`CONFIG_SMP` and `CONFIG_HOTPLUG_CPU`).

(3) using the features found in the previous step we can slice the configuration variability model. If we use its simplified version shown in Figure 4.1 we get as slice the set $\mathcal{R}_4$ as shown in Table 4.2.

(4) using the slices generated in the previous steps, lets call them s_1 and s_2 we can build the reasoning operation $\mathrm{SAT}(s_1 \wedge s_2)$ that will check if the added feature-dependent code from Listing 4.1 is conform with the variability models it depends on.

The examples and scenarios presented above aim at illustrating a few use cases of the slicing algorithm. A detailed and more formal study on how to improve reasoning on variability models with the slicing algorithm is presented in Chapter 5. The quantitative evaluation of the benefits that result from the application of the algorithm for real models is discussed in Chapter 6.

4.4 Related Work

Reasoning on variability models is a very broad area that has many links to our work. In Section 4.2 we referred to the approaches of BENAVIDES [BRCT05], MENDONÇA [Men09] and JANOTA [Jan10] to explore the scalability issues of variability models. Here we further discuss techniques that are related to our slicing approach.

ACHER et al. [ACLF11] tackle the problem of managing the complexity of building very large feature models. They argue that *decomposition* supported by *separation of concerns* can effectively improve the construction of such large models. They provide *composition* operators, with defined semantics that are preserved during transformation, for insertion, merging and aggregation of feature models. These operators enable feature models to be synthesized from smaller feature models. Most related to our work, however, is their approach for feature model slicing. This technique produces a projection of a feature model given a set of selected features, and it supports the creation of semantically meaningful decompositions of feature models. The slicing is an unary operation on a feature model that gets as input a subset of features of the model it is being applied to. The result is a new feature model that has the same semantics (for the subset of features) as the original model. The algorithm for calculating the slice has two steps. First, the propositional formula that represents the slice is calculated. Second, using the techniques developed to transform feature diagrams to logic and back [CW07, SLB+11], they transform the slice formula back to a feature diagram representation. The second step is essentially different to our approach for variability model slicing. For our slices we are only interested in propositional formulas to support reasoning, and not in diagrams or composable models. As a result, their approach is more complex, for example, by performing existential quantification using BDDs, and for the BDD construction the heuristics from MENDONÇA [Men09] are used, which limits the sizes of the models that can be used with this approach.

The crosschecking between different variability models is also related to our work. METZGER et al. [MHP+07] present several formalizations that allow for consistency checks between variability models, however these models must conform to a given format, and techniques to convert from other formats are not given. CZARNECKI et al. [CP06] present an approach to check the consistency of feature-based model templates, that is, checking consistency of feature models and model templates. THAKER et al. [TBKC07] present an approach for the safe composition of product lines, it is much related to our idea of crosschecking. However, their approach exploits the structure of the code, which is based on the *feature-oriented software development*. Therefore, it cannot be used in projects that rely on the CPP like the Linux kernel. We believe our technique is complementary to these approaches

as they also use propositional logic. If these formulas can be adapted to use our definition of presence conditions, they could also be used in combination with our slicing algorithm.

4.5 Summary

In this chapter we introduced the concept of model slicing. We motivated it by presenting the different roles that variability models can play, which led us to the classification of implicit and explicit variability models. Subsequently, we discussed the scalability issues of reasoning on variability models. We also showed some of the limits, intractability and unacceptable processing time, that other researchers have reached when performing reasoning operations on variability models. We introduced a new format of presence conditions for variation points and the slicing algorithm that allows us to combine such variation points for specific reasoning operations, such combinations are called model slices.

In the next chapter we present several reasoning operations that can benefit from the slicing algorithm. We also show in detail how to combine the techniques from the previous chapter, namely, the checker function for CPP-based source files, and the slicing algorithm in order to form a reasoning framework to reveal variability bugs in the Linux kernel. The quantitative and qualitative evaluation of the slicing algorithm is presented in Chapter 6.

```
1
2 [...] // 6622 lines
3
4 /*
5  * migration_call - callback that gets triggered when a CPU
6  * is added. Here we can start up the necessary migration
7  * thread for the new CPU.
8  */
9 static int __cpuinit
10 migration_call(struct notifier_block *nfb, unsigned long action,
11                           void *hcpu)
12 {
13
14 [...] // 32 lines
15
16 #ifdef CONFIG_CPU_HOTPLUG
17         case CPU_UP_CANCELED:
18         case CPU_UP_CANCELED_FROZEN:
19                 if (!cpu_rq(cpu)->migration_thread)
20                         break;
21 [...] // 59 lines
22
23 #endif
24
25 [...] // 2922 lines
```

Listing 4.1: Example of Conditional Blocks

5

Multi-model Reasoning About Variability Bugs

In the previous chapters we introduced the concepts of *source code variability models* and *slicing variability models*, here we use these concepts to explain how to perform multi-model reasoning operations, which are based upon a combination of different model types. First, we discuss how variability models can be hierarchically organized. Second, we introduce the basic reasoning operations that are able to reveal problematic variation points. Finally, we introduce our definition of variability bug. *This chapter shares material with the GPCE'10 paper 'Efficient Extraction and Analysis of Preprocessor-Based Variability'[STLSP10] and the FOSD'09 paper 'Dead or alive: Finding zombie features in the Linux kernel'[TSSPL09].*

5.1 Introduction

The main goal of this thesis is to provide techniques for the detection of problems that are caused by an incorrect management of variation points. We have argued that to find such problems the different variability models have to be converted to a common representation so that the corresponding reasoning operations can be applied. In Chapter 2 we explained that the usual abstraction means for reasoning in software product lines is propositional logic and there are established and formalized [BRCT05, CW07, MWC09, TBK09] reasoning operations using it. In order to better integrate CPP-annotated source files into the variability management process, we have introduced in Chapter 3 a lightweight method to build a variability model in the form of a propositional formula for CPP-based source files. Moreover, to deal with the problem size when combining several models, we presented in Chapter 4 the concept of model slicing that allows us to selectively use a subset of variation points from a variability model. These are, however, only a few of the building blocks that are required for the detection of variability problems in software product lines.

For the construction of a holistic approach that takes into consideration all models of a project, we not only have to (1) have all models converted to a common format, (2) deal with problem size, but also (3) understand how such models are hierarchically organized and (4) understand how one model can constrain other models. Topic (1) has been addressed in this thesis for source files, and for a diversity of models by many others, for example, feature models [BRCT05], KCONFIG [ZK10, TSD+12], KBUILD [BSL+10a, NH12, DTSPLa], and also for higher abstractions like parse trees [KAT+09] and type systems [KATS11]. Topic (2) has also been addressed by the slicing approach. In this chapter we will address the issues (3) and (4) in order to define reasoning operations that take into account variation points of possibly all models of a product line. In the following we assess how variability models are organized in product lines. Subsequently, we devise a set of reasoning operations that combine variation points from different models. Finally, we discuss the problems that can be detected with these operations and define the concept of variability bugs.

5.2 Hierarchical Organization of Variability Models

Large software product lines which employ a heterogenous variability management comprise several variability models. These models can be explicit like configuration models, or implicit in the form of build systems, techniques for feature realization in the source code, and others. These models are normally hierarchically organized so

that models on higher levels constrain models in lower levels.

SVAHNBERG [Sva00] studied in his thesis how variability is hierarchically organized in software product lines. He argues that variability points can be introduced at different levels of abstractions for example (from higher to lower level): *architecture description*, *design*, *source code*, *compiled code*, *linked code* and *running code*. According to him, a feature in a particular level of abstraction is specialized in a group of less abstract features in the lower level.

In our previous work [SSSPS07, SSP08] about variability management in the Linux kernel we drew similar conclusions regarding how models are hierarchically organized. In Linux we have at the highest level the KCONFIG model. This model defines all features that are available in the system, and their inter-dependencies, which in turn define the combinations of features that form valid and invalid configurations. Below the KCONFIG model, on the build system level, we find the KBUILD model. This model uses a given valid feature selection to choose the corresponding files that need to be selected. Inside the files that are to be compiled there are CPP directives, which form the next model in the hierarchy. The CPP directives are used to define the parts of the source code that should be selected during preprocessing for compilation. The result of compilation are binary objects that have to be linked, the rules that guide object linking can also be seen as a variability model, which comes next in the hierarchy. Finally, the code that is ready to run can contain instructions that use specific features of the configuration model. For example, in the Linux kernel it is normal to use conditionals in the source code (e.g. `if (CONFIG_SMP){}`) that check if a feature is selected. Of course, these language constructs represent decisions that will be taken at runtime. As a result, these runtime decisions form a variability model at the lowest level of the variability models hierarchy. It is important to understand what types of variation points each of these models define. In the following we highlight what are the *variation points* in the different *variability models* of the Linux Kernel:

KCONFIG. In the KCONFIG model the variation points are the **features** of the system.

KBUILD. In the KBUILD model the variation points are the **source files** and **object files** that need to be compiled and linked. The rules that define when a specific file will be compiled or linked are built as a logic expression that uses as symbols the features from the KCONFIG model.

CPP. The CPP annotations that control the **blocks of code** will be selected during preprocessing are the variation points at this level. They are also built as logic expressions over the features of the KCONFIG model.

C. At the C language level, the variation points are **sets of statements** that are defined under a logic expression built over features of the KCONFIG model.

It is easy to see that in Linux the variation points are: features, source files, blocks of code, object files, and sets of statements. All of them constrained through the features of the KCONFIG model. Note that, during the configuration process of a kernel image, the rules of the KCONFIG model take effect in different phases. First, for the selection of files to be compiled, then for choosing the combinations of files to be linked. In between these two phases preprocessing takes place, and so on. The interaction among models can be organized hierarchically, where the layers and arrows are used to distinguish if a model constrains other models, or if a model is constrained by other models. The higher the model the more it influences other models, an arrow between two models represents that the higher model constrains the lower model. The hierarchical organization of models of the Linux kernel is shown in Figure 5.1. This is, however, a simplification of the real hierarchy. One can go deeper on these layers and split them into sub-layers. For example, at the language level we could treat the syntactic and semantic levels separately. We could also treat the type system of the language at a different level. The 5-layered model presented in Figure 5.1 helps us to understand how variability models constrain each other. However, different projects can have a different hierarchical arrangement of their models. The key to performing multi-model reasoning operations is not imposing a specific arrangement of the models. Rather, it is essential to understand how the models interact with each other so that the multi-model reasoning operation can be precisely built.

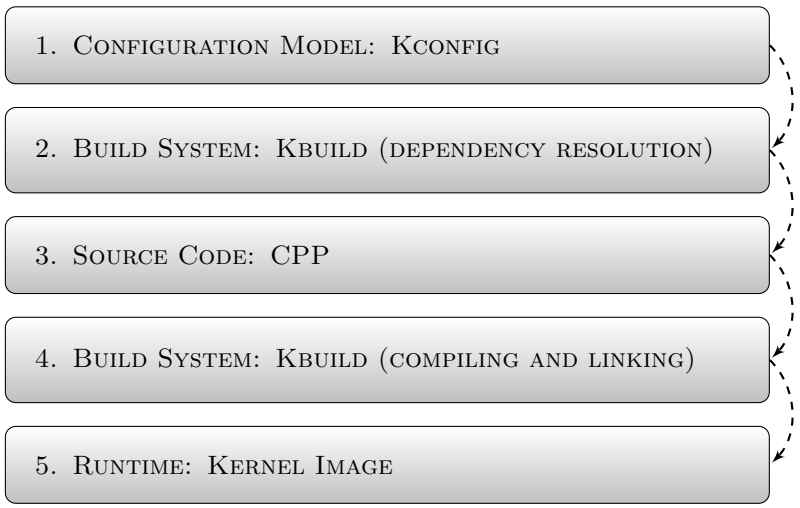


Figure 5.1: 5-layered Hierarchy of Variability Models in the Linux kernel.

To illustrate how higher-level models constrain lower-level models in practice we show some examples in Table 5.1. In this table we show excerpts of the definition of variation points at three different levels. In the first line we see the definition of the variation point `ACPI` which is a feature of the KCONFIG model. This feature has a few requirements that need to be met so that it can be selected. Namely, the feature `IA64_HP_SIM` must not be selected, either the feature `X86` or `I464` must be selected and the feature `PCI` must be selected. These requirements form the presence condition for the feature `ACPI`, which is shown as a boolean formula in Table 5.1. From this presence condition we can draw some conclusions. For example, the features `ACPI` and `IA64_HP_SIM` will never be present in the same kernel configuration, since the KCONFIG construct '`depends on ! IA64_HP_SIM`' in the definition of the feature `ACPI` makes them mutually exclusive. This is just one of the possible conclusions that can be drawn from this presence condition. We could further analyze the relation of `ACPI` and the other features it references in its definition: `IA64`, `X86` and `PCI`. However, the important lesson to take from here is that the relation between these features has an impact on lower-level variability models.

In the second line of Table 5.1 we see the KBUILD constructs that set the requirements for the file `bus.c` to be compiled. The construct '`obj-ACPI += acpi/`' simply informs the KBUILD system that the sub-directory `acpi/` must be processed if the feature `ACPI` is set. This is the subdirectory that contains the file `bus.c`. The line '`acpi-y += bus.o`' belongs to the makefile inside the subdirectory `acpi/` and informs the KBUILD system that whenever it is executed it must generate the file `bus.o`. As a result, the requirement for the compilation of the file `bus.c` is the selection of the feature `ACPI`. Consider what happens when a developer makes the following change:

```
1 diff --git a/drivers/acpi/Makefile b/drivers/acpi/Makefile
2 --- a/drivers/acpi/Makefile
3 +++ b/drivers/acpi/Makefile
4
5 -acpi-y += bus.o
6 +acpi-IA64_HP_SIM += bus.o
```

With this change, the file `bus.c` will be compiled when the features `ACPI` and `IA64_HP_SIM` are selected. Recall that in the KCONFIG model these features are defined to be mutually exclusive, thus with this change the file `bus.c` will actually never be compiled. This is the result of a higher-level model constraining a lower-level model. Moreover, there is no change that can be made on the Makefile that will have a similar impact on the KCONFIG model, this is because lower-level models cannot constrain higher-level models.

In the third line of Table 5.1 we show an excerpt of the file `bus.c` that is annotated with CPP directives. In this code there are two code blocks that will be selected

(or not) in the preprocessing phase. One is an `#if` directive that depends on the `X86` feature, the other is its `#else` counterpart. Note that the preprocessing will take place only if the KBUILD decides that the goal `bus.o` must be executed. As we have seen above, this goal will only be executed if the feature `ACPI` is selected, which, in turn, requires the feature `X86` to be selected. Consequently, whenever the file `bus.c` is compiled, the first branch of the `#if` directive will be selected, and the `#else` will never be selected. This is, again, the result of the higher-level models constraining a lower-level model.

MODEL	VARIATION POINT	DEFINITION

Kconfig	ACPI	`menuconfig ACPI`
		`  bool "ACPI Support"`
		`  depends on !IA64_HP_SIM`
		`  depends on IA64 || X86`
		`  depends on PCI`

	PC(ACPI)	$ACPI \rightarrow (\neg IA64\_HP\_SIM) \wedge (IA64 \vee X86) \wedge PCI)$
Kbuild	bus.c	`obj-$ACPI += acpi/` `acpi-y += bus.o`
	PC(bus.c)	$bus.c \rightarrow ACPI$

CPP	code block	`#ifdef X86`
		`static int set_copy_dsdt`
		`  (struct dmi_system_id *id)`
		`  { [...] } // Block 1`
		`#else`
		`static struct dmi_system_id`
		`  dsdt_dmi_table[] = {`
		`  { [...] } // Block 2`
		`};`

	PC(B1)	$B1 \rightarrow ACPI \wedge bus.c$
	PC(B2)	$B2 \rightarrow \neg B2$

Table 5.1: Examples of the definition of variation points at three different levels: configuration model, build system and source code.

5.3 Reasoning Operations on Variability Models

We discussed the general reasoning operations on single models in Section 2.1.4. However, our goal is to perform multi-model reasoning operations. With the understanding of the hierarchical structure of variability models we can now explore how to use the additional information to perform multi-model reasonings.

For modeling our reasoning operations we will use propositional logic. So, we will formulate questions in the form of a logic formula. The answer is provided by a SAT solver. These solvers get as input a boolean formula and respond with `YES` if an assignment that makes the formula evaluate to true is found, or `NO`, in case such an assignment cannot be found. That is, the solver responds the question whether the given formula is *satisfiable* or not. Thus, we have to construct the reasoning formulas so that the possible outcomes (satisfiable or unsatisfiable) have a meaning. For example, consider that φ is the formula representing a feature model, the operation $\mathrm{SAT}(\varphi)$ will return `YES` if a valid assignment is found or `NO` otherwise. Before applying such an operation, the meaning of the possible outcomes must be clear. In this case, if the formula is satisfiable it means that the feature model φ is not void, that is, it has at least one valid configuration. Consequently, if the formula is unsatisfiable the feature model has no valid configuration, as a result, the feature model is void.

When dealing with multi-model reasoning we not only have to formulate meaningful questions in the form of boolean formulas, but also we have to properly combine models. As we have discussed above, the constraints of higher levels can further constrain variation points in models in lower levels, but not vice-versa. To illustrate this, consider the rules from the build system that define the conditions under which a source file will be compiled. It does not matter how it is built, it will never have any influence on the configuration model on the highest level. On the other hand, if the configuration model defines any two features to be mutually exclusive, and we have a conjunction of these features as the rule that defines when a source file is to be compiled, we have an inconsistency. This is because no valid configuration allows the coexistence of both features together. This inconsistency is the result of a higher-level model constraining a lower-level model.

As we discussed in the previous chapter, we treat variability models as sets of presence conditions. The fundamental problem that can occur in variability models is the occurrence of variation points that are intended to be *configurable* (having the ability to be turned on and off) but actually cannot. This inconsistency can manifest itself in two different ways. First, in the form of a variation point that does not appear in any valid configuration, what we define as a **dead** variation point. Second, in the form of a variation point that appears in all valid variants, what we define as an **undead** variation point. The first type is always a problem, as it is a useless

variation point. The second needs more discussion. Of course a mandatory feature that is child of the root feature in a feature model will be present in all valid configurations, and it is not a problem. However, a block of code that is annotated with `CPP` directives will always be recognized by the developer as a *configurable* block, that is, a block that is not necessarily present in all configurations. Otherwise it would not be explicitly marked with preprocessing directives. These cases of *seemingly* conditional variation should be detected by reasoning operations so that developers can be warned.

On this basis, we define two reasoning operations that can be applied in multiple variability models to reveal variation points that are expected to be variable but are not.

Definition 6 *[Dead variation point] given a variation point v that belongs to the variability model φ_n and a set of higher-level models $\varphi_1, \varphi_2, \ldots, \varphi_{n-1}$ that constrain the model φ_n, we define the formula ϕ_d that can be used to check if v is dead as follows:*

$$\phi_d \stackrel{def}{=} \left((\bigwedge_{i=1}^{n} \varphi_i) \wedge v \right) \tag{5.1}$$

If the formula ϕ_d is satisfiable, which can be checked by the operation $\mathrm{SAT}(\phi_d)$, it means that the variation point v is not dead. If the SAT solver returns `False` it means that there exists no assignment that at the same time makes the models involved and the variation points true. This behavior characterizes a dead variation point.

Definition 7 *[Undead variation point] given a variation point v and its parent p that belongs to the variability model φ_n and a set of higher-level models $\varphi_1, \varphi_2, \ldots, \varphi_{n-1}$ that constrain the model φ_n, the formula that can be used to check if v is undead is defined as follows:*

$$\phi_u \stackrel{def}{=} \left((\bigwedge_{i=1}^{n} \varphi_i) \wedge \neg v \wedge p \right) \tag{5.2}$$

If the formula ϕ_u is satisfiable then the variation point v is not undead. This can be checked by the operation $\mathrm{SAT}(\phi_u)$. If the SAT solver returns `False` there is no assignment that makes the models true, the parent true and the variation point false. So, given the basic constraint of a variation point—its parent—it cannot be turned off, and will always be true. This behavior defines an undead variation point.

As we will see in the next chapter, where we apply these operations to artifacts of real projects, this approach leads to correct results and turns out to be practical. However, in the two definitions above we made assumptions that do not hold

```
1
2 diff --git a/kernel/smp.c b/kernel/smp.c
3 --- a/kernel/smp.c
4 +++ b/kernel/smp.c
5
6  -#ifdef CONFIG_CPU_HOTPLUG
7  +#ifdef CONFIG_HOTPLUG_CPU
```

Listing 5.1: Fix for a symbolic defect

for real-world projects. Recall that the models φ_1, φ_2, ..., φ_{n-1} are built as sets of presence conditions like $\mathrm{PC}(v)$. In real scenarios we have to use tools that extract these presence conditions from source files, build files, configuration files, and so on. Moreover, many projects do not offer tool support to check if these files are at least syntactically correct. As a result, we cannot assume that the extracted formulas in the form of $\mathrm{PC}(v)$ are always correct.

To illustrate this, consider the patch that was applied to the Linux kernel 2.6.30 that is shown in Listing 5.1. This patch changes only one line of code that contains an `#ifdef CPP` directive. We can see that the developer made a spelling error when referencing the feature he was about to implement. The inexistent symbol `CONFIG_CPU_HOTPLUG` was used instead of the correct symbol, that was defined in the KCONFIG model as `CONFIG_HOTPLUG_CPU`.

As a result, we have to detect and to deal with these kind of mistakes when building the presence conditions of variation points. In this particular case, a presence condition $\mathrm{PC}(v) =$ `CONFIG_CPU_HOTPLUG` would be wrong.

To deal with this problem we have to further constrain the presence conditions that make use of symbols that are not defined in the configuration model. Basically, these symbols cannot be selected by the user in the configuration process, and, therefore, can never be set to true. Thus, in the reasoning operations that use these symbols, we have to set them to `False`. We define the following formula to set the value of these symbols to be always false:

Definition 8 *[Missing symbols] given a set of boolean symbols $\mathcal{S}$, a configuration model φ_1 built over the symbols from $\mathcal{S}$, a variation point v and its presence condition $\mathrm{PC}(v)$ from φ_1, we define a boolean formula that represents the missing symbols of $\mathrm{PC}(v)$ as follows:*

$$\phi_m \stackrel{def}{=} \left(\neg \bigwedge_{s \notin \mathcal{S}} s \right) \tag{5.3}$$

This formula is the negation of the conjunction of all symbols from $\mathrm{PC}(v)$ that do not belong to $\mathcal{S}$. This formula will evaluate to `True` only when all symbols $s \notin \mathcal{S}$

are set to `False`. We can use this formula in combination with reasoning operations that use the presence condition $\mathrm{PC}(v)$ in order to reveal problems that stem from the incorrect use of boolean symbols.

Using the formulas ϕ_d and ϕ_u we can perform reasoning operations to detect dead and undead variation points, respectively. We can further constrain these formulas with the formula ϕ_m to detect symbolic problems, in total, we have the cases shown in Table 5.2. Note that we constructed the operations in a way so that when the SAT solver returns `True` there is no problem detected. When the SAT solver returns `False` then we have one of the four problematic cases.

Variation Point Problem	Revealed by
Logic Dead	$\mathrm{SAT}(\phi_d) == \texttt{False}$.
Logic Undead	$\mathrm{SAT}(\phi_d \wedge \phi_m) == \texttt{False}$.
Symbolic Dead	$\mathrm{SAT}(\phi_u) == \texttt{False}$.
Symbolic Undead	$\mathrm{SAT}(\phi_u \wedge \phi_m) == \texttt{False}$

Table 5.2: Four types of problems with variation points.

The reasoning operations presented above can be further optimized to use the slicing algorithm presented in the previous chapter. By employing slicing we can reduce the size of the models φ. Instead of combining the models $\varphi_1 \dots \varphi_n$, we could use the models $\varphi'_1 \dots \varphi'_n$ that can be obtained by subsequent calls of the slicing algorithm. The first call has to initialize the set of variation points to $\{v\}$, and the resulting slice can be used as the initial set for the following upper model in the hierarchy. This can be done successively up to the highest level. The benefits of slicing in this fashion is studied in the next chapter.

Also important when applying reasoning to variability models is to produce good explanations about the problems found. Using the formulas from Definition 6 and Definition 7 we observe the following characteristics that ease the generation of plausible explanations:

1. The operations can be applied to both single- and multi-models. Moreover, the models $\varphi_1, \dots, \varphi_n$ built as sets of presence conditions (see Section 4.3.1) can be individually validated (checking for dead and undead variation points) in preparation for multi-model reasoning operations. Consequently, the internal inconsistencies of the variability model can be fixed or ruled out.

2. When combining multiple models using the formulas from Definitions 6 and 7 we can perform repeated calls to the SAT solver instead of only one in order to find out which model is triggering the inconsistency. Consider a 3-layered project with source files (φ_3), build system (φ_2) and configuration model (φ_1).

> It is desired to check if a CPP-based block of code (v) is dead. The reasoning operation according to Definition 6 would be $\mathrm{SAT}(\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge v)$. Suppose the SAT solver returns `False`, so the variation point v is dead. The questions that immediately arise in this case are: *why is v dead?*, *what causes it?* The inconsistency could be in any of the possible combinations of the involved models. To improve on this situation we can apply successive SAT calls, for example: (1) $\mathrm{SAT}(\varphi_3 \wedge v)$, (2) $\mathrm{SAT}(\varphi_2 \wedge \varphi_3 \wedge v)$, (3) $\mathrm{SAT}(\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge v)$. Starting from the lowest-level model, and adding the higher-level models to the reasoning formula up to the highest, we are able to distinguish which is the lowest-level model that is in contradiction with models on the previous levels.

It is important to note that Definitions 6 and 7 are the basic operations for performing multi-model reasonings. They allow us to combine models and discover variation points that are not *variable*, which we consider to be a problem. Moreover, depending on how the variation point v is defined, the operation will have a specific meaning. Naturally, these variation points can be representations of a block of code in a source file, a source file that is controlled by the build system, or even a feature of the feature model, among other artifacts of software product lines.

5.4 Variability Bugs

In the previous section we defined reasoning operations that can be used to reveal variation points that are not variable. In order to understand why this is a problem, let us revisit a broadly accepted definition of variability [SvGB06]:

> Software variability is the ability of a software system or artifact to be efficiently extended, changed, customized or configured for use in a particular context.

The specific parts of the system that are marked to be extended, changed, customized or configured are represented as variation points. Clearly, from the definition we can conclude that a variation point that cannot be configured does not contribute to the overall variability. Furthermore, variation points basically exist to mark specific parts as configurable. As a result, we consider *invariable* variation points to be problematic, as they contradict their own fundamental purpose.

In previous work, we focused on variability problems that are caused due to inconsistencies between the problem space and the solution space [STE+10, TLSSP11]. In these works, we defined these inconsistencies as *configurability defects:*

> A **configurability defect** (short: **defect**) is a configuration-conditional

> **item** that is either **dead** (never included) or **undead** (always included) under the precondition that its **parent** (enclosing item) is included.

In this thesis we introduced several techniques that allowed us to build a generic reasoning infrastructure for multi-model projects. By following a few definitions like converting the variability model to propositional logic, and employing our description of presence conditions, any project can benefit from our reasoning operations to detect problems with variation points. As a result, we need a more generalized definition to categorize the problems that can be revealed with our reasoning infrastructure.

A *software bug* is the usual definition given to the malfunction of software systems. This malfunction occurs due to a *fault* that leads the system to behave in unexpected or undesired ways. Very often, this fault can be the result of an error in the source code or in the design of the system. Actually, the variability problems that we discuss in this thesis can be also seen as some kind of software bug. First, they represent unexpected and undesired behavior, since variation points are expected to be variable. Second, they can be the result of mistakes in the definition and usage of variation points, which is done in source files and is also part of the design. However, simply defining the variability problems as software bugs is not adequate. The class of software bugs is too general, and involves too many types of problems. The problems that we deal with in this work are in fact a very particular type of bug.

We define the behavior caused by variation points that are not variable as variability bugs:

> A **variability bug** is the unexpected behavior that is caused due to a faulty definition or usage of variation points.

The basic four types of variability bugs that we discussed are: dead and undead variation points that can be further categorized in logic and symbolic (see Table 5.2). They represent the elementary variability bugs, that can actually trigger any type of software bugs. For example, defining a function in a context, and its call in other contexts, if these contexts (which have their own presence conditions) can be mutually-exclusively configured, the configuration with the call and without the function definition will not compile in some languages and will not link in others. Variability bugs can be a problem on the variability management level, as a feature of the configuration model that can never be selected, a source file that is never compiled, or a conditionally-compiled source code block that is always compiled. Also, as exemplified above, it can trigger any other type of software bug on different abstraction levels.

5.5 Summary

In this chapter we studied how variability models are hierarchically organized in product lines. We introduced a typical 5-layered hierarchical organization that we used to derive how models can constrain each other. On this basis we defined the basic reasoning operations that allow us to find variations points that are not variable. We categorized this problematic variation points in dead and undead that can further be sub-classified in logic and symbolic.

Furthermore, we discussed the faulty behavior that is caused by problematic variation points and defined it as variability bug. In the next chapter we will study in detail variability bugs in real-world, large-scale projects.

"One of my most productive days was throwing away 1000 lines of code."

Ken Thompson

6 Variability Bugs in the Real

This chapter aims to evaluate the techniques and tools which were developed in the context of this thesis. We will perform a detailed analysis of *variability bugs* in the code base of the Linux kernel. For the analysis we use our tool `undertaker`, which implements the algorithms for source code variability models described in Chapter 3, the slicing algorithm explained in Chapter 4 and the multi-model reasoning framework described in Chapter 5. *This chapter shares material with the EuroSys'11 paper 'Feature Consistency in Compile-Time–Configurable System Software'[TLSSP11].*

6.1 Introduction

In this thesis we have studied several topics related to reasoning operations in the realm software product lines. We provided techniques that are able to reveal variability bugs. To achieve this goal we have introduced the concepts of source code variability models (Chapter 3), slicing variability models (Chapter 4), and multi-model reasoning operations to reveal variability bugs (Chapter 5). Nevertheless, as we pointed out in Section 4.2, reasoning techniques based on propositional logic frequently exceed the performance limits of BDD libraries and SAT solvers, which turns the tackled problems intractable by these approaches. Therefore, a reasonable evaluation of techniques that involve reasoning operations has to carefully analyze the performance for the *generation* and *solution* of the underlying computational representation of the reasoning problem. Moreover, in this thesis we also raised the hypothesis that variability bugs will eventually appear in projects that employ a heterogeneous variability management. In order to verify this claim we have to apply our techniques to real-world projects.

As a consequence, the evaluation presented is this chapter focuses on three criteria:

Performance. Our techniques aim to reveal variability bugs automatically. Normally, a static analysis tool can be employed in two distinct ways: it can both support developers while they implement new features, as well as when they want to perform a quality analysis on their complete code base. The former is what we call support for *incremental builds*, that is, the developer makes changes on a few files, and wants to check if any variability bug was introduced by these changes. The latter can be regarded as a comprehensive analysis of the code base, that is, all files are analyzed at once in order to find potential bugs. In both cases, developers will adopt the analysis tool only if it can deliver results quickly. To build a Linux kernel image many steps are involved, like dependency resolution, preprocessing, compiling and linking. If we want to include some analysis is this process, it has to be fast.

Accuracy. When building large projects from code, the developers are overwhelmed with messages from different tools employed in the build process. These messages appear mostly in the form of errors or warnings. Adding more messages to this process regarding variability bugs will further increase the bulk of information that the developers have to deal with. In this situation it is crucial for the acceptance of the analysis tool by the developers, that the generated messages are accurate. Accuracy in this context means that the messages about variability bugs are always correct, that is, the tool should not report false negatives or false positives. Otherwise, if developers would have to check whether

each error message is really accurate, then the tool for variability bug detection would be perceived as a burden and would not be adopted.

Relevance. There are a few important questions regarding the relevance of our approach that have to be answered: *Do variability bugs exist in real-world projects? If yes, how much? Is the fixing of variability bugs important to the maintainers?* Answering these questions will help us to understand if our techniques can support developers to find problems that are important to them. If we can answer all questions positively by means of our evaluation, we can also conclude that in general our tooling and approaches are relevant to the solution of variability bug problems in real-world projects.

In the next section we briefly introduce the tool that implements the techniques of this thesis. In the following, we present a series of experiments that confirm the performance, accuracy and relevance of our approach for the automatic detection of variability bugs in the Linux kernel.

6.2 The `Undertaker` Tool

The techniques regarding source code variability models, model slicing, and variability bug detection were presented in Chapter 3, Chapter 4 and Chapter 5, respectively. Although motivated by practical examples extracted from the Linux kernel, we explained these techniques in an abstract form, that is, we did not introduce these concepts in the context of a specific tool or project. We aimed to provide a logical foundation of our techniques so that they could be used in other contexts and projects. However, all of our approaches were implemented and tested.

The initial ideas of this thesis were implemented in several prototypes that later led to the development of the open source project `undertaker`. The `undertaker` tool set is part of the VAMOS[1] project (VAriability Management in Operating Systems) and was developed by members of this project, including the author of this thesis.

Moreover, the tool supports various variability reasoning operations and offers a rich set of features. Therefore, the complete functionality goes beyond the scope of this thesis. For this reason we will focus on the parts that directly make use of the contributions of this work, namely, source code variability models, model slicing and variability bug detection.

The `undertaker` tool set is composed of several tools and modules that implement:

[1]The VAMOS project was partly supported by the German Research Foundation (DFG) under grant no. SCHR 603/7-1 and SFB/TR 89, see `vamos.informatik.uni-erlangen.de`

1. the extraction of CPP directives from source files.
2. the generation of CPP models in form of boolean formulas.
3. extraction and generation of boolean formulas for KCONFIG files.
4. calculation of slices from models.
5. generation of formulas for the detection of dead and undead variation points.
6. a backend that uses the SAT solver PicoSAT [2] in order to find a solution for boolean formulas.

The algorithmic part of the toolset—that basically deals with the generation of reasoning operations—is written in `C++`, the SAT solver is written in `C` and the extractors that deal with source files, KCONFIG files, and KBUILD files are written in `Python`. In this thesis we give special attention to the algorithmic parts of the toolset, which comprises the topics (2), (4) and (5). For this reason, these topics are the focus of our evaluation.

6.3 Experiment 1: Reasoning on Source Code Variability Models

Goal: The goal of this experiment is to evaluate the performance of our tool during the generation of the boolean formulas that represent source code variability models (as discussed in Chapter 3). Additionally, we also evaluate the performance of our tool with respect to find a solution for these formulas. We are interested in two time metrics: (1) the time that is necessary to generate boolean formulas representing the source code variability model and (2) the time required for finding solutions for these formulas using the SAT solver backend of our tool.

Setup: For this experiment we used the `undertaker` tool version `1.2` compiled with the `g++` version `4.4.3` using the options `-Wall -Wextra -O2`. The experiments were executed on a machine with an Intel quadcore 2.83 GHz processor. We used the Linux kernel version `3.6.0-rc1`. The list of 20,946 files that were analyzed comprise the source files of all architectures and subsystems (excluding only the subdirectories `tools`, `Documentation` and `scripts`) that contain at least one `CPP` directive for conditional compilation. In total, 111,997 conditional blocks were analyzed. We used the option `-j dead` of the `undertaker` tool to perform an analysis of dead and undead code blocks. A few functions of the tool were instrumented

[2] `http://fmv.jku.at/picosat/`

to measure and output the required runtimes. Specifically, we instrumented the functions responsible for building the checker function as explained in Definition 3 of Chapter 3. Also, we instrumented the function that builds the reasoning operation to detect dead features as described in Definition 6 of Chapter 5. This function converts the boolean formula representing the source file to CNF (conjunctive normal form) and passes it to the SAT solver that checks for the satisfiability of the formula.

Results: The data of this analysis contains $20{,}946$ entries representing the time required to build the boolean formulas for CPP-based source files, and $111{,}997$ entries representing the time needed to check whether a conditional block is dead or not. In order to reduce the impact of any undesired influence[3] on our results we ran the experiment 10 times. The data that we discuss here is the average of the data accumulated during these runs.

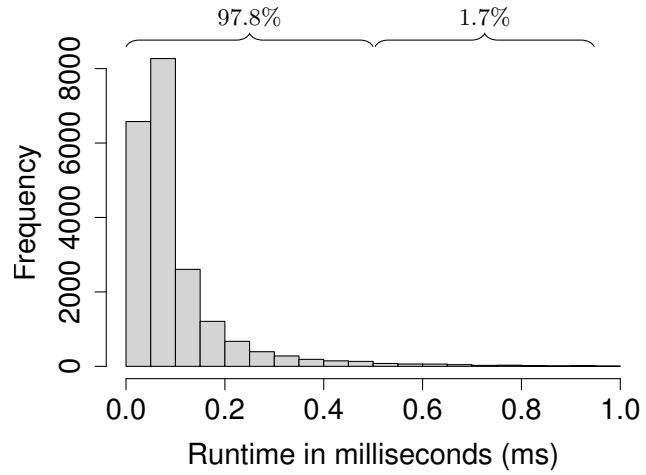


Figure 6.1: Runtime analysis for the generation of source-code boolean formulas. The histogram shows the frequencies of runtimes between **0** and **1ms**. The percentages shown on the top are relative to the total of entries.

First, we analyze the runtimes for generating the source-code boolean formulas. The data representing the time in milliseconds (ms) for the generation of the formulas has the following statistical properties: mean = 0.11, median = 0.07, minimal value = 0.03, maximum value = 9.99, and standard deviation $\sigma = 0.20$. In Figure 6.1 we show the distribution of the entries between 0ms and 1ms.

[3] In order to filter out any noise originate from the operating system or the cpu.

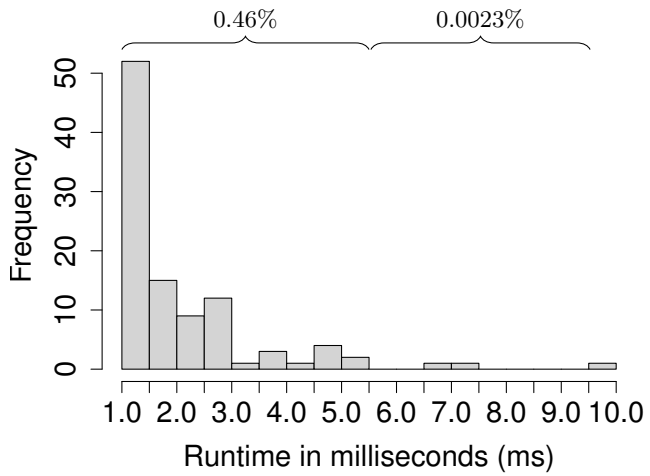


Figure 6.2: Runtime analysis for the generation of source-code boolean formulas. The histogram shows the frequencies of runtimes between **1** and **10ms**. The percentages shown on the top are relative to the total.

We can also see in this graph that for 97.8% of the source files a formula is generated in less than 0.5ms. The time required for 1.7% of the entries is in the range between 0.5ms and 1ms. Figure 6.2 shows the distribution of the remaining entries from 1ms up to 10ms. In this graph we see that for 0.46% of the files a runtime between 1ms and 5ms is required. For only 0.0023% of the files a runtime greater then 5ms is required, which corresponds to 5 source files.

The runtime, also in milliseconds (ms), for the detection of dead code blocks has the following statistical properties: mean = 0.3, median = 0.1, minimal value = 0.03, maximum value = 187.59, and standard deviation $\sigma = 1.78$. To give a better overview of these numbers we show three histograms in Figures 6.3, 6.4 and 6.5.

In these graphs the x-axis represents the runtime measured in milliseconds and the y-axis the frequency of these runtimes (that is, for how many blocks they were obtained). In Figure 6.3 we show the frequency of the runtimes below 1ms, which comprise $107{,}371$ entries representing 95.8% of all data. We can see that for 92.7% of all blocks the backend of our tool delivers an answer in less than 0.5ms. For 3.1% of the code blocks the runtime for the dead code analysis takes between 0.5ms and 1ms. In Figure 6.4 we show the frequencies for runtimes between 1ms and 10ms. The runtimes between 1ms and 5ms correspond to 3.3% of the total while the runtimes between 5ms and 10ms correspond to 0.5% of the total. In Figure 6.5 we see the distribution of the runtimes above 10ms up to 190ms. In this graph we see that 0.03% of the measurements are between 10ms and 100ms. Above this we have the runtimes of only 550 conditional blocks, which correspond to 0.005% of the total.

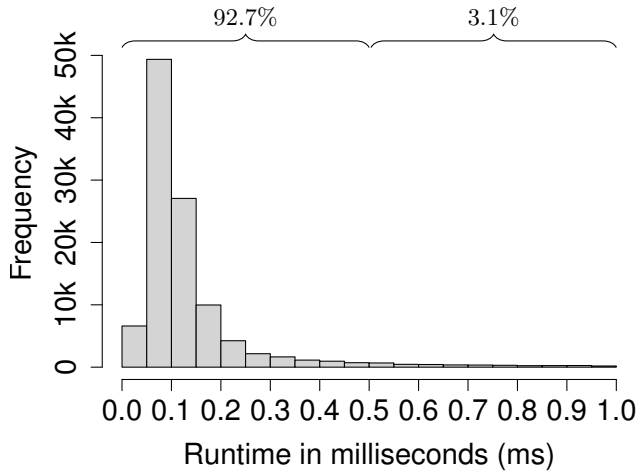


Figure 6.3: Runtime analysis for the search of dead and undead code blocks. This histogram shows the frequencies of runtimes between **0** and **1ms**. The percentages shown on the top are relative to the total of entries.

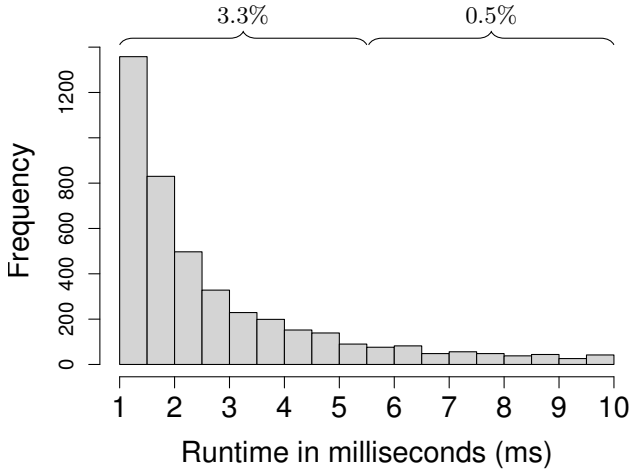


Figure 6.4: Runtime analysis for the search of dead and undead code blocks. This histogram shows the frequencies of runtimes between **1** and **10ms**. The percentages shown on the top are relative to the total of entries.

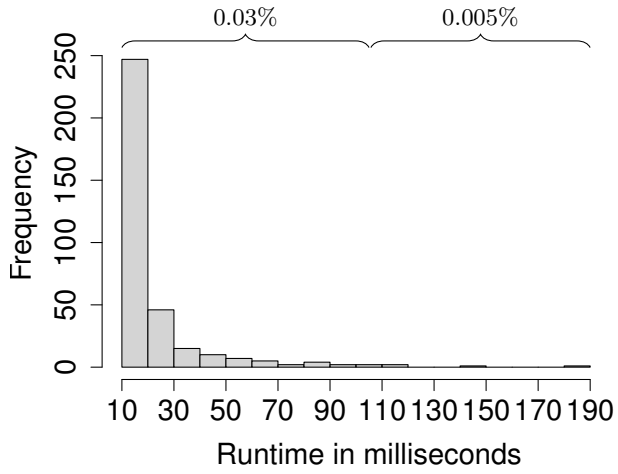


Figure 6.5: Runtime analysis for the search of dead and undead code blocks. This histogram shows the frequencies of runtimes between **10** and **190ms**. The percentages shown on the top are relative to the total of entries.

Analysis: Both parts of this experiment delivered good results. The average times 0.3ms and 0.11ms are acceptable for these kind of operations, especially if we take into consideration that both data sets have relatively low standard deviations of 0.2 and 1.78, respectively. The upper bounds at 10ms and 190ms show that even for the worst cases (which are the large source files with many conditional blocks), these operations can be performed rather quickly. Based on these results, we conclude that our approach for the generation and satisfiability check of source code variability models scales well to very large code bases. Moreover, based on the runtimes we presented, we are confident that introducing these checks in the build process would result in an acceptable amount of overhead. Moreover, as the runtime for the analysis explained in this experiment took a total of 7.1 minutes to analyze more than one hundred thousand code blocks, we are confident to claim that our approach is highly performant. Also, it can support both developers interested in a few files, as well as developers interested in analyzing the full code base.

6.4 Experiment 2: The benefits of applying the slicing algorithm

Goal: This experiment aims to analyze the time required to perform the slicing algorithm as introduced in Chapter 4. We also aim to point out the benefits of the slic-

ing algorithm. Recall that this algorithm calculates a sub-model from a given model. Moreover, it is only advantageous to use slicing when the returned sub-model is considerably smaller than the original model given as input. As a consequence, to evaluate the benefits of slicing we apply the algorithm to each and every KCONFIG feature and measure its runtime and the size (in terms of features) of the returned sub-model.

Setup: The configuration of this experiment is virtually identical to the setup used in the previous one. The only differences are the options passed to the `undertaker` tool. To generate the slices we used the option `-j interesting`. When using this option the tool expects two additional parameters. First, the name of a feature that will be used as the initial set of the algorithm. Second, the architecture must also be specified so that the corresponding KCONFIG model can be chosen. We have generated a slice for each feature of each architecture. The corresponding functions were instrumented to measure and output the time necessary to generate the slice and its size. The entries representing time are the average over 10 measurements. For the entries representing the size of slices only 1 run is necessary as they do not vary through different runs.

Results: The data for the runtime measurements of the slicing algorithm is shown in Table 6.1. This table is organized as follows. Each line represents the measurements on one architecture. In Linux, the first choice that has to be made for the configuration of a Linux kernel image is the architecture. Based on this choice the main KCONFIG file for the selected architecture is included in the configuration process. This main KCONFIG file contains directives for the inclusion of other KCONFIG files. For example, in the file `/arch/x86/Kconfig`, which is the main KCONFIG file for the architecture `x86`, we find directives like `source "net/Kconfig"`, which will include the KCONFIG definitions for the networking subsystem. In total there are 919 KCONFIG files, the main file for the `x86` architecture includes 30 other KCONFIG files. Naturally, these included files might include other files as well. The 27 architectures in Linux are defined in this manner. They are specified in the first column of Table 6.1. In the second column we see the number of features that each architecture contains. The following columns are the results of measurements applied on each of the features of the architecture specified in the first column. In the third column we show mean of time (in ms) for individually slicing the feature of an architecture. For example, the value 1.47ms for the `alpha` architecture was obtained by calculating the mean over 13,198 data points (that is the number of features of this architecture). The fourth column shows the upper bound (in ms) for the slicing algorithm in each architecture, that is, the longest time needed among all features of each architecture.

arch	Features (#)	Mean	Max	$\leq$ 10ms	$\leq$ 30ms	$>$ 30ms
alpha	13,198	1.47	13.54	97.95%	2.05%	0%
arm	14,703	22.93	57.53	50.76%	0%	49.24%
avr32	13,309	0.87	12.46	99.79%	0.21%	0%
blackfin	13,817	1.16	13.53	98.05%	1.95%	0%
c6x	13,060	0.78	12.45	99.98%	0.02%	0%
cris	7,932	0.48	10.47	99.96%	0.04%	0%
frv	13,186	0.62	11.36	99.98%	0.02%	0%
h8300	7,127	0.38	10.05	99.96%	0.04%	0%
hexagon	13,063	0.81	11.91	97.92%	2.07%	0%
ia64	13,451	1.35	14.1	99.37%	0.62%	0%
m32r	13,192	0.62	11.92	99.98%	0.02%	0%
m68k	13,298	1.08	12.86	99.59%	0.41%	0%
microblaze	13,113	0.83	12.35	99.98%	0.02%	0%
mips	13,762	14.89	38.55	47.5%	50.76%	1.74%
mn10300	13,250	0.75	11.7	99.98%	0.02%	0%
openrisc	13,060	0.85	12.62	99.98%	0.02%	0%
parisc	13,206	0.6	11.59	99.98%	0.02%	0%
powerpc	13,862	10.12	30.54	49.43%	50.56%	0.01%
s390	13,164	0.87	12.27	99.97%	0.03%	0%
score	13,056	0.56	11.12	99.98%	0.02%	0%
sh	13,533	8.54	26.25	52.19%	47.81%	0%
sparc	13,281	0.92	12.02	99.7%	0.3%	0%
tile	13,161	1.18	11.92	99.98%	0.02%	0%
um	85	0.08	0.33	100%	0%	0%
unicore32	13,265	0.84	12.78	99.98%	0.02%	0%
x86	13,709	3.76	17.63	96.47%	3.53%	0%
xtensa	13,163	0.62	11.53	99.98%	0.02%	0%

Table 6.1: Data representing the runtime to apply the slicing algorithm for each feature in every architecture of the Linux kernel.

The remaining three columns in Table 6.1 show the distribution of the runtimes of the slicing algorithm as percentages relative to the total number of features shown in the second column. The fifth column shows the percentage of features for which the slicing operation was performed in less than 10ms. We can see that for most of the architectures this range (from 0 to 10ms) contains most of the entries. The exceptions are the architectures `arm`, `mips`, `powerpc` and `sh`. The second range (from 10 to 30ms) is shown in the sixth column. For most of the architectures the percentages are very low for this range, except for the architectures `mips`, `powerpc` and `sh`. The last range (above 30ms) is shown in the last column of Table 6.1. Only three architectures have features by which the slicing algorithm requires mode than 30ms. They are `mips` and `powerpc` with very few features, and the `arm` architecture with 49.24 of its features requiring over 30ms to get a slice and the maximum at 57.53ms.

Table 6.2 summarizes the data regarding the sizes of the slices obtained for each feature of every architecture. Again, the first column shows the name of the architecture for which the results are presented in the remaining columns. The total number of features is omitted because the values are the same as shown in the second column of Table 6.1. We applied the slicing algorithm to each feature and measured the size in terms of features of the resulting slices. The mean over all features in a given architecture is shown in the second column. We see that for many architectures (e.g. `cris` and `frv`) we obtained mean values below 40. This means that even for models with more than 13,000 features the mean size of the slice is rather low. On the other hand, for the `arm` architecture, that has 17,703 features, the mean slice size is 776.66. The third column shows the upper bound for slice sizes, that is, the size of the largest size for each architecture. For most of the architectures the maximum number of features resulting from the slicing operation is about 600. The exceptions are the architectures `arm`, `mips`, `powerpc` and `sh`, with `arm` topping the others having the largest slice comprised of 2,004 features. The following columns show the percentages, relative to total number of features in an architecture, in four different ranges. In the fourth column we see the percentage of features that produce a slice containing up to 10 features. We can see that for all architectures at least 20% of their features produce slices with at most 10 features. In the fifth column we show the percentages for the range 10 to 100. This range covers most of the features for a large number of architectures. The sixth column shows the percentages for the range 100 to 500. This is the largest range for the architectures `ia64`, `tile` and `x86`. The seventh column shows the percentages of features that generate a slice with 500 or more features. Almost all architectures have at least a few features in this range, but for the architectures `arm`, `mips`, `powerpc` and `score` a substantial part of their features yields results in this range.

arch	Mean	Max	≤ 10	≤ 100	≤ 500	500
alpha	55.5	628	27.68%	52.12%	20.18%	0.02%
arm	776.66	2,004	21.77%	28.99%	0%	49.24%
avr32	56.36	637	30.01%	46.22%	23.75%	0.02%
blackfin	53.38	642	28.94%	51.62%	19.42%	0.02%
c6x	46.93	634	30.59%	51.25%	18.13%	0.02%
cris	32.98	554	34.62%	57.5%	7.84%	0.04%
frv	39.96	589	28.76%	60.23%	10.98%	0.02%
h8300	26.55	532	38.05%	55.16%	6.75%	0.04%
hexagon	50.53	609	30.76%	56.17%	10.98%	2.07%
ia64	89.74	721	26.12%	29.33%	44.35%	0.2%
m32r	40.28	609	29.81%	58.93%	11.23%	0.02%
m68k	61.83	651	25.72%	53.59%	20.67%	0.02%
microblaze	52.56	649	30.02%	50.91%	19.04%	0.02%
mips	554.73	1,486	23.37%	24.12%	0%	52.5%
mn10300	49.69	602	28.33%	52.23%	19.41%	0.02%
openrisc	52.59	658	30.01%	48.44%	21.52%	0.02%
parisc	38.92	604	30.65%	59.09%	10.24%	0.02%
powerpc	456.52	1,324	25.23%	24.19%	0%	50.57%
s390	61.5	666	23.97%	49.89%	26.12%	0.02%
score	36.12	581	30.68%	62.24%	7.05%	0.02%
sh	362.43	1,090	25.44%	26.75%	0%	47.81%
sparc	64.82	633	28.9%	41.58%	29.5%	0.02%
tile	92.15	614	21.6%	14.38%	63.99%	0.02%
um	5.71	29	82.35%	17.65%	0%	0%
unicore32	53.83	668	29.7%	47.76%	22.52%	0.02%
x86	199.5	856	24.79%	26.62%	45.41%	3.17%
xtensa	39.81	593	30.2%	58.42%	11.36%	0.02%

Table 6.2: Data representing the size of the slices generated for each feature in every architecture of the Linux kernel.

Analysis: This experiment focused on two metrics. The time required to generate a slice given a single feature of the model, and the number of features of the generated slice. The slice generation showed good performance with the longest runtime at 57ms, while for the vast majority of the slices the generation was performed in under 10ms.

Regarding the size of the generated slices the results are also very good. The majority of the slices, including all architectures, contain less than 100 features. The worst case is found in the `arm` architecture that has a feature that produces a slice with 2,004 features, considering that this architecture contains almost 15,000 features, even largest slice is relatively small, it represents a 88% size reduction, compared to total number of features of the model. We believe that the results regarding both aspects analyzed in this experiment, runtime for slice generation and slice size, substantiate our claims of high performance and scalability. The runtime for slice generation takes 57ms in the worst case, which is a very satisfactory number, specially when we consider that the worst slice provides a 88% model size reduction. Recall that in the previous experiment we analyzed more than one hundred thousand code blocks. If we wish to extend that analysis with KCONFIG constraints to perform multi-model reasoning we would have to call the SAT solver with a formula containing the constraints of at least 13,709 features (for the `x86` architecture). This is because in multi-model reasoning, without slicing, the constraints of all features of the architecture are required. The model size reduction provided by our slicing algorithm allows us not only to apply reasoning operations to very large code bases, but also to achieve a good level of performance and scalability.

6.5 Experiment 3: Finding Variability Bugs in the Linux kernel

Goal: The goal of this experiment is to verify the existence of variability bugs in the source code of the Linux kernel. In our first experiment we assessed the performance of our tool for the generation of source-code boolean formulas, and also, the performance of the SAT solver for finding a solution for these formulas. In the second experiment we measured the time required for generating model slices on the configuration models of the Linux kernel, moreover, we also analyzed the size of the produced slices. In this experiment we combine the functionalities that we have already individually evaluated to perform multi-model reasonings. We focus on detecting dead and undead code blocks combining two variability models: source code and KCONFIG. We also analyze the distribution of the detected issues among the different subsystems of the kernel.

Setup: We used once more the basic setup of the two previous experiments. For the detection of dead and undead conditional code blocks we used the option `-j dead`. Additionally, we set the option for loading models for all architectures. These models are also generated with a tool of the `undertaker` tool set. However, for the detection of variability bugs it is not possible to analyze the source files of specific architectures separately. This is because the only files that are architecture specific are those in the sub-directory `arch`, all other files can be used in different architectures and can contain architecture-specific conditional blocks. For this reason, in this experiment we analyze only blocks that are *globally* dead or undead. When our tool reports a code block to be dead/undead, it further marks the defect to be global if the code block either belongs to the `arch` subsystem or if it is dead/undead in all architectures. Recall the formula for detecting a dead variation point v introduced in the previous chapter (see Definition 6), this formula would be $\phi_d = \mathrm{SAT}(\varphi_c \wedge \varphi_K \wedge v)$ for detecting whether the block v is dead considering φ_c as the formula extracted from the source code and φ_K as the formula from the KCONFIG model. However, note that the KCONFIG formula is extracted per architecture. For this reason to detect a globally-dead feature we have to generate the formula ϕ_d for all architectures, where in each formula φ_K is the model of one architecture. The variation point can be considered dead only if ϕ_d is unsatisfiable for all architectures. This is the mechanism by which the `undertaker` tool detects globally dead/undead code blocks. In the following we analyze only globally dead/undead blocks, and not blocks that are problematic in one or a few architectures.

Results:

Table 6.3 summarizes the dead and undead blocks found by our tools. In the first column we indicate in which subsystem the issues were found. The following columns are used to classify the issues as follows:

- The columns marked with CD (code dead) and CU (code undead) represent the issues that were found using a formula that takes into consideration only the constraints found in the source file, that is, the CPP constraints. We deliberately left the KCONFIG constraints out.

- The columns marked with KD (KCONFIG dead) and KU (KCONFIG undead) represent the issues that were found using the same formula described above extended with the KCONFIG formulas for each architecture. Only if the formulas for all architectures are unsatisfiable the block is considered dead/undead.

subsystem	CD	CU	KD	KU	MD	MU	$\sum$
arch/	526	85	31	3	255	46	**946**
block/	2	0	0	0	0	0	**2**
crypto/	0	0	0	0	0	0	**0**
drivers/	1237	175	9	0	144	14	**1579**
firmware/	0	0	0	0	0	0	**0**
fs/	29	5	2	2	8	1	**47**
include/	21	28	5	3	2	2	**61**
init/	0	0	0	0	0	0	**0**
ipc/	0	0	0	0	0	0	**0**
kernel/	1	3	1	4	2	0	**11**
lib/	0	0	0	0	0	0	**0**
mm/	15	4	0	0	0	0	**19**
net/	13	0	0	0	4	0	**17**
security/	2	2	0	0	0	0	**4**
sound/	102	3	0	0	5	1	**111**
virt/	0	0	0	0	0	0	**0**
$\sum$	**1948**	**305**	**48**	**12**	**420**	**64**	**2797**

Table 6.3: Distribution of the detected variability bugs among the Linux kernel subsystem. CD: CODE DEAD, CU: CODE UNDEAD, KD: KCONFIG DEAD, KU: KCONFIG UNDEAD, MD: MISSING DEAD, MU: MISSING UNDEAD.

- The columns marked with MD (missing dead) and MU (missing undead) represent the issues that were found using the same formula described above extended with the missing symbols. Recall from Defintion 8 in Chapter 5, that the features used in the definition of conditional blocks must be set to false if they cannot be selected in the KCONFIG model.

In summary, we start with a basic formula containing only code constraints, and incrementally enhance it with the constraints from the KCONFIG models, and then with the missing symbols. Thereby, we are able to classify more precisely which constraints make the blocks dead or undead.

Table 6.3 shows the number of dead and undead blocks with the classification as explained above. Moreover, the last column shows the sub-totals per subsystem, and the last row shows the sub-totals of the different classifications. This way, the last cell in the last column shows the total number of issues found, that is, $2{,}797$ variability bugs.

The most common type among the reported issues are code dead ($1{,}948$) and code undead (305). This is a surprising result because, in theory, this should be the easiest type of variability bug to be detected without tool support, as all information required for this manual inspection is in a single source file. As we explained, this type is found with the less constrained formulas. Moreover, most of these issues are found in the subsystems `arch` ($526 + 85 = 611$) and `drivers` ($1237 + 175 = 1412$). The number of issues of the types KCONFIG dead (48) and KCONFIG undead (12) is considerably smaller, and again most of the issues are found in the subsystems `arch` ($31 + 3 = 34$) and `drivers` ($9 + 0 = 9$). The issues of the type missing are also mostly found in the subsystems `arch` ($255 + 46 = 301$) and `drivers` ($144 + 14 = 158$). There were a few subsystems where there were no issues found: `crypto`, `firmware`, `init`, `ipc`, `lib` and `virt`. In all other subsystems, 9 in total, dead and undead blocks were found.

Analysis: Most of the $2{,}797$ defects are found in `arch/` and `drivers/`. However it is important to note that these subsystems account for 75% of the `#ifdef`-blocks found in the code base. These subsystems being the most *problematic* corroborates the observations (e.g., [ECH+01]) that "most bugs can be found in driver code", which apparently holds for variability bugs as well.

In our opinion the number of bugs that our tools were able to detect is very impressive. Certainly, the current situation of the Linux kernel code base calls for tool support as provided by the `undertaker` tool. Moreover, the result from this experiment should be seen as an alert to other projects that employ a heterogeneous variability management. As we discussed in the beginning of this thesis, the Linux kernel development process benefits from a large number of developers and a strict

code review process, and even in this scenario the variability bugs that we identified could not be detected without the appropriate tool support.

Furthermore, the numbers revealed in this experiments support our hypothesis that the variability that is handled with different tools will eventually diverge over time, leading to variability bugs.

6.6 Experiment 4: Fixing Variability Bugs of the Linux Kernel

Goal: The goal of this experiment is to evaluate if the variability bugs, as detected in the previous experiment, are considered to be a problem by the actual developers and maintainers of the Linux kernel. To evaluate the quality of our findings, we analyzed the detected variability bugs, then we proposed a change and submitted the patch upstream to the responsible kernel maintainers.

Setup: The setup of this experiment differs from the previous ones. To evaluate the quality of our findings, we decided to inspect the variability bugs found by our tool chain and propose a solution to kernel developers. Our rationale is that the more our fixes are acknowledged and accepted the more relevant is the detection of variability bugs.

The first step is the variability bug analysis: We had to look up the source-code position for which the defect is reported and understand its particularities, which in many cases might also involve analyzing KCONFIG dependencies and further parts of the source code. This information was then used to develop a *patch* that fixes the defect. Based on the response to a submitted patch, we improved and resubmitted. Finally, we classified it and the defects it fixes in two categories: *accept (confirmed variability bug)* and *reject (confirmed rule violation)*. The latter means that the responsible developers consider the defect for some reason as *intended*. As a matter of fact, these defects are added into a local whitelist so that we can filter them out in future analysis.

Results: In the period of February to July 2010, a total of 123 patches were submitted. The submitted patches focus on the `arch/` and `driver/` subsystems and aim to fix 364 identified variability bugs. Table 6.4 summarizes the fixes we proposed to the Linux kernel developers. Note that normally one patch fixes several variability bugs. In this table the defects are classified in logic and symbolic. The first refers to the problems characterized as CD, CU, KD and KU. The second refers to the variability bugs classified as MD and MU. We make this distinction because the fixes for

the symbolic problems are generally easier to elaborate. In most cases, these problems are the result of a misspelling of a feature in the source code, therefore, we call it symbolic variability bug. On the other hand, the logic variability bugs are much harder to reason about, because the problem is caused due to a logical inconsistency among a set of features that form the variation point. In the first line we see the number of fixes we proposed, 150 for the logic variability bugs and 214 for symbolic variability bugs. Out of these proposals, 154 were confirmed as actual bugs by the developers and maintainers. Also, 109 were not confirmed as bugs, that is, according to the developers the dead and undead blocks are desired to be kept like this. These cases we classify as confirmed rule violation. For 101 fixes we received no answer. The numbers in brackets shown in Table 6.4 refer to the variability bugs that we see as critical. These are variability bugs that actually trigger an undesired effect on the binary code. That is, variability bugs that trigger software bugs.

Table 6.5 also classifies the submitted patches as *critical* and *noncritical*. Critical patches fix software bugs that are triggered by variability bugs. Noncritical patches remove *cruft*, that is, irrelevant or unwanted code. We can see in Table 6.5 that the responsible developers consider them as worth fixing: 16 out of 17 (94%) of our critical patches have been answered; 9 have already been merged into Linus Torvalds' master git tree for Linux 2.6.36. The majority of our patches fixes defects that affect the source code only. However, even for these noncritical patches 57 out of 106 (54%) have already reached *acknowledged* state or better. These patches clean up the kernel sources by removing 5,129 lines of configurability-related dead code and superfluous `#ifdef` statements. We consider this as a strong indicator that the Linux community is aware of the negative effects of configurability on the source-code quality and welcomes attempts to improve the situation.

	Logic	Symbolic	$\sum$
fix proposed	150 (1)	214 (22)	364 (23)
confirmed variability bug	38 (1)	116 (20)	154 (21)
confirmed rule violation	88 (0)	21 (2)	109 (2)
pending	24 (0)	77 (0)	101 (0)

Table 6.4: Fixes for variability bugs proposed to the Linux kernel developers.

Analysis: In general, we see that our patches are well received: 87 out of 123 (71%) have been answered; more than 70% of them within less than one day, some even within minutes (see Figure 6.7). We take this as indication that many of our patches are easy to verify and in fact appreciated.

Figure 6.6 depicts the impact of our work on a larger scale. To build this figure, we ran our tool on previous kernel versions and calculated the number of variability bugs that were *fixed* and *introduced* with each release. Most of our patches entered the mainline kernel tree during the merge window of version 2.6.36. Given that the patch submissions of two students have already made such a measurable impact, we expect that a consequent application of our approach, ideally directly by developers that work on new or existing code, could significantly reduce the problem of variability bugs in the Linux kernel.

patch status	critical	noncritical	$\sum$
submitted	17	106	123
unanswered	1	35	36
ruleviolation	1	14	15
acknowledged	1	14	15
accepted	5	3	8
mainline	9	40	49

Table 6.5: *Critical* patches do have an effect on the resulting binaries (kernel and runtime-loadable modules). Noncritical patches remove text from the source code only.

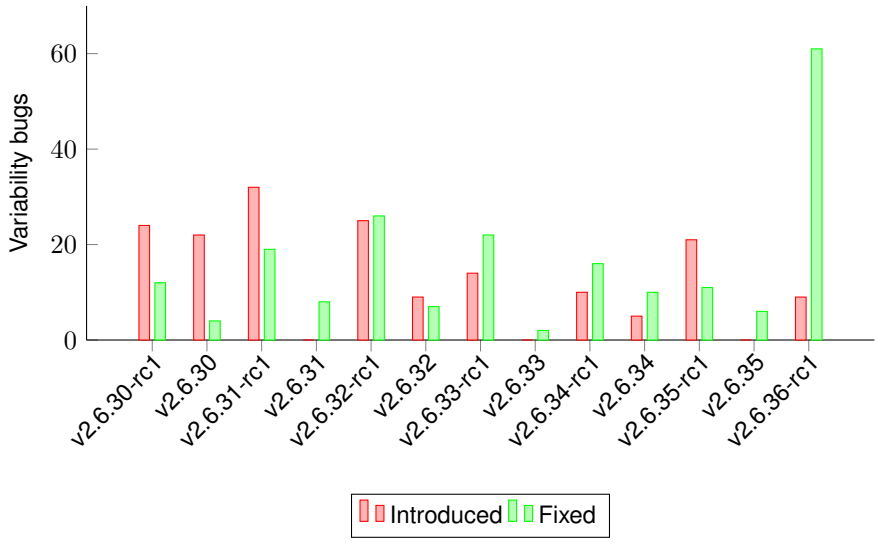


Figure 6.6: Impact of our patches on Linux releases.

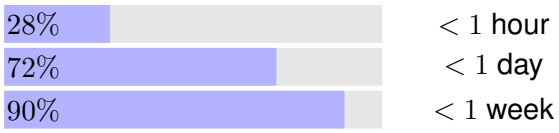


Figure 6.7: Response time of 87 answered patches

6.7 Summary

In this chapter we evaluated the techniques and tooling developed in this thesis. We focused on three evaluation criteria, namely, performance, accuracy and relevance. Our rationale was that if these three criteria are met, it is a very good indication that our tools and techniques can deal with the challenges of large-scale projects, and deliver results that are not only correct, but also trigger the interest of the project maintainers.

In our first experiment we evaluated the performance of our algorithm for generating boolean formulas from CPP-annotated source files. The average time for this operation, over all CPP blocks found in the Linux kernel code base, was 0.11ms, and the worst case 9.99ms. Moreover, we used these formulas to perform reasoning operations for the detection of dead and undead code blocks. For these operations our SAT backend required in average 0.3ms to provide an answer, and 187.58ms in the worst case. We believe that the results of this experiment help us to support the claim of having met the criterion *performance*.

In the second experiment we analyzed the benefits of applying the slicing algorithm to the configuration models of the Linux kernel. We applied our slicing algorithm to every feature of all 27 architectures of the Linux kernel. We showed that for the vast majority of all features the algorithm generates the slice in under 10ms, and in the worst case the algorithm needed 57.53ms. Regarding the size of the generated slice, the majority of slices contains less than 100 features, the largest slice contained 2,004 features. Considering that state-of-the-art techniques for reasoning in feature models do not use similar optimizations as our slicing algorithm, even in our worst case (a slice with 2,004 features from a model with 14,703 feaures) we have a model size reduction of 88%. The results of this experiment also help us to support the claim of having met the criterion *performance*.

The third experiment aimed at revealing variability bugs in the Linux kernel code base. Our tool revealed a total of 2,797 variability bugs. We described the distribution of these issues among the different subsystems. We detected that most of the problems are in the `arch` and `drivers` subsystem, which correlates with previous work that claims that most of the bugs in operating system are found in

the `drivers` subsystem. Moreover, our algorithm for the detection of variability bugs is correct by construction. This means that if our tool reports that a code block is either dead or undead, we can recheck it with the standard tools (e.g. CPP) and it is always correct. However, it does not mean that the Linux maintainers have the urge to fix all the $2{,}797$ issues we detected. We learned that in many cases it is desired to leave dead and undead code blocks in the code base, for reasons like: documentation, incomplete feature implementation, and so on. The results of this experiment helped us to support the claim of that we met the criteria *performance* (an analysis of the full code base takes around 40 minutes) and *accuracy* (the detected variability bugs are always correct).

In the fourth experiment we evaluated whether the variability bugs found by our tool are of interest for the Linux kernel maintainers. To this end, we submitted a total of 123 patches to fix a total of 364 variability bugs. For 154 variability bugs our fixes were accepted, and for 109 the developers confirmed the issue but decided to keep the code as it was. Moreover, for 90% of our patches we received an answer in less than one day. We believe that these numbers are a good indication that the Linux developers have interest in fixing the variability bugs that can be automatically found with our tools. The results of this experiment help us to support the claim of having met the criterion *relevance*.

7

Conclusions and future work

Software product lines are the result of a complex development strategy. The goal to build a single software repository that can be tailored to meet the requirements of a broad variety of products in the same domain imposes many challenges. Among these challenges, to deal with the commonality and variability that form the spectrum of products is one of the key success factors of the development. Variability management is the discipline responsible for coping with these challenges. Basically, variability management can be performed in two ways: In homogeneous variability management a common language and tooling is used to define the whole variability in the software product line. In heterogeneous variability management different languages and tools are employed to specify the variability in different artifacts.

In general terms this thesis addressed problems of heterogenous variability management. We developed techniques to ensure that the variability described in different artifacts and possibly in different formats are consistent. We have seen that in large projects, and discussed in detail in the context of the Linux kernel, the definition of variation points spans over several artifacts and is managed by independent tools. On this basis, we raised the hypothesis that, without the appropriate tool support, inconsistencies among these independently-managed variation points will inevitably arise.

In order to verify this hypothesis we studied in detail how variability is described and implemented in the Linux kernel. As a result, we identified how variation points in different artifacts, that relate to the same features of the system, are defined and

constrained. In the case of the Linux kernel, we identified that variation points defined for the configuration tool are further used to define variation points in source files, in rules of the build system, and so on. We discovered that the definition of these variation points by different and totally independent tools is an error-prone process regarding variability consistency.

To address these issues we followed a strategy that is commonly applied in the context of software product lines: We converted the information from variability models into propositional logic—the common format—in order to perform reasoning operations with the help of SAT solvers. During the design of our solution we could benefit from several works that had previously studied the manipulation of variability in propositional logic. However, we identified the need to adapt and extend such techniques to the context of system software. We used the Linux kernel, as our system software platform, for our case studies. Our analysis to compare the variability management techniques of the Linux kernel with the ones advocated by the software product line community revealed the following observations:

- Several of the techniques to manage variability in the Linux kernel are very similar to what is used by the SPL community. For example, Linux employs a clear separation of problem space and solution space. The problem space is modeled with help of the KCONFIG tool, which has many similarities with feature modeling tools.

- For the mapping between problem space and solution space the Linux kernel employs its build system KBUILD. This tool processes the user selection generated during the configuration phase to select artifacts that need to be preprocessed, compiled and linked to form the final kernel image.

- For the realization of variability in the source code Linux employs the C preprocessor. Although this technique has been criticized many times, it is still the technique of choice of many practitioners and researchers for the implementation of variability in software product lines.

Following the analysis of the processes and tools described above we initially concluded that the Linux kernel is an appropriate platform for the study of software product line engineering. However, the more we studied the platform, implemented and evaluated SPL-related techniques on its basis, we finally concluded that the Linux kernel is in fact a software product line. Since we published this conclusion [SSSPS07], many studies have followed this line of research [PS08, Lot09, ZK10, SLB+10].

To bridge the gap between reasoning techniques commonly used in SPLs and the variability management approach of the Linux kernel we decided to use propo-

sitional logic as the abstraction means and SAT solvers as the logic engine. Moreover, in oder to achieve our goal, that is, to design, develop and test an approach that automatically identifies variability inconsistencies among all variability models, we had to address the following individual problems: (1) convert the variability described as CPP directives into propositional logic, (2) combine the formulas from different variability models and deal with model size explosion and (3) design the reasoning problems to reveal variability inconsistencies such as dead and undead variation points. The solutions of these problems are the building blocks that form our approach for the automatic, precise and high-performance detection of variability bugs.

As a result, we developed a lightweight technique to convert the CPP directives into propositional formulas. Although there exists previous work to convert CPP directives to many formats, we focused on the directives that are used to implement variability. Moreover, we leveraged in our approach the guidelines that rule how the directives must be used to describe variability. By doing so we were able to design a very lightweight approach that builds a propositional formula that mimics the semantics of the CPP. The algorithm that builds the formulas scales linearly with respect to the number of conditional blocks in the file being processed. Evaluating this approach using the Linux kernel code base we identified that the generation of formulas representing a whole file takes at most 9.99 milliseconds, and in average 0.11 milliseconds. We also analyzed the runtimes for the detection of inconsistencies in these formulas using a SAT solver, we identified an upper bound of 187.49 milliseconds and an average of 0.3 milliseconds. We consider these numbers to be very acceptable, and we believe that this performance supports our claim that our approach is lightweight, and, as a consequence, shows high performance.

To address the problem of combining the formulas of many variability models but still to cope with the model size we introduced the concept of model slicing. Basically, model slicing allows us to use specific parts of the formulas that represent variability models when performing specific reasonings. To use the algorithm the variability models have to be described as propositional formulas and these formulas have to follow a given format. That is, each variation point must imply its presence condition. The models are then just the conjunction of these implications. The slicing algorithm gets as input a variability model, in the format just described, and a set of variation points of interest. On this basis the algorithm selects the other variation points of the model that might influence the ones given as input. This way, if we want to reason about a conditional block in a source file that uses only two features (variation points), we give the whole variability model and these two features as input to the algorithm. Then we get as output the set of variation points that has to be included in the reasoning operation that uses the two initial features

of interest. Clearly, it is advantageous to use slicing when the output is considerably smaller than the original model. Moreover, it is easy to see that the best case is an output equal to the set of variation points given as inputs, and the worst case is the output equal to the original model. To investigate these issues we analyzed the performance of the slicing algorithm for all architectures of the Linux kernel. In total there are 27 architectures containing up to $14{,}000$ features. As the resulting slice size heavily depends on the inter-constraints of the model, the results among the different architectures show a high degree of variation. However, even for the worst cases we obtained the following: the largest time required to generate a slice for an individual feature was 57.53 milliseconds and the largest slice contained $2{,}004$ features. Consider that even for the largest slice from a model with over $14{,}000$ features, we obtained a 88% model size reduction. Based on these figures we are confident to affirm that slicing is very appropriate to be applied to the variability models of the Linux kernel, as it greatly helped us to reduce model size, and, as a result, improved the performance of our reasoning operations.

Using the two techniques described above we were able to devise reasoning operations to reveal inconsistency problems. We designed a set of operations to reveal the core problems when dealing with multiple variability models that share a common set of features. For example, the features defined in the configuration model are further used and combined to define constraints in the build system, or as conditionals in source files. We targeted the core problem in this context. That is, the automatic identification of variation points that are only seemingly variable. In other words, variation points that are expected to be variable but cannot be. Examples are variation points that are unconditionally always selected or unconditionally always deselected. We call the former a dead variation point and the latter an undead variation point.

Our reasoning operations allow us to perform multi-model reasoning by combining variability models of different types and variation points of interest. By doing so, and with the help of a SAT solver, we are able to reason about specific variation points. For instance, checking if the constraints of all models together allows the selection or deselection of a given variability point is at all possible.

To evaluate our techniques for automatic detection of variability bugs we have implemented the techniques described in this thesis in the `undertaker` tool. We applied it to the Linux kernel code base to detect two specific kinds of variability bugs. Namely, dead and undead conditionally compiled code blocks in source files. Our experiment revealed a great number of such variability bugs in many subsystems of the Linux kernel. In total, we have found $2{,}797$ dead and undead code blocks in the Linux kernel version `2.6.30-rc1`. To assess the relevance of these issues we addressed the Linux kernel community. We submitted 123 patches that corrected a

total of 364 variability bugs. From these patches, 72 were acknowledged as correct, and 49 made already their way up to the mainline (the official repository). Moreover, 72% of the emails with our patches were responded in less than one day, and 90% of the total was responded in less than a week. We see this as a good indication of the relevance of our reports, and, as a consequence, of the general importance of fixing variability bugs.

Taking these results into consideration we revisit the initial hypotheses of this thesis:

- We confirmed that it is possible to perform reasoning operations with boolean formulas and SAT solvers in real-world, large-scale software projects. Our techniques enable both incremental analysis where the detection of variability bugs is performed only on a few files of the system. This strategy is good to support the development of new features in order to avoid the introduction of new variability bugs. Also, our tools can be used to perform a complete analysis including all files of the project. We showed that even for Linux, which contains more than 20,000 files, an analysis can be performed in less than 30 minutes.

- We confirmed that without appropriate tool support variability bugs will inevitably appear in the code base. Our findings shows that even in a large, important and well-developed project like the Linux kernel the code base contains an substantial number of variability bugs.

- We confirmed the importance of efficient methods to detect variability bugs by providing fixes to hundreds of dead and undead conditional blocks. The feedback from the community was very easy to interpret. Not only the majority of our fixes was accepted, but the answers were given very fast and several developers demonstrated great appreciation to our contributions.

7.1 Future Work

In this work we presented several techniques that can be combined for the automatic detection of variability bugs. In the following we discuss research opportunities that can both reuse these techniques or extend them:

- In this work we focused heavily on the Linux kernel. Although it is a very relevant project that is an almost ideal testbed for the development of variability management techniques, we believe that our techniques could also be applied to other projects. The easiest way to do so would be to use systems that have a similar structure. That is, systems that use the same tools for variability

management, for example, the combination of KBUILD and the C preprocessor. We know some projects that have exactly this setup: the `Fiasco` microkernel, the `Coreboot` firmware, the `uClinux` micro-controller project, among others. However, it is also possible to apply our techniques to projects with a different management process. In this case, our tools have to be extended to extract the variability that is described in formats that it cannot handle yet.

- In this thesis we performed basically two types of reasoning operations. Namely, operations that take into consideration the constraints of a single model, for example, dead features in source code that were found without the KCONFIG constraints. Also, multi-model operations, where the constraints from two models were used to reveal dead code blocks. However, our reasoning framework can easily include information from more variability models. The natural extension of our work in the context of the Linux kernel would be to include the KBUILD variability model into the reasoning process. We believe that with this information our tool could find even more variability bugs.

- The reasoning framework that we introduced in this thesis can be used to reason about the *correctness* of arbitrarily combined variation points. The user of our tool has just to specify the set of variation points that are of interest, and our tool automatically builds the required formula that is used to check if the combination of variation points is valid. This functionality can be used to support variability-aware analysis tools. The two prominent examples of this category are `TypeChef`, which is a tool for type checking C code, and `SuperC`, a tool for parsing C code. These tools implement algorithms that take into consideration CPP information during parsing and type checking, which makes them variability aware. We believe that the `undertaker` tool could be used as a back-end for these tools to provide the correct presence conditions (even from many models) to the variation points that these tools manipulate.

- The evaluation of the techniques developed in this thesis were carried out as a quantitative assessment. We mostly focused on measuring the efficiency of our techniques, and also, assessing the quantity of variability bugs that could be found. However, there are still many questions that can be investigated: *How are variability bugs introduced in the code base? How long does it take for the community to find and fix them? How serious are the bugs that have an impact in the object code?* Answering these type of questions would help to provide a deeper qualitative evaluation of our techniques.

Bibliography

[ACLF11] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Slicing feature models. In *ASE*, pages 424–427, 2011.

[AK09] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84, July/August 2009. Refereed Column.

[Bat05] Don S. Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th Software Product Line Conference (SPLC '05)*, pages 7–20, 2005.

[Bax02] Ira D. Baxter. DMS: program transformations for practical scalable software evolution. In *Proceedings of the 5th International Workshop on Principles of Software Evolution (IWPSE'02)*, pages 48–51, New York, NY, USA, 2002. ACM Press.

[BFG+01] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, Henk Obbink, and Klaus Pohl. Variability issues in software product lines. In *Proceedings of the 4th International Workshop on Software Product-Family Engineering*, Heidelberg, Germany, 2001. Springer-Verlag.

[BKPS04] Günter Böckle, Peter Knauber, Klaus Pohl, and Klaus Schmid. *Software-Produktlinien: Methoden, Einführung und Praxis*. dpunkt.verlag GmbH, Heidelberg, 2004.

[BM01] Ira D. Baxter and Michael Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE '01)*, page 281, Washington, DC, USA, 2001. IEEE Computer Society Press.

[BN00] Greg J Badros and David Notkin. A framework for preprocessor-aware c source code analyses. *Software: Practice and Experience*, 30(8):907–924, 2000.

[BPSP04] Danilo Beuche, Holger Papajewski, and Wolfgang Schröder-Preikschat. Variability management with feature models. *Science of Computer Programming*, 53(3):333–352, 2004.

[BRCT05] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAISE '05)*, volume 3520, pages 491–503, Heidelberg, Germany, 2005. Springer-Verlag.

[BSL+10a] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wąsowski. Feature-to-code mapping in two large product lines. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond*, volume 6287 of *Lecture Notes in Computer Science*, pages 498–499. Springer Berlin / Heidelberg, 2010.

[BSL+10b] Thorsten Berger, Steven She, Rafael Lotufo, Andrezj Wasowski, and Krzysztof Czarnecki. Variability modeling in the real: A perspective from the operating systems domain. In *25th IEEE/ACM International Conference on Automated Software Engineering*, 09/2010 2010.

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 35(6), 2010.

[BSTRC06] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. A first step towards a framework for the automated analysis of feature models. In *Managing Variability for Software Product Lines: Working With Variability Mechanisms*, 2006.

[BSTRC07] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. FAMA: Tooling a framework for the automated analysis of feature models. In *Proceedings of the 1th International Workshop on Variability Modelling of Software-intensive Systems (VAMOS '07)*, 2007.

[CE00] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications*. Addison-Wesley, May 2000.

[CHE04] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged configuration using feature models. In *Proceedings of the 3th Software Product Line Conference (SPLC '04)*, pages 266–283, Heidelberg, Germany, 2004. Springer-Verlag.

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged configuration through specialization and multilevel configuration of

feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[CHH09] Andreas Classen, Arnaud Hubaux, and Patrick Heymans. A formal semantics for multi-level staged configuration. In *Proceedings of the 3th International Workshop on Variability Modelling of Software-intensive Systems (VAMOS '09)*, pages 51–60, 2009.

[CP06] Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE '06)*, pages 211–220, New York, NY, USA, 2006. ACM Press.

[cpp] The C Preprocessor – GNU Project, author = Richard M. Stallman and Zachary Weinberg note = `http://gcc.gnu.org/onlinedocs/cpp/`, key = cppsite, url = http://gcc.gnu.org/onlinedocs/cpp/, category = c,.

[CW07] Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. In *Proceedings of the 11th Software Product Line Conference (SPLC '07)*, pages 23–34. IEEE Computer Society Press, Sept. 2007.

[Dij68] Edsger Wybe Dijkstra. The structure of the THE-multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[DTSPLa] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. A robust approach for variability extraction from the linux build system. To appear.

[DTSPLb] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Understanding Linux feature distribution.

[EBN02] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002.

[ECH+01] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72, New York, NY, USA, 2001. ACM Press.

[GA01] Cristina Gacek and Michalis Anastasopoules. Implementing product line variabilities. In *Proceedings of the 2001 Symposium on Software Reusability (SSR '01)*, pages 109–117. ACM Press, 2001.

[Gar05] Alejandra Garrido. *Program refactoring in the presence of preprocessor directives*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2005. Adviser-Johnson, Ralph.

[HEB+10] Wanja Hofer, Christoph Elsner, Frank Blendinger, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Leviathan: SPL support on filesystem level. 14th International Software Product Line Conference (SPLC '10), October 2010. Poster.

[HFC76] Arie Nicolaas Habermann, Lawrence Flon, and Lee W. Cooprider. Modularization and hierarchy in a family of operating systems. *Communications of the ACM*, 19(5):266–272, 1976.

[HMDL00] Ying Hu, Ettore Merlo, Michel Dagenais, and Bruno Lagüe. C/C++ conditional compilation analysis using symbolic execution. In *Proceedings of the 16th IEEE International Conference on Software Maintainance (ICSM'00)*, page 196, Washington, DC, USA, 2000. IEEE Computer Society Press.

[Int05] International Organization for Standardization. *ISO/IEC 9899:TC2: Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, 2005.

[Jan10] Mikolas Janota. *SAT Solving in Interactive Configuration*. PhD thesis, University College Dublin, 2010.

[JB02] Michel Jaring and Jan Bosch. Representing variability in software product lines: A case study. In *Proceedings of the 2nd Software Product Line Conference (SPLC '02)*, pages 15–36, London, UK, 2002. Springer-Verlag.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[JK07] Mikolás Janota and Joseph Kiniry. Reasoning about feature models in higher-order logic. In *Proceedings of the 11th Software Product Line Conference (SPLC '07)*, pages 13–22. IEEE Computer Society Press, 2007.

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th International Conference*

on Software Engineering (ICSE '08), pages 311–320, New York, NY, USA, 2008. ACM Press.

[Käs10] Christian Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, University of Magdeburg, May 2010. Logos Verlag Berlin, isbn 978-3-8325-2527-9.

[KAT+09] Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don Batory. Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In *Proceedings of the 47th International Conference Objects, Models, Components, Patterns (TOOLS EUROPE)*, volume 33 of *Lecture Notes in Business Information Processing*, pages 175–194. Springer Berlin Heidelberg, June 2009.

[KATS11] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. Type checking annotation-based product lines. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2011. to appear; submitted 8 Jun 2010, accepted 4 Jan 2011.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, November 1990.

[ker05] The linux 2.5 kernel summit. http://lwn.net/2001/features/KernelSummit/, 2005.

[KGO11] Christian Kästner, Paolo G. Giarrusso, and Klaus Ostermann. Partial preprocessing C code for variability analysis. In *Proceedings of the Fifth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 137–140, New York, NY, USA, January 2011. ACM Press. Acceptance rate: 55 % (21/38).

[KS94] Maren Krone and Gregor Snelting. On the inference of configuration structures from source code. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 49–57, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[LAL+10] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE '10)*, New York, NY, USA, 2010. ACM Press.

[Lat04] Mario Latendresse. Rewrite systems for symbolic evaluation of c-like preprocessing. In *CSMR '04: Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, page 165, Washington, DC, USA, 2004. IEEE Computer Society.

[Loh09] Daniel Lohmann. *Aspect Awareness in the Development of Configurable System Software*. PhD thesis, Friedrich-Alexander University Erlangen-Nuremberg, 2009.

[Lot09] Rafael Lotufo. On the complexity of maintaining the linux kernel configuration. Technical report, Electrical and Computer Engineering University of Waterloo, 2009.

[LSB+10] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. Evolution of the linux kernel variability model. In *SPLC*, pages 136–150, 2010.

[Men09] Marcilio Mendonça. *Efficient Reasoning Techniques for Large Scale Feature Models*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 2009.

[mF95] Jean marie Favre. The cpp paradox. In *European Workshop on Software Maintenance*, 1995.

[MHP+07] Andreas Metzger, Patrick Heymans, Klaus Pohl, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the documentation of variability in software product lines. In *Proceedings of the 15th IEEE International Conference on Requirements Engineering (RE'07)*, pages 243–253, Washington, DC, USA, 2007. IEEE Computer Society.

[MO04] Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. In *Proceedings of ACM SIGSOFT '04 / FSE-12*, November 2004.

[Mor05] Andrew Morton. The linux kernel development process. http://aycinena.com/index2/index3/archive/andrew%20mo rton%20&%20the%20linux%20kernel.html, 2005.

[MWC09] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. Sat-based analysis of feature models is easy. In *Proceedings of the 13th Software Product Line Conference (SPLC '09)*, pages 231–240, Pittsburgh, PA, USA, 2009. Carnegie Mellon University. ISBN 978-0-9786956-2-0.

[MWCC08] Marcílio Mendonça, Andrzej Wasowski, Krzysztof Czarnecki, and Donald D. Cowan. Efficient compilation techniques for large scale feature models. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE '08)*, pages 13–22, New York, NY, USA, 2008. ACM Press.

[NC01] Linda Northrop and Paul Clements. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[NH11] Sarah Nadi and Richard C. Holt. Make it or break it: Mining anomalies from linux kbuild. pages 315–324, 2011.

[NH12] Sarah Nadi and Richard C. Holt. Mining Kbuild to detect variability anomalies in Linux. Washington, DC, USA, 2012. IEEE Computer Society Press. To appear.

[Par76] David Lorge Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *European Conference on Object-Oriented Programming*, pages 419–443, 1997.

[PS08] Hendrik Post and Carsten Sinz. Configuration lifting: Verification meets software configuration. In *Proceedings of the 23th IEEE International Conference on Automated Software Engineering (ASE '08)*, pages 347–350, 2008.

[pvs] Pure-systems GmbH – pure::variants. `http://www.pure-systems.com/`.

[Ray] Eric S. Raymond. The cml2 resources page. http://www.catb.org/ esr/cml2/.

[Ref09] Jacob G. Refstrup. Adapting to change: Architecture, processes and tools: A closer look at hp's experience in evolving the owen software product line. Keynote at the Software Product Line Conference (SPLC '09), 2009.

[SB10] Steven She and Thorsten Berger. Formal semantics of the Kconfig language. Technical note, University of Waterloo, 2010.

[SC92] Henry Spencer and Gehoff Collyer. #ifdef considered harmful, or portability experience with C News. In *Proceedings of the 1992 USENIX Annual Technical Conference*, Berkeley, CA, USA, June 1992. USENIX Association.

[SDNB07] Marco Sinnema, Sybren Deelstra, Jos Nijhuis, and Jan Bosch. COVAMOF: A framework for modeling variability in software product families. In *Proceedings of the 11th Software Product Line Conference (SPLC '07)*, Heidelberg, Germany, 2007. Springer-Verlag.

[SJ04] Klaus Schmid and Isabel John. A customizable approach to full lifecycle variability management. *Sci. Comput. Program.*, 53(3):259–284, 2004.

[SLB+10] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. The variability model of the linux kernel. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VAMOS '10)*, Linz, Austria, January 2010.

[SLB+11] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. Reverse engineering feature models. In *ICSE*, pages 461–470, 2011.

[SSP08] Julio Sincero and Wolfgang Schröder-Preikschat. The linux kernel configurator as a feature modeling tool. In Steffen Thiel and Klaus Pohl, editors, *Proceedings of the 12th Software Product Line Conference (SPLC '08), Second Volume*, pages 257–260. Lero Int. Science Centre, University of Limerick, Ireland, 2008.

[SSSPS07] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Is the linux kernel a software product line? In Frank van der Linden and Björn Lundell, editors, *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*, Kyoto, Japan, 2007.

[STB+04] Mirjam Steger, Christian Tischer, Birgit Boss, Andreas Müller, Oliver Pertler, Wolfgang Stolz, and Stefan Ferber. Introducing pla at bosch gasoline systems: Experiences and practices. In *Proceedings of the 3th Software Product Line Conference (SPLC '04)*, pages 34–50, Heidelberg, Germany, 2004. Springer-Verlag.

[STE+10] Julio Sincero, Reinhard Tartler, Christoph Egger, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Facing the linux 8000 feature nightmare. ACM SIGOPS/EuroSys European Conference on Computer Systems 2009 (EuroSys '09), April 2010. Talk & Poster.

[STL10] Julio Sincero, Reinhard Tartler, and Daniel Lohmann. An algorithm for quantifying the program variability induced by conditional compilation. Technical Report CS-2010-02, University of Erlangen, Dept. of Computer Science, January 2010.

[STLSP10] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Efficient extraction and analysis of preprocessor-based variability. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE '10)*, New York, NY, USA, 2010. ACM Press.

[Sva00] Mikael Svahnberg. *Variability in Evolving Software Product Lines*. PhD thesis, Blekinge Institute of Technology, 2000.

[SvGB06] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Software - Practice and Experience*, 35(8):705–754, 2006.

[TBK09] Thomas Thum, Don Batory, and Christian Kästner. Reasoning about edits to feature models. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, pages 254–264, Washington, DC, USA, 2009. IEEE Computer Society Press.

[TBKC07] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE '07)*, pages 95–104, New York, NY, USA, 2007. ACM.

[TLSSP11] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2011 (EuroSys '11)*. ACM Press, April 2011.

[TSD+12] Reinhard Tartler, Julio Sincero, Christian Dietrich, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Revealing and repairing configuration inconsistencies in large-scale system software. *International Journal on Software Tools for Technology Transfer (STTT)*, 2012.

[TSSPL09] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Dead or alive: Finding zombie features in the Linux kernel. In *Proceedings of the 1st Workshop on Feature-Oriented Software Development (FOSD '09)*, pages 81–86. ACM Press, 2009.

[vG06] Jilles van Gurp. OSS Product Family Engineering. In *1st International Workshop on Open Source Software and Product Lines (SPLC 2006)*, 2006.

[ZK10] Christoph Zengler and Wolfgang Küchlin. Encoding the Linux kernel configuration in propositional logic. In Lothar Hotz and Alois Haselböck, editors, *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration 2010*, pages 51–56, 2010.