Variability Anomalies in Software Product Lines

by

Sarah Nadi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctoral of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2014

Author’s Declaration

Statement of Contributions

I would like to acknowledge the names of my co-authors who contributed to the research described in this dissertation. These include:

- Dr. Richard C. Holt
- Dr. Reinhard Tartler
- Christian Dietrich
- Dr. Daniel Lohmann
- Dr. Thorsten Berger
- Dr. Christian Kästner
- Dr. Krzysztof Czarnecki

Abstract

Software Product Lines (SPLs) allow variants of a software system to be generated based on the configuration selected by the user. In this thesis, we focus on C based software systems with build-time variability using a build system and C preprocessor. Such systems usually consist of a *configuration space*, a *code space*, and a *build space*. The configuration space describes the features that the user can select and any configuration constraints between them. The features and the constraints between them are commonly documented in a *variability model*. The code and build spaces contain the actual implementation of the system where the former contains the C code files with conditional compilation directives (e.g., `#ifdefs`), and the latter contains the build scripts with conditionally compiled files. We study the relationship between the three spaces as follows: (1) we detect variability anomalies which arise due to inconsistencies among the three spaces, and (2) we use anomaly detection techniques to automatically extract configuration constraints from the implementation.

For (1), we complement previous research which mainly focused on the relationship between the configuration space and code space. We additionally analyze the build space to ensure that the constraints in all three spaces are consistent. We detect inconsistencies, which we call *variability anomalies*, in particular dead and undead artifacts. *Dead* artifacts are conditional artifacts which are not included in any valid configuration while *undead* artifacts are those which are always included. We look for such anomalies at both the code block and source file levels using the Linux kernel as a case study. Our work shows that almost half the configurable features are only used to control source file compilation in Linux's build system, KBUILD. We analyze KBUILD to extract file *presence conditions* which determine under which feature combinations is each file compiled. We show that by considering the build system, we can detect an additional 20% variability anomalies on the code block level when compared to only using the configuration and code spaces. Our work also shows that file level anomalies occur less frequently than block level ones. We analyze the evolution of the detected anomalies and identify some of their causes and fixes.

For (2), we develop novel analyses to automatically extract configuration constraints from implementation and compare them to those in existing variability models. We rely on two means of detecting variability anomalies: (a) conditional build-time errors and (b) detecting under which conditions a feature has an effect on the compiled code (to avoid duplicate variants). We apply this to four real-world systems: uClibc, BusyBox, eCos, and the Linux kernel. We show that our extraction is 93% and 77% accurate respectively for the two means we use and that we can recover 19 % of the existing variability-model constraints using our approach. We qualitatively investigate the non-recovered constraints and find that many of them stem from domain knowledge. For systems with existing variability models, understanding where each constraint comes from can aid in traceability between the code and the model which can help in debugging conflicts. More importantly, in systems which do not have a formal variability model, automatically extracting constraints from code provides the basis for reverse engineering a variability model.

Overall, we provide tools and techniques to help maintain and create software product lines. Our work helps to ensure the consistency of variability constraints scattered across SPLs and provides tools to help reverse engineer variability models.

Acknowledgements

It is hard to believe that this day has finally come. I know I would not have reached this stage if it was not for the help, support, and guidance of many great people who I was lucky to have as part of my life. My Masters and PhD advisor, Richard C. Holt, definitely makes the top of that list, for he has really shaped the researcher I am today. Ric was always generous with his time providing me the guidance I needed while still giving me the freedom to pursue my own research interests. I will truly miss our blackboard discussions and drawings (which seemed to stay on the board for eternity). As much as they were exhausting sometimes, I always came out of his office seeing things with a different perspective. These discussions always challenged me to think outside the specifics of a project which I might be too indulged in and to always think in terms of the bigger picture.

I was lucky to have great committee members who each provided me with valuable feedback and different perspectives on my work. I greatly enjoyed discussing my thesis with my external examiner, Don Batory. His comments forced me to think beyond the immediate scope of my thesis and to think about next generation programming languages and support tools. Thanks to Joanne Atlee for carefully reading my thesis despite her busy schedule and providing many suggestions to make it better. I also always enjoyed talking to Jo and found her experience and knowledge of the software engineering community very useful. I am happy that I crossed paths with Krzysztof Czarnecki during my comprehensive exam who gave me valuable research advise and provided me the opportunity to explore different research directions with other members of his group. Last, but definitely not least, I would like to thank Michael W. Godfrey who always manages to look at my work from a different angle and to ask me questions that I never expect. His feedback always forces me to think of the broader use of my research and how it can be relevant to different stakeholders. I am also thankful to Mike for never turning me down when I bombarded him with career-related questions. He always took the time to talk to me or answer my emails, and I truly feel lucky to have such a trusted, experienced person to turn to for advice.

During my PhD work, I have learned something new from everyone I collaborated with. I have spent the last year or so of my PhD working closely with Thorsten Berger and Christian Kästner, an experience I greatly enjoyed. Thorsten was always available to discuss things even when on a different time zone and patiently put up with me while I learned Scala. I always thought I was an organized person, but working with someone as organized as Christian made me realize that there is still so much to improve. He also tolerated all the parsing errors I constantly asked him about while I gradually learned how TypeChef works. I also appreciate the discussions I had with both Thorsten and Christian about my next steps after the PhD. I would also like to thank Reinhard Tartler, Christian Dietrich, Christopher Egger, and Daniel Lohmann for welcoming my extension of their undertaker tool and giving me the opportunity to work with them. I definitely learned many experiment setup and LaTeX tricks from them which I continue to use until today.

I was fortunate to be part of the SWAG lab which was always full of bright people to talk to. I would especially like to thank my office-mate, Olga Baysal, who always seems to know something about everything, constantly making our conversations entertaining and a good break from the long working hours. It will feel different not sharing an office with you any more, but I hope we both move onto bigger and better things, such as having offices with windows! I will definitely miss Ian Davis stopping by every now and then to check how I am doing (and probably to make sure I have not lost my sanity yet). Having conversations with Reid Holmes was always fun, and he was always open about sharing advise from his academic experiences which I always found useful. I will truly miss all SWAG members, but I look forward to staying in touch with everyone.

I would also like to acknowledge the scholarships and financial support I received from the following entities during my PhD.

- University of Waterloo: Provost's Doctoral Entrance scholarship, J. Alan George Student Leadership Award, and President's Scholarship
- David R. Cheriton and the David R. Cheriton School of Computer Science: Cheriton Scholarship
- NSERC: NSERC CGS-D

I would like to thank my parents, my sister Ghada, my brother Amr, and my childhood friends Heba, Amira, and May for always supporting my career choices even when it meant being on a different continent and seeing much less of me. Well, I guess if we are going to talk about that, I would like to thank Skype, Whatsapp, and MagicJack as well for making this distance much more bearable. Although nothing can replace family, my stay in Waterloo would not have been the same without the friends I have made here over the years who made it seem a bit more like home. Thank you Zainub, Noran, Allaa, and Safaa for the wonderful memories which I will always cherish. A special thanks to our dear neighbors Mona, Adham, little Maya, and baby Laila for being like a second family to us here. I will surely miss our tea times which I always looked forward to. To all my Waterloo friends, I hope that this is not the end of the journey for us!

Last but not least, I would like to thank my husband, Karim Ali. You have been my partner in crime for the last four years. I do not know whether it was wise for both of us to do our PhDs at the same time or not, but at least I am thankful that our deadlines alternated throughout the year. I do not know how we would have survived otherwise! Thanks for always being there, for taking care of me during those crazy deadlines, and for always forcing me to believe in myself. You have made a tough journey much more bearable and enjoyable!

SARAH NADI

University of Waterloo
June 2014

To my family...

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Henry Ford once said “Any customer can have a car painted any color that he wants so long as it is black” [45]. That quote is representative of an era where customization was difficult and expensive. A simple choice such as the color of the car you buy may not have been feasible since that might change the fixed process used to build the car. As the production process evolved, the concept of product lines in factories was adopted where each part of the product line is responsible for a certain component, and components are then packaged together as they progress through the product line. Software systems went through the same evolution process. Typically, a company would build a single software system for a specific purpose and all its users would expect the same functionality from it with no room for changes or customization. As users started requesting different variations of the product, software engineers created new development techniques to facilitate building similar software products. These include concepts such as software reuse [64], software components [75], information hiding [93] etc. Similarly, *Software Product Lines* (SPLs) [26] have been introduced to provide a systematic way of configuring different, and yet similar, products without having to develop new systems from scratch every time. A software product line builds different variants of the same software from reusable components according to the functionalities selected by the user.

Central to software product lines is the idea of *features*. A feature is some behavior or functionality that is modeled and implemented in a software system [5, 10, 23, 28, 53]. An SPL usually offers several features with interdependencies between them. For example, in an operating system, you cannot have multi-threaded I/O locking without having threading support enabled in your system. These kinds of specifications describe what is referred to as the *problem space* [27, 28, 30]. The problem space describes the features that will be offered to the user and any constraints or dependencies between them. The information from the problem space is usually documented in a *variability model*. Automated and interactive configurators use such models to support users in navigating through valid configurations [14, 33, 126, 127]. *Valid configurations* are combinations of features that respect the constraints documented in the variability model. The features described in the variability model are then implemented in the *solution space* [27, 28, 30] which contains the actual code files and build scripts providing the functionalities intended for these features. Thus, features are described in the problem space and mapped to implementation artifacts in the solution space.

SPLs promise many advantages such as reduced costs of building tailor-made software, improved quality, and reduced time to market which encouraged many companies such as Boeing, Bosch, and General Motors to adopt them for developing their products [4]. There are two sides to SPL development: *maintaining existing* SPLs and *creating new* SPLs. Despite the benefits of software product lines, maintaining existing software product lines or creating new ones is not trivial. Each side comes with its set of challenges.

Maintaining a software product line from which thousands of variants can be generated is not an easy task. Apart from typical challenges of developing large software systems, there are problems unique to SPLs due to their configurable nature. First, as opposed to traditional software where there is only one system to analyze, debug, and test, an SPL has many variants that need to be analyzed. In an SPL with only ten optional features, we already have 2^{10} variants to analyze which makes finding any incorrect behavior more costly and more difficult. Second, the information related to the configurability or *variability* of the system is scattered. As can be seen from the above description, the features provided by the system and their interdependencies are described in a variability model and then implemented in code and build artifacts. This scattering of information leads to redundancies and conflicts between the feature dependencies which may lead to incorrect behavior of the system.

When creating new software product lines, developers can start from scratch, or may re-use existing code (i.e., migrate towards an SPL). In both cases, an SPL should have some form of documented variability model which describes the features provided by the system and their interdependencies. We focus on the migration case where existing systems may have variability in their source code (e.g., through C preprocessor directives), but no documented variability model. Instead, they rely on informal textual descriptions of feature dependencies (e.g., the FreeBSD kernel [103]). As the number of features and their dependencies increases, configuration becomes more challenging [50, 103]. Introducing an explicit variability model is often the way out to control complexity and have one central—human- and machine-readable—place for documentation [80]. Thus, identifying configuration constraints to create variability models is an important step in the creation or migration of SPLs. However, manual extraction of configuration constraints from implementation is a daunting task which calls for automation. In order to automate the creation of variability models, we need to understand the sources of configuration constraints and how we can identify and extract these constraints.

1.1 Motivating Example

Consider a mobile phone company, AwesomePhone, which produces several models of its phones targeted to different markets. For each model, considerations such as the hardware it will run on, software dependencies, and the target price it will be sold for will govern the features the company decides to

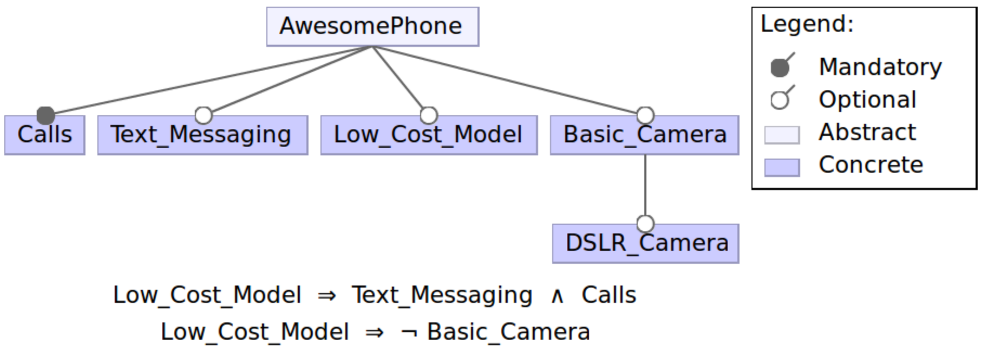


Figure 1.1: AwesomePhone variability model shown in feature modeling notation

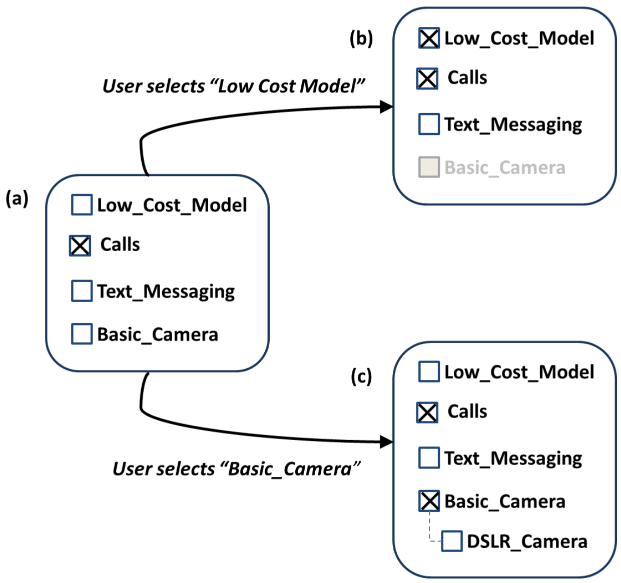


Figure 1.2: AwesomePhone configurator. (a) default view, (b) selecting `Low_Cost_ Model` results in `Basic_Camera` being unselectable, (c) selecting `Basic_Camera` results in `DSLR_Camera` being visible for selection.

include in the phone. For example, one low-cost model of the phone which targets basic users will only contain call and texting capabilities, but no camera. Cathy, one of AwesomePhone's developers, designs and implements a software product line to achieve this.

Cathy discusses the different requirements and functionalities of the phone with her team and then defines the supported features and creates the variability model shown in Figure 1.1. The variability model, created using FeatureIDE [68], is shown in feature modeling notation [53] where the relationships

```
1 #ifdef Basic_Camera
2   //basic camera support code
3   #ifdef DSLR_Camera
4   //DSLR camera support code
5   #endif
6 #else
7   //no camera support
8 #endif
```

Figure 1.3: C code implementing AwesomePhone's camera support

```
1 #ifdef Basic_Camera
2   //camera support code
3 #else
4   //no camera support
5   #ifdef DSLR_Camera
6     //DSLR camera support code
7   #endif
8 #endif
```

Figure 1.4: Bug in the C code implementing AwesomePhone's camera support. DSLR camera support code is dead.

between the different identified features are shown. For example, AwesomePhone must have calling capabilities (mandatory features), but it is optional for it to have text messaging or a camera (optional features). The variability model also shows that `DSLR_Camera` depends on `Basic_Camera` as shown from the tree hierarchy. Cross-tree constraints can be expressed as additional propositional formulas (also part of the variability model) shown at the bottom of Figure 1.1. For example, the additional propositional formulas shown state that the low cost model must have calling and text capabilities but no camera.

Cathy then designs a configurator shown in Figure 1.2a which reads the variability model and displays it to the users of the software product line such that they can select the features they want. Note that since the variability model indicates that cameras are not supported on the low cost model, once the user selects `Low_Cost_Model`, the `Basic_Camera` option becomes grayed out (i.e., unselectable) as shown in Figure 1.2b. Similarly, since `DSLR_Camera` depends on `Basic_Camera`, the user must select `Basic_Camera` in order for `DSLR_Camera` to appear in the menu and become selectable as shown in Figure 1.2c.

Through conditional compilation (e.g., `#ifdefs`), Cathy then uses the features defined in the variability model as macros in the code to implement the corresponding functionality. We show the code relating to the camera functionality in Figure 1.3. The code shows that to get basic camera support, the user must select the `Basic_Camera` option (Line 1). Additionally, to get the DSLR camera support, the user must select *both* the `Basic_Camera` option as well as the `DSLR_Camera` option (Line 1 and Line 3).

1.1.1 Variability Anomalies

Let us assume that Bob, who is an AwesomePhone developer, now uses the configurator to configure a phone that supports a DSLR camera. Bob uses the configurator and selects both `Basic_Camera` and `DSLR_Camera`. Based on this, he expects that when he compiles the software, he would have the software support needed for the DSLR camera to work. To his surprise, he finds that his configured phone does not contain the DSLR camera support code. Bob then complains to Cathy who tries to figure out what went wrong.

Upon inspection of the system's implementation, Cathy realizes that she had incorrectly changed the place of the `#ifdef` check for the DSLR camera support from that shown in Figure 1.3 to that shown in Figure 1.4. Thus, the `#ifdef DSLR_Camera` check on Line 5 in Figure 1.4 is incorrectly placed in the #else branch of the `#ifdef Basic_Camera` check. Since the variability model would never allow Bob to select `DSLR_Camera` without `Basic_Camera` because of the stated dependencies, the DSLR support code on Line 6 of Figure 1.4 is dead. This means that although based on the configuration Bob selected, he expects to get the DSLR camera support, he does not get that functionality because the use of the configuration features in the code is not consistent with those presented in the variability model. We call such situations *variability anomalies* since they can reflect real errors (such as this case) or code smells [46, 121]. Cathy then fixes this anomaly by moving back to the correct implementation shown in Figure 1.3.

1.1.2 Identifying Configuration Constraints

Now suppose that AwesomePhone had never taken the time to identify and document the dependencies between its features. Instead, the company relies on the existing code as well as developers' knowledge of the supported configuration features and their dependencies. While this can work with a small number of features, it does not scale well. As the number of features increases, managing and keeping track of the features and their dependencies becomes difficult. Thus, having a documented variability model (such as that in Figure 1.1) is helpful for telling Cathy the feature dependencies such that she can properly maintain the project and debug problems similar to those Bob faced.

If the company now decides to create such a variability model to better manage its software product line, Cathy might be overwhelmed with the task of going through code implemented and modified over several years to manually identify these dependencies. Cathy would, therefore, appreciate having some form of automated support. In this case, an automatic tool can analyze the code in Figure 1.3 and realize that unless `Basic_Camera` is enabled, `DSLR_Camera` has no effect on the compiled code. From that information, the tool can deduce that `DSLR_Camera` depends on `Basic_Camera`. On the other hand, upon examining the code in the rest of the system (not shown in examples), Cathy finds that the feature `Low_Cost_Model` is not used in the implementation. This means that an automatic tool would not be able to identify that `Basic_Camera` depends on not having `Low_Cost_Model` since this is not a technical dependency reflected in the code. This illustrates that some constraints in the variability model may not be discoverable from analyzing the code. In that case, Cathy may still need to manually look at other sources of information such as marketing requirements. Thus, there are different sources of configuration constraints: some may be discoverable from the implementation and some may only be reflected in other documentation.

1.2 Thesis Research Themes

Our examples involving AwesomePhone have introduced key concepts involving software product lines, features, feature dependencies (or configuration constraints), sources of configuration constraints, variability models, and variability anomalies. We now give an overview of the entire thesis which discusses how variability anomalies can be detected and used to help in both the *maintenance* and *creation* of software product lines.

With respect to the *maintenance* of software product lines, we detect variability anomalies that manifest themselves in terms of dead and undead artifacts. A *dead* artifact is a conditionally-compiled artifact that is never included in any variant of the program irrespective of the user's selection, while an *undead* artifact is a conditionally-compiled artifact that is always included in every variant. Checking that an SPL does not contain any dead artifacts ensures that when a user selects a specific feature, they get the corresponding functionality from the implementation. In some cases, dead code may be useless code that is left behind. Such dead code can unnecessarily complicate the implementation, and developers may end up wasting time on unnecessary maintenance. Checking that an SPL does not contain undead code ensures that the corresponding functionality offered is actually conditional on the user's feature selection. It also ensures that the implementation is not complicated by redundant checks.

To aid in the *creation* of software product lines, we support the reverse-engineering of variability models from implementation. We do so by analyzing the implementation and automatically detecting configuration constraints which prevent the manifestation of variability anomalies. We study the extent to which we can automatically recover existing variability-model constraints using our analysis and classify what additional sources are needed to identify the remaining constraints.

For both maintenance and creation of software product lines, we focus on *build-time binding* which means that the necessary parts of the code are selected at build-time. Other types of binding (e.g., load-time) are described elsewhere [4, 116]. Specifically, we focus on C based systems with build-time variability using the build system and C preprocessor.

1.2.1 Detecting and Analyzing Variability Anomalies

To detect dead and undead code, feature dependencies from different parts of the system (e.g., the variability model, code, and build files) are extracted and cross-checked for consistency. Previous work on consistency checking by other researchers has focused mainly on the code and the variability model without considering the build files [118, 122]. However, developers invest considerable time in maintaining build systems which suggests that they play an important role in the system's implementation [65, 99]. The build system (which usually consists of Makefiles and/or build scripts) is the ultimate controller of what ends up in the final software product. The configuration features presented to users are usually also used in the build system to control which files get compiled. We, therefore, analyze the variability constraints enforced in build systems and study their effect on the detected variability anomalies.

We use the Linux kernel as a case study since it is the largest open source software product line with build-time binding available. We develop a Makefile constraint extractor, MAKEX, which determines under which feature selection will a file get compiled. We use our extractor to quantify the variability in the Linux kernel's build system, KBUILD. We show that almost half (46%) of the configuration features in the Linux kernel are used only in the build system, which indicates that it has important role in the variability implementation of the system.

We extend previous work by Tartler et al. [118] which detected dead and undead code blocks by analyzing the constraints in the code and the variability model. We add the constraints we extract from the build files to the analysis and show that we can detect an additional 20% variability anomalies. We also work on the file level to detect dead and undead files due to conflict of constraints, as well as due to internal problems in KBUILD. We find that anomalies occur rarely on the file level when compared to the code block level. For both levels, we also study the evolution of such anomalies over time. To the best of our knowledge, our work is the first to highlight the importance of the build system constraints in variability analysis *and* to identify possible causes and fixes of variability anomalies. Detecting such anomalies is important for maintenance activities and ensuring that variability remains correctly implemented as the system evolves.

1.2.2 Extracting and Classifying Configuration Constraints

To help in automatically creating variability models, we investigate different sources of configuration constraints and to what extent we can *automatically* and *accurately* extract such constraints from existing implementations using static analysis techniques. We develop two rules which describe desirable properties in configurable software. The two rules rely on detecting build-time errors (preprocessor, parser, type, and linker errors) as well as feature combinations that have no effect on the code. The latter can be viewed as a variation of dead code detection which has a more global nature. Specifically, the two rules are: (1) all valid configurations should build correctly, and (2) they should all yield different products. For both rules, we propose novel scalable extraction strategies based on the structural use of `#ifdef` directives, on parser and type errors, and on linker checks. We design an infrastructure that accurately represents C code based on previous research on variability-aware parsing and type checking [60, 61, 70] where we statically analyze build-time variability effectively without examining an exponential number of configurations. We demonstrate scalability by extracting constraints from four large open-source systems (uClibc, BusyBox, eCos, and the Linux kernel) and evaluate accuracy by comparing the constraints to existing developer-created models. Our results show that our extraction is 93% and 77% accurate respectively for the two rules we use, and can scale to the size of the Linux kernel in which we extract over 250,000 unique constraints. We also show that we can recover 19 % of the existing variability-model constraints using our approach.

In order to determine what additional analyses are needed to recover the remaining constraints, we manually investigate the constraints we could not recover using our static analysis techniques. To classify the sources of configuration constraints, we qualitatively inspect a sample of the variability-model constraints our analysis could not recover. We find five cases where the source of the constraint is beyond our analysis. For example, we find that 28% of these constraints stem from domain knowledge. This includes knowing which features are related and should thus appear in the same configurator menu or knowing which functionalities only work on certain hardware. To the best of our knowledge, our work is the first to quantify the recoverability of variability-model constraints from code using an automated approach and to qualitatively analyze non-recovered ones.

1.3 Contributions

To summarize, we claim the following contributions. These contributions have been published in various conferences and journals (see references in the following list of contributions).

- *Demonstrating the role of build systems in variability implementation* by quantifying it and comparing it to other parts of the Linux kernel [86].
- *Detecting block-level* and *file-level variability anomalies* in Linux by considering constraints in its *build system*, KBUILD [84–86].
- *Identifying causes and fixes* of variability anomalies [83, 84].
- *Developing new analyses to automatically* and *accurately extract configuration constraints* from existing implementation [82].
- *Determining the extent* to which variability model constraints can be *automatically recovered* from implementation and *classifying* the cases in which they cannot [82].

Overall, our work provides tools and techniques to help maintain software product lines by ensuring the consistency of variability constraints scattered across the system. Additionally, we provide automated techniques for extracting configuration constraints from implementation that can be used to reverse engineer variability models. Our work highlights where automatic extraction mechanisms would fall short such that developers and project managers can have realistic expectations.

1.4 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 provides background about the concepts used in this thesis and discusses related work. Chapter 3 describes the configuration information in the build system and how we extract configuration constraints from it. We then show how we detect variability anomalies. In Chapter 4, we discuss block-level anomalies as well as their evolution in Chapter 5. In Chapter 6, we present file-level anomalies. In Chapter 7, we describe the methodology we use to automatically extract variability constraints from implementation and how we compare them to those in the existing variability models. Chapter 8 summarizes the conclusions we draw from the work presented in this thesis.

Chapter 2

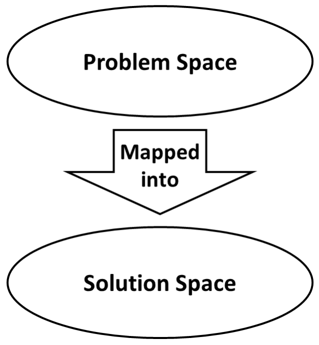
Background and Related Work

The use of software product lines has become increasingly common in both industry and research due to their anticipated benefits. There has been a lot of work done to provide support for maintaining software product lines as well as providing support to migrate legacy code towards a software product line. In this chapter, we present the work most related to the topics discussed in this dissertation. To relate previous research to that presented in this thesis, we first discuss some background and terminology. We then use this terminology to present related work.

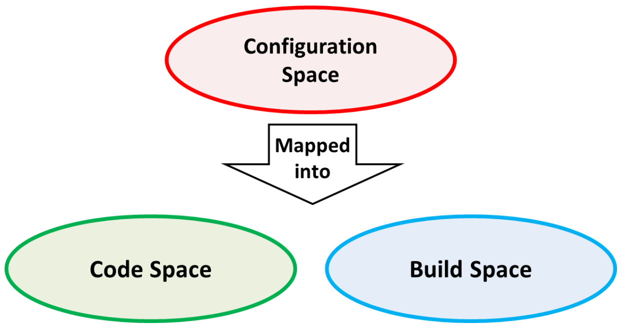
Chapter Organization. In Section 2.1, we present the background behind the three variability spaces discussed in this thesis: *configuration space*, *code space*, and *build space* by using the Linux kernel as an example. We also describe the Linux kernel's development process. We then use these three variability spaces to categorize the relevant related work. In Sections 2.2–2.4, we present previous research which analyzes each of the three spaces. In Section 2.5, we then discuss the work that has been done to analyze the relationship between these variability spaces in terms of co-evolution (Section 2.5.1) and consistency problems (Section 2.5.2). In Section 2.6, we present some of the existing work on software product line creation and migration. We conclude this chapter by summarizing the state of the art and how it relates to the work in this dissertation.

2.1 Background and Terminology

When developing highly configurable software or software product lines, a set of customizable functionalities that can be provided to the user is identified. Requirement engineers, domain experts, and often developers have this information in their mind. For example, a convertible car cannot have a sunroof. These kinds of specifications describe what is referred to as the problem space [27, 28, 30]. The *problem space* essentially describes the different features that will be offered to the user and any constraints or



(a) Problem and Solution Spaces



(b) Three Variability Spaces

Figure 2.1: Features in the problem space are mapped into implementation files in the solution space (a). The solution space can be further divided into the code space and the build space (b)

dependencies between them. These features are then implemented in the *solution space* [27, 28, 30] which contains the actual code files and scripts which provide the functionalities intended for these features. Thus, features cross both spaces: they are described in the problem space and mapped to implementation artifacts in the solution space as shown in Figure 2.1a.

In this thesis, we focus on C-based systems with build-time variability. Such systems have different types of artifacts that support variability. For example, configuration files contain information captured by the problem space while source code and build files contain the low level implementation of the functionalities in the solution space. Thus, we use additional terminology to further identify the different parts of a software system related to its variability implementation. As shown in Figure 2.1b, we use the term *configuration space* to describe the features and their dependencies documented in configuration files. We then divide the solution space into two spaces, *code space* and *build space*, to differentiate where the information lies in the system. There may of course be other types of artifacts such as XML documents or grammar files etc. as noted by previous work [11, 120]. However, although not applicable in the context of the C-based systems we focus on, we would consider such files as part of the code space. For example, the main source code artifacts of a website might be its HTML files. On the other hand, the build space is different since it would consist of any scripts which aggregate or compile the relevant code artifacts.

In this section, we first give an overview of how variants of C-based systems with build-time variability are usually generated. We do so by using the Linux kernel as an example since it is the main subject of study for most of this thesis. Operating systems have to serve different hardware platforms as well as different applications ranging from those on high-end servers to those on embedded devices. The Linux kernel is an example of such an open-source operating system which is considered to be a software product line [106]. Since Linux is the biggest open-source configurable software supporting build-time variability with over 10,000 configurable features [118], we also believe it is a good example of the level of configurability

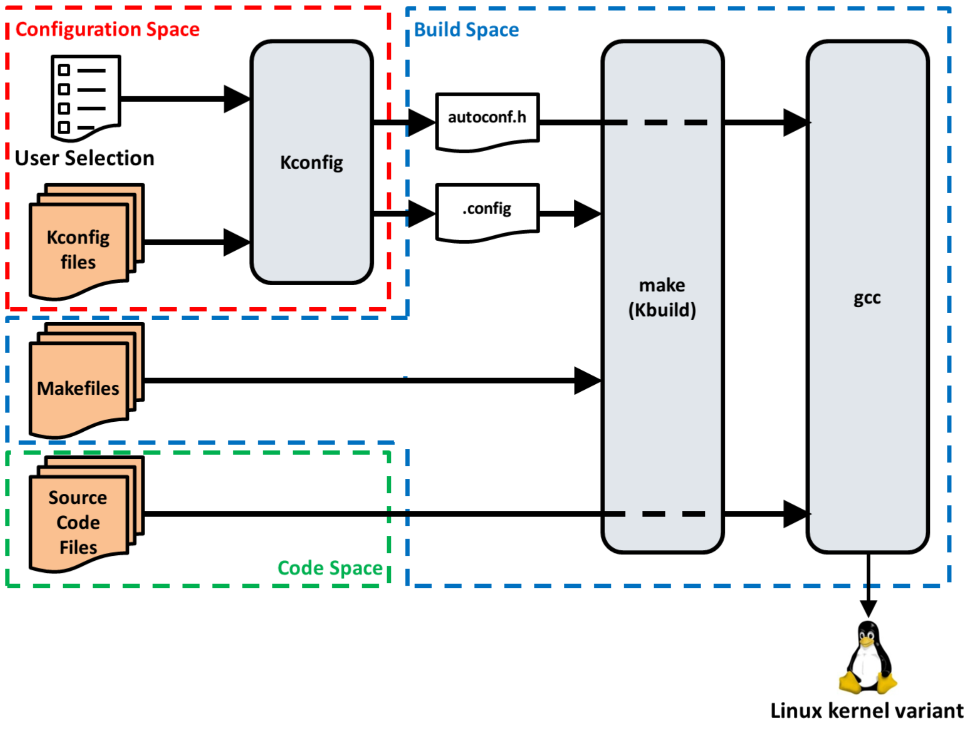


Figure 2.2: The build process of the Linux kernel which is considered to be a software product line

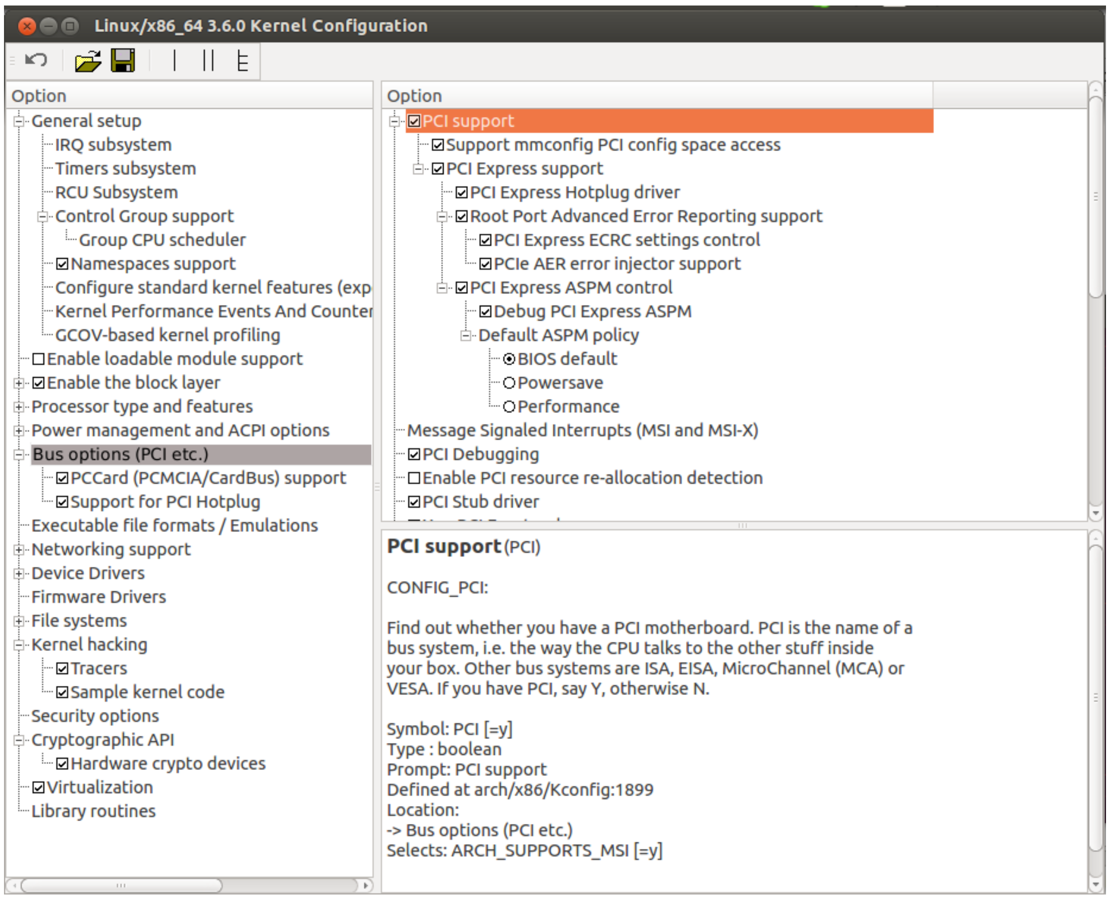
that may be practiced in industry. The concepts discussed here generally apply to build-time configurable systems as will be shown in the other subject systems we study later in this thesis (e.g., BusyBox and eCos).

We show the Linux kernel's build process in Figure 2.2 and use it to explain the details of the three variability spaces in the next three subsections. We also describe its development process as background to the evolutionary studies we perform later in this thesis.

2.1.1 The Linux Kernel Build Process

The Linux build process relies on three kinds of artifacts in the Linux kernel source tree: the source code files, the KCONFIG files, and the Makefiles. These three artifacts are shown on the left in Figure 2.2.

The first step in building the Linux kernel is configuring it. This is done using tools that read the configuration options from the KCONFIG files (details to follow) and display them to the user in a menu format. These tools include `menuconfig`, `xconfig`, and `qconfig` which provide different formats of a configurator [32, 53, 110]. *Configurators* are interactive tools that help users achieve a desired configuration



```
General setup
    IRQ subsystem
    Timers subsystem
    RCU Subsystem
    Control Group support
        Group CPU scheduler
    ☑Namespaces support
    Configure standard kernel features (exp
    Kernel Performance Events And Counter
    GCOV-based kernel profiling
☐Enable loadable module support
☑Enable the block layer
Processor type and features
Power management and ACPI options
Bus options (PCI etc.)
    ☑PCCard (PCMCIA/CardBus) support
    ☑Support for PCI Hotplug
Executable file formats / Emulations
Networking support
Device Drivers
Firmware Drivers
File systems
Kernel hacking
    ☑Tracers
    ☑Sample kernel code
Security options
Cryptographic API
    ☑Hardware crypto devices
☑Virtualization
Library routines
```

```
☑PCI support
    ☑Support mmconfig PCI config space access
    ☑PCI Express support
        ☑PCI Express Hotplug driver
        ☑Root Port Advanced Error Reporting support
            ☑PCI Express ECRC settings control
            ☑PCIe AER error injector support
        ☑PCI Express ASPM control
            ☑Debug PCI Express ASPM
            Default ASPM policy
                ◉BIOS default
                ○Powersave
                ○Performance
Message Signaled Interrupts (MSI and MSI-X)
☑PCI Debugging
☐Enable PCI resource re-allocation detection
☑PCI Stub driver
```

PCI support (PCI)

CONFIG_PCI:

Find out whether you have a PCI motherboard. PCI is the name of a bus system, i.e. the way the CPU talks to the other stuff inside your box. Other bus systems are ISA, EISA, MicroChannel (MCA) or VESA. If you have PCI, say Y, otherwise N.

Symbol: PCI [=y]
Type : boolean
Prompt: PCI support
Defined at arch/x86/Kconfig:1899
Location:
-> Bus options (PCI etc.)
Selects: ARCH_SUPPORTS_MSI [=y]

Figure 2.3: Linux kernel configurator

by offering intelligent choice propagation and conflict resolution facilities [14, 33, 126, 127]. We show a snapshot of the Linux kernel's configurator in Figure 2.3.

After the user configures the kernel through the configurator, two files are produced: the `.config` file used internally by the Linux kernel's build system KBUILD and the `autoconf.h` file used by the GCC compiler. These files contain the user's selection of features. Although they contain the same information, they have different formats since they are used in different places. Entries in the `.config` file used by KBUILD have the format shown in Listing 2.1. This format defines environment variables that will be used in the KBUILD Makefiles to control which files get compiled. A feature that is selected will be defined as an environment variable with the value 'y'. Note that it is a convention to attached the prefix CONFIG_ to a configuration option when used in the implementation to differentiate it from other macros. For example, in Line 1 of Listing 2.1, the entry shows that feature `PCI` is selected in this configuration.

```
CONFIG_PCI=y
CONFIG_PCI_MMCONFIG=y
```

Listing 2.1: Examples of .config entries

On the other hand, the same information will be present in the `autoconf.h` file, but with the format shown in Listing 2.2. This is essentially a header file that defines preprocessor directives that control selective compilation by the GCC compiler. Here, selected features have the value '1'. For example, Line 1 in Listing 2.2 also shows that feature `PCI` is selected in this configuration by defining its corresponding macro to 1.

```
#define  CONFIG_PCI 1
#define  CONFIG_PCI_MMCONFIG 1
```

Listing 2.2: Examples of autoconf.h entries

Based on the features defined in the `.config` file, the Makefiles instruct GCC to compile and link certain files into the final kernel variant shown at the bottom of Figure 2.2. The Makefiles also force the header file `autconf.h` to be included in all source code compilation. Accordingly, when the C preprocessor reads these source files, the features defined in the header file `autoconf.h` determine which parts of the code get passed to the compiler based on the preprocessor directives (`#ifdefs`).

Figure 2.2 shows how the three artifacts as well as the processes and tools fit into three spaces that ultimately control variability of the kernel. These are the *configuration space* (consisting of KCONFIG files), the *code space* (consisting of source code files), and the *build space* (consisting of KBUILD Makefiles). The following three subsections provide more details about these three spaces.

2.1.2 Configuration Space

We use the term *configuration space* to describe the set of features supported by the system and the constraints between them. Ideally, the information in the configuration space (or problem space) would be documented in some form of *variability model* that describes features and constraints in a central place. In Linux, the configuration space consists of KCONFIG [132] files which document the various configuration options provided and their interdependencies.

Each configuration feature has a `config` entry in a KCONFIG file. Additionally, there are menu items used to group related features together for better display. Listing 1 shows examples of KCONFIG entries. The first entry is a menu which displays the different bus options. Note that this is the highlighted item on the left hand side of the configurator in Figure 2.3. The next entry is a feature called `PCI` of type `bool`. This

Listing 1 KCONFIG example

```
menu "Bus options (PCI etc.)"

config PCI
  bool "PCI support"
  default y

config PCI_MMCONFIG
  bool "Support mmconfig PCI config space access"
  depends on PCI

config XEN_PCIDEV_FRONTEND
  tristate "Xen PCI Frontend"
  depends on PCI
  select PCI_XEN
  help
    The PCI device frontend driver allows the kernel to import arbitrary
    PCI devices from a PCI backend to support PCI driver domains.

endmenu
```

means that the feature can be either selected or not selected at any given time. In this case, it is selected by default since it has a `default y` value. Note that the text between quotes after the feature type (`bool` here) shows the prompt being displayed to the user. This can be seen as the first entry highlighted on the right hand side of the configurator in Figure 2.3.

The next entry in Listing 1 defines feature `PCI_MMCONFIG`. Here, this feature depends on `PCI`. This means that it cannot be selected unless `PCI` is also selected. Note that when displayed in the configurator, feature `PCI_MMCONFIG` whose prompt is "Support mmconfig PCI config space access" is displayed as a child of the "PCI support" feature (i.e., feature `PCI`) showing that features in KCONFIG have a hierarchy.

The third entry in Listing 1 shows another feature, `XEN_PCIDEV_FRONTEND`, of type `tristate` which means that the feature can be compiled as a loadable module. In this case, the feature can be selected and would have value `y` in the generated `.config` file or can be selected to be built as a loadable module and will have value `m` in the generated `.config` file. This entry also depends on `PCI`. Additionally, it has a *reverse dependency* on `PCI_XEN` (feature not shown) as seen from the `select` clause. This means that when the user selects `XEN_PCIDEV_FRONTEND`, the configurator would automatically select `PCI_XEN`. Such dependencies are *cross-tree* in the sense that there is no parent-child relationship here, but the feature selected may appear elsewhere in the model. For more details about different relationships in KCONFIG, we refer the reader to the work done by She et al. [104].

The dependencies enforced in KCONFIG (i.e., the configuration constraints) can stem from *technical restrictions* present in the solution space such as dependencies between two code artifacts. Additionally, they can stem from outside the solution space such as external hardware restrictions. Constraints can also be *non-technical*, stemming from either domain knowledge outside of the software implementation, such as marketing restrictions, or from configurator-related restrictions, such as to organize features in the configurator or to offer advanced choice propagation.

For example, in the Linux kernel, a *technical constraint* which is reflected in the code is that "multi-threaded I/O locking" depends on "threading support" due to low-level code dependencies. A *technical constraint* which cannot be detected from the code is that "64GB memory support" excludes "386" and "486" CPUs, which stems from the domain knowledge that these processors cannot handle more than 4GB of physical memory. A *non-technical*, configurator-related, constraint is that feature "PCI support" appears under the menu feature "Bus Options" in the configurator hierarchy.

In this section, we have discussed the information contained in the configuration space: configuration features and their dependencies. In the next two sections, we discuss the information contained in the two other spaces: code space and build space.

2.1.3 Code Space

As previously mentioned, the solution space contains the implementation files (e.g., code, scripts, Makefiles etc.) that implement the features defined in the configuration space. We use the term *code space* to describe the set of source code files (e.g., C or C++) files that implement the functionality of these features. In order to have certain functionalities present only when their respective features are chosen, corresponding blocks in the source code are conditionally compiled. In Linux (as well as in the other subjects studied in this thesis), this is done through C Preprocessor (CPP) directives such as `#ifdef`, `#ifndef`, `#elif`, and `#else`. Listing 2 shows an example of a conditionally compiled code block.

Listing 2 Variability in the Source Code

```
#ifdef CONFIG_PCI
//Block 1
#else
//Block 2
#endif
```

In this example, Block 1 will be compiled only if feature `PCI` is selected (i.e., it is defined in the generated `autoconf.h` header file). Note that we can tell that this is a configuration option from the `CONFIG_` prefix. On the other hand, Block 2 will only be compiled if feature `PCI` is not selected (i.e., it is not defined in the generated `autoconf.h` header file). We use the term *presence condition* to refer to the Boolean expression under which an artifact (a code block in this case) is compiled. Thus, the presence condition of Block 1 in this example is `PCI`.

2.1.4 Build Space

The build space consists of all the build files and scripts (e.g., Makefiles) that are responsible for compiling and linking the source code. For example, Linux's build system, KBUILD, is composed of a collection of KBUILD files (which are essentially Makefiles) as well as some build scripts. The Makefiles contain entries that control the compilation of whole source files, thus controlling what ends up in the final product variant. While some source files are always compiled, others are *conditionally* compiled according to some feature selection. Thus, similar to code blocks, files also have *presence conditions*. For example, the presence condition of a file `foo.c` may be feature `FOO` which means that `foo.c` is not compiled unless `FOO` is selected.

KBUILD uses a specialized syntax to express these file presence conditions. We will not go into the details of this syntax here as we dedicate Chapter 3 to describe KBUILD's syntax in details and how we identify the file presence conditions.

2.1.5 The Linux Development Process

So far, we have described the three types of artifacts contributing to variability in the Linux kernel. In this subsection, we describe the development process followed to modify these artifacts since it relates to the evolution studies we describe in related work below and which we also perform in this thesis.

Changes to the Linux kernel artifacts described above happen through a strict review process. A proposed *change* may add new functionality as well as modify, correct, or remove existing functionality. The change is first proposed on focused mailing lists that are read by experts in the Linux subsystem that the change addresses. Proposed changes are required to include a summary of the problem, a detailed description, as well as the suggested code change in the so called *unified diff* format[1]. The content, description and format of the suggested change are then reviewed by developers who are familiar with the code to ensure that the presented change is formally correct, understandable, and complete. Proposed changes get approved by subsystem maintainers by including the proposed change as a GIT commit into the focused subsystem repositories that only they control.

Each GIT commit contains information such as the author and date of the commit, a detailed description of the commit, as well as the patch applied. The *patch* contains the textual change to the modified files. These modified files can be documentation files, KCONFIG files, source code files, or a Makefile. The commit patch can be displayed in unified diff format where lines removed are prefixed with a '-', while lines added are prefixed with a '+'. For example, the following snippet shows an example of a KCONFIG patch renaming a feature in KCONFIG.

[1]A standard format for interchanging code changes that is understood by the UNIX patch(1) tool

```
-config SPI_BFIN
+config SPI_BFIN5XX
  tristate "SPI controller driver ... "
```

The patch shows that the line containing the old feature name has been removed (`-config SPI_BFIN`), while the line containing the new feature name has been added (`+config SPI_BFIN5XX`). This can be interpreted that this KCONFIG feature is being renamed. Since each *commit* implements some *change* and has an associated *patch* transcribing this change, we use the three terms interchangeably throughout the thesis for simplicity.

At the beginning of each new Linux release cycle, referred to as the *merge window*, subsystem maintainers ask Linus Torvalds, who maintains the Linux master GIT repository, to integrate the batched changes from their subsystem into his master repository. From this master repository, official Linux releases are made for use by Linux distributors and end-users. This organizational structure ensures that all code and the corresponding descriptions of the changes in the master repository have been reviewed by at least two experts. The case studies in this thesis can therefore reliably identify focused, well-documented changes in a large-scale, professionally driven open-source project.

In the remainder of this chapter, we use the concepts we have described in this section (Section 2.1) to present related work where we describe previous work done to analyze each of the three variability spaces as well as their inter-relationships.

2.2 Analyzing the Configuration Space

Hubaux et al. [50] conducted a survey of configuration challenges in the Linux kernel and eCos. One of their findings indicates that developers often face situations where conflict resolution is needed, and that current tools still lack proper guidance. The dependencies in a variability model are typically expressed uniformly as a single large Boolean function expressed in propositional logic to describe all valid configurations and help reasoning about them [19, 102, 118, 128]. Such semantics include looking at this configuration information in terms of *feature models*. Feature models are a method of variability modeling to represent the commonalities and variabilities within a software system [53], and have been closely linked to the field of software product lines [79]. Feature models have a tree-like structure where child-parent relationships (*hierarchy constraints*), as well as inter-feature constraints (*cross-tree constraints*) can be identified [18, 53]. An example of a feature model for a part of the Linux kernel is shown in Figure 2.4. In this example, we can see the tree-like structure showing the hierarchy of features. Feature relationships include direct dependencies, exclusions, as well as alternative groups.

The constraints shown in Figure 2.4 are examples of what we call *configuration constraints*. For example, we can see that `cpu_hotplug` $\rightarrow$ `powersave` based on the cross-tree edge shown in the model. There has been much research to extract such constraints from the configuration space [16, 102, 118]. Such extractors can interpret the semantics of different variability modeling languages to extract both hierarchy

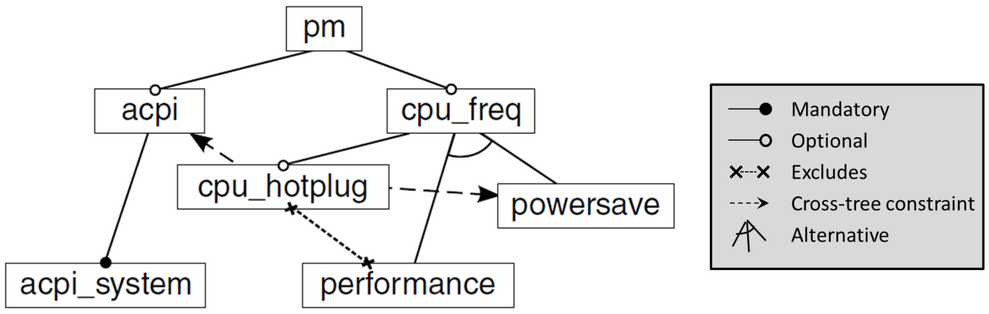


Figure 2.4: Example of a feature model for part of the Linux kernel's power management subsystem [103]

and cross-tree constraints as well as represent all constraints in a single Boolean formula. For example, She et al. [104] and Sincero et al. [107] present Linux's use of KCONFIG [132] as a large-scale example of a variability model used in practice. She et al. [104] show how feature modeling constructs are used in practice in the Linux kernel. Similarly, Berger et al. [19] look at more uses of feature modeling concepts by comparing the variability models of the Linux kernel which uses the KCONFIG language and eCos which uses the CDL language [125].

While we only focus on Boolean features (i.e., those that can only be selected or deselected), non-Boolean features and dependencies do exist in practice. For example, Passos et al. [95] study the non-boolean constraints present in eCos. Their findings show that non-boolean constraints are heavily used in eCos. However, encoding non-boolean features and their dependencies in logical formulas is a difficult problem which we do not attempt to address in this thesis.

2.3 Analyzing the Code Space

Despite the disadvantages of using the C preprocessor to implement variability at the source code level [112], it remains a popular tool not only in the Linux kernel but in many other open-source [42, 69] and industrial software systems [51]. To that extent, there has been much research into understanding the variability introduced by the preprocessor directives and finding ways to accurately parse the code. While parsing C code is a well-solved problem, parsing *unpreprocessed* C code (i.e., while it still has the #idefs in place) is challenging. Such a task becomes necessary when you want to reason about the behavior of the source code in all possible variants of the program. However, generating all possible variants to analyze them is not practical. There has been several research efforts to address this.

Padioleau [91] developed a parser for C/C++ code which can take the C preprocessor (cpp) constructs directly into account without preprocessing. That way, an AST can be created with a representation of the cpp constructs. The author's main goal is to allow easier evolving and refactoring of such code. However, the technique depends on several heuristics noted from the cpp usage in the code and does not capture all uses (unsound). In a similar effort, Sincero et al. [109] analyze the source code to derive presence conditions for each line of code. In this case, the presence condition is simply the `#ifdef` condition the code is enclosed within, including nested conditions. However, the authors do not expand macros or consider the interaction between `#define` and #if directives. They rely on the explicit macro uses in the code without expanding the necessary macros and propagating variability. While this is an approximation which may work on a local scale to understand under which conditions a piece of code is compiled, it is also unsound. The work done by Baxter and Mehlich [12] also attempts to parse unpreprocessed C code. While they correctly propagate macro definitions through `#define`, for example, they assume some limitations to how the preprocessor directives can be used. This includes assuming that preprocessor directives such as #if and `#ifdef` can only wrap entire statements or functions. In practice, preprocessor directives can have different levels of granularity that may not align with statements or functions [55].

To overcome these shortcomings, Kästner et al. [60] have recently developed a variability-aware code analysis infrastructure, TYPECHEF[2] which consists of three variability-aware components. The first is the partial preprocessor [59] which includes all necessary header files and expands all macros (unlike most preceding work) while keeping conditional compilation directives such as `#ifdefs` in the code. Each recognized token has a presence condition which shows under which condition it has been read. The conditional token stream generated by the partial preprocessor is then passed to a variability aware parser which tries to optimize the parsing process through splitting and combining different token streams according to their presence conditions. This allows the whole conditional-token stream to be parsed in a single pass and a conditional single abstract syntax tree (AST) to be produced. Each node in the conditional AST is guarded by a Boolean expression which shows under which configuration can this node be reached. The conditional AST can then be used for different static analyses. While TypeChef has mainly been used to analyze build-time behavior such as parse and type errors, other work such as that by Bodden et al. [22] looks at feature-aware data flow analysis for software product lines. They design an infrastructure with similar concepts (but different underlying data structures) to support feature-aware inter-procedural analysis for Java programs.

2.4 Analyzing the Build Space

There have been studies showing that the build system is an integral part of any software system [49, 65]. Developers do interact often with the build system and may spend considerable amounts of time doing a simple task if the build system is very complex [88]. The build system becomes even more important from a variability perspective since it is the final controller of what actually gets built in a software system. However, with the exception of the work by Berger et al. [17] to extract file presence conditions and that of Trujillo et al. [124] to refactor build files into the related modules, the build space has typically been analyzed without the role of variability in mind. For example, Adams et al. [1] developed a tool, MAKAO, for visualizing and manipulating build systems, and have applied it to KBUILD. They mainly focused on

[2]https://github.com/ckaestne/TypeChef

the targets that appear in Makefiles and their dependencies, but did not study the configuration features that appear in Makefiles and how they contribute to the Linux kernel's configurability. Along the same lines, Tamrawi et al. [117] create a symbolic build dependency graph by analyzing a build system using symbolic execution. Symbolic execution can handle the dynamic nature of build systems (e.g., variables depending on the operating environment) while tools like MAKAO analyze the build system for a specific environment to produce a concrete dependency graph. There has been more recent work to analyze the variability in the build space [34, 35] which we discuss in Chapter 3.

2.5 Relation Among Variability Spaces

As seen in the build process of the Linux kernel described in Section 2.1.1, the information from all three spaces interacts to generate the final variant. Based on that, there has been much research studying the relationships between these three variability spaces on different subject systems. This was either studied from a co-evolution perspective to understand if changes may span multiple spaces or from a conflict perspective to see if the constraints in these spaces are consistent. We discuss previous work related to both of these aspects in the next two subsections. We would like to note that previous work by Batory et al. [11] has pointed out that software product lines (or more broadly feature-oriented systems) have several *representations* such as code, documentation, grammar files etc. and that all such representations must be kept consistent. Based on that, they developed an generic algebraic model, AHEAD, that helps with the composition of any representation into the final product. In our work, we focus on the three spaces we describe because all artifacts related to producing the final product can be found in these spaces. We do not deal with non-implementation related artifacts such as documentation and describe the following related work from that perspective.

2.5.1 Co-evolution

Traditionally, the source code has been the main subject of evolution studies (e.g., [47]). However, as researchers realized the importance of other artifacts [99], more studies about the evolution of artifacts such as the configuration files [123] as well as co-evolution studies of several artifacts emerged. Lotufo et al. [72] study how the variability model of the Linux kernel (i.e., the KCONFIG files) evolves over time. The various versions of KCONFIG are compared in terms of their complexity. The authors find that the KCONFIG feature model grows linearly in size (in terms of LOC) with the Linux source code which indicates their close relation and suggests that the Linux kernel development is feature driven.

Similarly, Adams et al. [2] study the evolution of the Linux build system (KBUILD). Their work focuses on how Linux Makefiles evolve over time in comparison to other artifacts in the system. They show that the build system does indeed grow in complexity over time, and that it exhibits similar evolution patterns to source code. Additionally, they show that it is constantly maintained to reflect design decisions in the kernel. However, as they point out, their study focuses on using one preset configuration of the kernel which means that it is not variability-aware and does not examine the interplay between the configuration files and the build files.

There have also been co-evolution studies on systems other than the Linux kernel including those which may not necessarily be categorized as SPLs. For example, McIntosh et al. [76, 77] study the evolution of Java build systems such as ANT and Maven. They also find similar growth patterns to those found in the Linux kernel, as well as similar correlation with source code changes. Additionally, they find that although build systems account for a small percentage of the files in a project, they have a comparable churn rate to source code [78] which suggests that they are also likely to have defects. The authors indicate that it is necessary to understand the impact of a change in the source code on the supporting build system. However, these studies only correlate the rate of evolution of source and build files, but do not examine the details of how they actually cooperate together to provide the final product.

To consider the relation between all three variability spaces, Passos et al. [94] recently study the co-evolution of the variability model with the other artifacts. They document the patterns they observe in the evolution of the Linux kernel (e.g., adding a modular feature) and how related artifacts such as the code and build file get affected. Their findings show that changes in the variability model often need to be accompanied with changes in the code and build files indicating that they are closely related. Since all the above evolution studies show the close relationship between the different artifacts, we believe it is important to study this from a variability perspective and ensure that all three artifacts are consistent to decrease the chance of anomalies.

2.5.2 Consistency Among Variability Spaces

Configurable software allows the user to build a product with a selected set of features. In turn, when users select a specific feature, they expect certain functionality to accompany it. Figure 2.5 shows the three spaces responsible for implementing variability: the configuration space, the code space, and the build space. In order for variability to be correctly implemented, each space must contain the information needed to achieve the intended variability and all of the three spaces must be kept consistent. Any inconsistencies would lead to what we call, variability anomalies.

Variability anomalies could be in the form of conditionally dead or undead code as shown in the introduction. They can also be in the form of build-time errors such that the selected variant does not build correctly (e.g., has a type error). Alternatively, a selected configuration may build correctly but may not behave correctly at run-time. We use the term anomalies as a general umbrella for all these problems. This is because in some cases dead code, for example, may represent code which the developers do not care about rather than a missing functionality. In other cases, dead code may be a result of an error. Similarly, if a valid configuration does not build correctly, this is an error that needs to be fixed. Thus, the term anomaly indicates something which is different than the expected behavior and can represent both types of problems. We now discuss the previous work related to both categories.

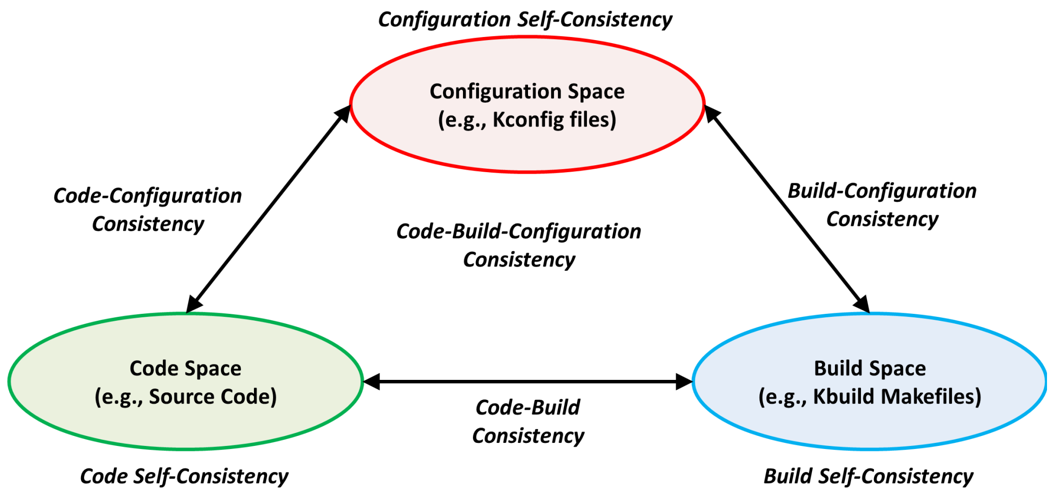


Figure 2.5: Consistency among the three spaces

We use the consistency relationships illustrated in Figure 2.5 to categorize the related work that has been done to ensure consistency of the variability spaces. We present the three spaces and discuss the consistency checks that relate to them. Note that the related work discussed here is mainly that which takes the variability or configurability into account.

Configuration Space Consistency. There has been a lot of work in making sure that the information in the configuration space is consistent (i.e., configuration self-consistency). Checking the consistency of feature models, as a representation of the configuration space, has been tackled by several researchers. Translating feature models into propositional logic [74, 101, 115] to help with reasoning about them has become a common practice. For example, Batory [9] translates feature models to propositional logic to facilitate debugging them such as finding contradictions. There has also been work done on debugging individual configurations. That is, provide support for users to identify why there is a conflict in their selected configuration and what they can do to resolve it. For example, White et al. [126] use Constraint Satisfaction Problems (CSPs) to debug individual configurations. Alternatively, Xiong et al. [127] propose the idea of range fixes which provides the user with a set of fixes which can resolve the conflicts in their configuration. Their technique can handle non-boolean constraints which is a limitation of other related techniques (e.g., [87]).

Code Space Consistency. Many of the related work for analyzing the code space discussed in Section 2.3 has been used as a foundation for detecting build-time errors in variable software (e.g., [22, 60, 109]). In that sense, it ensures code self-consistency. However, it is often the case that information from the variability model about the allowed valid configurations is used to limit the combination of features analyzed. In that sense, the code-configuration consistency has by far received the most attention. This is usually done by checking if the valid configurations enforced by the variability model contain any build-time errors such as parsing or type errors. To do so, the variability model is again expressed as propositional logic.

Thaker et al. [121] and Czarnecki et al. [29] were among the first to show that propositional logic and SAT solvers can be successfully used for such consistency checks. Thaker et al. [121] have introduced the term *safe composition* to indicate that all modules of an SPL can be safely combined for valid configurations allowed by the variability model. Ensuring type safety was the main underlying premise of safe composition where different constraints are expressed using propositional formulas and a SAT solver is used to detect any composition problems. While Thaker et al. mainly focused on code artifacts (although their discussion includes other artifacts as well), Czarnecki et al. [29] check for consistency with template based models (i.e., related to UML representations). They also check for consistency of the feature model with respect to the rest of the system, but their work focuses on feature-based model templates where they ensure that no ill-formed template can be produced.

Detecting conditional type errors has actually been a popular way to find inconsistencies [6, 8, 63] where conditional symbol tables based on `#ifdef` usage are used to detect type errors. Also related is the optional feature problem discussed by Kästner et al. [56] as a mismatch between the intended variability in the configuration space and that implemented in the code space. The optional feature problem is when two features are presented as optional features in the configuration space, but their implementations are not independent. This means that the implementation of one feature actually affects the implementation of the other even if not selected. Tartler et al. [118] look at consistency from a different perspective by comparing the constraints in both the configuration space and the code space in the Linux kernel to detect any inconsistencies which might lead to dead or undead code. The research efforts discussed here are those most related to this thesis. For more details on other strategies to analyze software product lines or detect other inconsistencies, we refer the reader to the survey by Thüm et al. [122].

Build Space Consistency. We are not aware of any work that specifically looks at the relation between the variability in the build system and that in the code space or in the configuration space. While Berger et al. [17] analyze the build system, they do not check for consistency. Although with a different intent, an exception is the work done by Kästner et al. [60] and more recent work by Liebig et al. [70]. In both works, the authors use the file presence conditions from the build system to limit the combination of features they need to analyze for parsing and type checking. However, they do not analyze the direct relationship (and thus any inconsistencies) between the three variability spaces.

2.6 Supporting the Creation of SPLs

Given the benefits of software product lines, there has been much research about how to aid the migration of legacy systems to software product lines. It is common for several similar products to be developed for several years before a company attempts to create a software product line [44]. This is commonly done through cloning mechanisms where the original system is cloned several times and changed to support different requirements. While some researchers looked into providing ways to manage existing clones [100], others have advocated refactoring the system into a software product line by analyzing the commonalities and differences between these existing products [36, 37]. Researchers have looked at the best way to re-architect the system [13, 20, 41, 105, 114] including identifying the parts of the code related to a specific feature (i.e., feature location) [40, 58]. Identifying variation points between existing products is also important to identify the configurable features in the first place [131]. Other researchers have developed re-engineering approaches by analyzing non-code artifacts such as product listing comparisons [31, 48].

Creating a variability model is an important aspect of software product lines and one of the hardest to do manually, especially for large systems. For example, in an experience report by Danfoss Drives on their migration to a software product line [51], they mention creating a feature model as one of their main challenges. The authors indicate that the company followed a bottom-up approach where they merged the code of different existing products and looked at the source code for variation points (e.g., `#ifdef` directives) to identify features and possible relationships. However, the process does not seem to be automated or is probably semi-automated, at best. In an effort to help in automatically creating feature models, She et al. [103] and Anderson et al. [3] develop algorithms to reverse engineer feature models from a set of constraints. The algorithm requires a set of constraints as an input and can then determine the feature hierarchy and other relationships between features such as groups etc. In practice, these input constraints would be extracted from the code or provided by domain experts, for example. She et al. [103] test their algorithm on the Linux kernel, eCos, and FreeBSD. However, they approximate the constraints they extract from the code using an inexact parser for the code. Zhang and Becker [129] also propose a technique to extract configuration constraints from code. However, they rely on local approximations of the `#ifdef` usages and do not analyze the global usage of features across the system.

Although our focus is build-time variability in this thesis, it is worth mentioning that there has been work done on analyzing load-time variability. For example, Rabkin and Katz [96] identify where configuration options are used in the code through points-to analysis and call graphs. While they identify the valid values for these options, they focus on Java programs and do not identify any dependencies between the options. Along the same lines, Zhang and Ernst [130] also look at load-time options but with the purpose of identifying incorrectly configured software.

2.7 Summary

At the time we started this work, research efforts on the consistency of configurable software has mainly focused on the configuration and code spaces, but less focus has been given to the build space. In this thesis, we argue that including the build space in variability analysis is important. We study the variability in the build space and show its role in detecting inconsistencies depicted in terms of dead and undead code.

We also study the evolution of variability anomalies to understand why they occur and how they can be fixed.

While we have discussed some previous work that analyzes variability in the code to extract configuration constraints, the methods used were usually either imprecise approximations or local to specific code. To the best of our knowledge, there has been no work that accurately extracts global configuration constraints from the full implementation of a system (including the build space). Additionally, most work that extracts configuration constraints focuses on consistency problems for error reporting and does not study the sources of these constraints. Understanding what knowledge configuration constraints reflect is important to aid in the creation and migration of variability models for software product lines. In this thesis, we propose a new technique to accurately extract configuration constraints from implementation and provide a classification of the different sources of configuration constraints in existing variability models.

Chapter 3

Variability in Build Systems

As discussed in Chapter 2, most of the literature on the consistency of configurable software has focused on the configuration and code spaces. One of the contributions of this thesis is analyzing the build space and studying its effect on the consistency of the system being analyzed. We use the Linux kernel as a case study since it is the biggest open source configurable software supporting build-time variability available, where we analyze its build system, KBUILD.

KBUILD consists of several Makefiles which are responsible for compiling the source code files. In some cases, these files are *conditionally compiled* which means they only get compiled if the user selects a certain feature. Thus, a file has a *presence condition* which is the combination of features that need to be selected in order for it to compile. We call these file presence conditions *the build space constraints*. In this chapter, we want to analyze KBUILD to extract these constraints as shown in Figure 3.1. We do so through our tool MAKEX. We will use the extracted build space constraints later in this thesis to detect conflicts at both the code block and code file levels. In this chapter, we also use MAKEX to quantify the variability in KBUILD. We do so by proposing metrics such as measuring the number of configuration features used in KBUILD and the complexity of the extracted constraints.

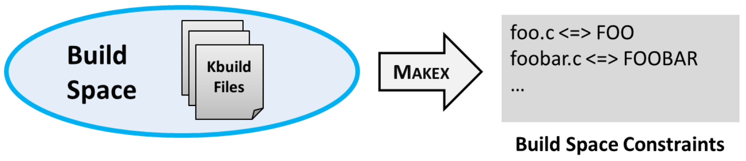


Figure 3.1: Analyzing KBUILD using MAKEX to extract the build space constraints

Chapter Organization. Section 3.1 provides a high-level recap of how the Linux kernel's build system (KBUILD) works. Section 3.2 then describes how KBUILD conditionally compiles files. Section 3.3 discusses how we extract configuration constraints from KBUILD using our *Makefile constraint extractor*, MAKEX. Section 3.4 presents the metrics we define to analyze and quantify the variability in KBUILD. In Section 3.5, we report the results of applying these metrics to several releases of the Linux kernel, and in Section 3.6, we discuss our insights into understanding variability in KBUILD. We then present some related work in Section 3.7 and provide a summary of the chapter in Section 3.8.

Related Publications. The work described in this chapter has been published in the following papers:

Sarah Nadi and Ric Holt. "Make it or break it: Mining anomalies in Linux Kbuild". In: *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. 2011, pp. 315–324

Sarah Nadi and Ric Holt. "Mining Kbuild to detect variability anomalies in Linux". In: *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. Los Alamitos, CA: IEEE Computer Society, 2012, pp. 107–116

Sarah Nadi and Ric Holt. "The Linux kernel: A case study of build system variability". In: *Journal of Software: Evolution and Process* (2013). Early online view. `http://dx.doi.org/10.1002/smr.1595`

3.1 Recap of Linux's Build System (KBUILD)

As shown in the Linux kernel build process in Figure 2.2, KBUILD uses Makefiles in the Linux source code tree which are responsible for compiling and linking the source code into the particular kernel variant. The Linux kernel source code is stored in many directories where each directory has a collection of source files responsible for a certain functionality or a certain subsystem. There is usually a Makefile in each of those directories, and each Makefile is mainly responsible for compiling the files in its directory [66]. The Makefile at the root of the Linux source tree (the top Makefile) is responsible for setting up all the environment variables that are needed during the build process, e.g., the CPU architecture being built, compiler options etc. The top Makefile reads the `.config` file which comes from the Linux kernel configuration process (see Figure 2.2) and which specifies all the features that have been selected by the user. All Makefiles in the system have access to the features defined in the `.config` file. The build process starts with the top Makefile and then recursively descends into the Makefiles of the subdirectories as directed by the current Makefile.

```
1 obj-y += fb_notify.o
2 obj-$CONFIG_FB_FFB) += sbuslib.o
3 obj-y += omap2/
4 obj-$(CONFIG_VT) += console/
5 fb-y := fbmem.o fbmon.o
6 obj-$(CONFIG_FB) += fb.o
7 obj-$(CONFIG_FB_CG6) += sbuslib.o
```

(a) drivers/video/Makefile

```
1 obj-$(CONFIG_OMAMP2_VRAM) += vram.o
```

(b) drivers/video/omap2/Makefile

```
1 ifeq ($(CONFIG_FB_TILEBLITTING),y)
2   obj-$(CONFIG_FRAMEBUFFER_CONSOLE)+= tileblit.o
3 endif
```

(c) drivers/video/console/Makefile

Figure 3.2: Examples of Makefiles in KBUILD. The Makefiles illustrate the different types of entries controlling source file compilation in KBUILD.

3.2 Source File Compilation in KBUILD

There are different ways a file can get compiled in KBUILD. In this section, we illustrate the different entries in KBUILD that control the compilation of source code files and contribute to the variability of the Linux kernel. There are certain conventions followed in KBUILD that are enforced through *implicit rules* [98] defined in the Makefiles. For each directory, there is an `obj-y` variable which contains a list of files that are to be compiled and linked. The various entries in the Makefile append more files to this list. All the files in this list (i.e., the value of the `obj-y` variable) are then compiled and built into a `built-in.o` object for that directory. At the end of the build process, all the `built-in.o` objects in the directories are linked into the final product (i.e., the specific kernel variant).

Figure 3.2 shows an example with three Makefiles. The Makefile in Figure 3.2a is in the directory `drivers/video`, and the other two Makefiles (Figure 3.2b and Figure 3.2c) are in two separate subdirectories nested under it: `omap2` and `console` respectively. We will use these Makefiles to illustrate the different KBUILD entries. We divide the entries in KBUILD contributing to the configurability of the kernel into six categories: (1) *non-conditional files*, (2) *conditional files*, (3) *non-conditional directories*, (4) *conditional directories*, (5) *composite objects*, and (6) *executable files*. We discuss each of these entry types in the subsections below using the three Makefiles shown in Figure 3.2.

3.2.1 Non-Conditional Files

In Line 1 of Figure 3.2a, `fb_notify.o` is added to the list of files that will be compiled into the `built-in.o` for that directory. KBUILD's implicit rules state that each .o file should be compiled from a corresponding .c file. In this case, `fb_notify.c` will be compiled into `fb_notify.o`. Since there are no configuration features on Line 1 controlling the compilation of `fb_notify.c`, we know that the compilation of fb_notify.c does not depend on any configuration feature, and will, therefore, always be included in any kernel variant.

3.2.2 Conditional Files

The format obj-$(CONFIG_FEATURE) is used to allow the compilation of certain files only if their respective features are chosen. An example can be seen on Line 2 of Figure 3.2a. Recall from Section 2.1.1 that the features the user selects during the configuration process are stored in the .config file. For example, if the user selects the feature USB, an entry CONFIG_USB = y will be included in the .config file. All Makefiles in the Linux kernel have access to the values in this .config file. Thus, if CONFIG_FEATURE is set to be y, the entry obj-$(CONFIG_FEATURE) evaluates to obj-y and the file(s) it controls will thus be built into the kernel as part of the built-in.o file. If it is set to be m, it will be part of a different list obj-m, and it will be compiled as a loadable module. If this feature is not selected, then it will not be defined in the .config file, and thus the variable name will be obj- which is ignored.

Going back to the example on Line 2 of Figure 3.2a, sbuslib.c will be included in the list of files to be compiled if the feature FB_FFB is selected. Files can also be conditionally added through an ifeq or ifneq statements as shown in Line 1 of Figure 3.2c. This notation indicates the feature, and the value it should or should not be equal to (i.e., y, m or empty) for the statements inside the condition to be visited. In this case, Line 2 will be visited only if FB_TILEBLITTING = y.

3.2.3 Non-Conditional Directories

Directories can also be added to the obj-y list. For example, Line 3 of Figure 3.2a adds the directory omap2 to the obj-y list. This means that a sub-Makefile is being invoked. This tells make to visit the omap2 directory, but does not tell it what to do there. The Makefile in the omap2 directory (shown in Figure 3.2b) is the one that will specify which files from that directory will be compiled. Since there is no configuration feature in the entry on Line 3, directory drivers/video/omap2 will always be visited every time drivers/video is visited.

3.2.4 Conditional Directories

Similar to files, directories can also be conditionally compiled. Line 4 of Figure 3.2a shows such an example where directory console will be visited only if feature VT is selected. Again, the Makefile found in that directory is the one responsible for specifying which files there will be compiled.

3.2.5 Composite Objects

Sometimes, a combination of several source files implement one feature which makes it more convenient to group them into one list. This whole list can then be compiled if the corresponding feature is chosen. These are called composite objects. Line 5 and Line 6 in Figure 3.2a show an example of a composite object and its usage. If `CONFIG_FB` is `y` or `m`, the compiler will go ahead and build the `fb.o` object on Line 6. In this particular example, there is no `fb.c` file in the directory. Therefore, KBUILD checks if a composite object using variables `fb-y` or `fb-objs` is defined instead. This notation is enforced through KBUILD's implicit rules, and these composite objects also serve as lists of files and directories. In this example, `fbmem.c` and `fbmon.c` will be compiled into the `fb.o` object (Line 5) and will then be included in the `obj-y` or `obj-m` list according to the value of `CONFIG_FB` (Line 6).

3.2.6 Executable Files

Special cases include executable files that `make` creates on the host machine for use during compilation [81]. These are part of the `hostprogs-y` variable. In these cases, a `fileName.c` is indicated as part of the executable files if it appears in an entry such as `hostprogs-y += fileName` or conditional ones such as `hostprogs-$(CONFIG_FEATURE) += fileName`. There are other special cases, and other lists in KBUILD (such as `head-y`, `lib-y`, etc.). However, we only describe the major parts of KBUILD related to variability here. For more information, we refer the reader to the KBUILD Documentation [81].

3.3 Extracting Build Space Constraints

In order to analyze the variabiliy in KBUILD, we need to extract the presence condition of each source file. A *presence condition* of a source file determines under which combination of feature(s) this file is compiled. In other words, it is a Boolean expression (involving feature variables) which must be satisfied for the file to compile. In this section, we present our tool, MAKEX, which extracts these presence conditions.

3.3.1 *Makefile Constraint Extractor*: MAKEX

We implement a prototype constraint extractor MAKEX[1] which recursively reads all the Makefiles in the source code directories and generates the corresponding constraints (i.e., file presence conditions). Listing 3 shows an example of such constraints extracted from the Makefiles in Figure 3.2. The notation "<->" indicates that a file will be compiled *if and only if* the corresponding feature(s) are selected. We now explain how these constraints can be extracted. Since some source files are only compiled on certain CPU architectures, we extract a different set of build space constraints for each architecture supported in Linux.

[1] MAKEX is available online at `http://swag.uwaterloo.ca/~snadi/KbuildVariability.html`

Listing 3 Example of file presence conditions extracted from the Makefiles in the `drivers` directory shown in Figure 3.2. These constraints determine the configuration features that each code file depends on.

```
1 video/fb_notify.c
2 video/console/tilebit.c <-> CONFIG_VT & CONFIG_FB_TILEBLITTING & CONFIG_FRAMEBUFFER_CONSOLE
3 video/omap2/vram.c   <-> CONFIG_OMAP2_VRAM
4 video/sbuslib.c      <-> CONFIG_FB_FFB | CONFIG_FB_CG6
5 video/fbmem.c        <-> CONFIG_FB
6 video/fbmon.c        <-> CONFIG_FB
```

MAKEX is implemented in Java and uses text based pattern matching to extract the constraints from the Makefiles. MAKEX searches for the `obj-y` occurrences and the files added to them. For example, Line 1 of Figure 3.2a indicates that `fb_notify.c` is compiled unconditionally. Therefore, MAKEX generates the entry in Line 1 in Listing 3 which means that this file has no constraints.[2]

Line 2 of Listing 3 indicates that `titleblit.c` is compiled only if features `VT`, `FB_TILEBLITTING`, and `FRAMEBUFFER_CONSOLE` are *all* selected. This is because any file in the `console` directory is compiled only if feature `VT` is selected (Line 4 in Figure 3.2a) while file `tileblit.c` itself is compiled only if features `FRAMEBUFFER_CONSOLE` and `FB_TILEBLITTING` are both selected (Line 1 and Line 2 of Figure 3.2c) which means that we have to combine all conditions when extracting the file presence condition. On the other hand, directory `omap2` is unconditionally added in Line 3 of Figure 3.2a which means that we only have to check for the file conditions inside `drivers/video/omap2/Makefile` which is shown in Figure 3.2b. The corresponding constraint is shown on Line 3 of Listing 3 where `vram.c` depends only on `CONFIG_OMAP2_VRAM` according to Line 1 of Figure 3.2b.

Similar to conjunctions resulting from the conditionally nested directories, disjunctions can happen when the same file is added to the `obj-y` list under two different conditions. For example, in Line 2 and Line 7 of the Makefile in Figure 3.2a, file `sbuslib.c` is compiled if either features `FB_FFB` or `FB_CG6` are selected which leads to the disjunction in Line 4 of Listing 3.

Finally, Line 5 and Line 6 of Listing 3 are the result of analyzing the composite object shown in Line 5 and Line 6 of Figure 3.2a. Since the composite object involves two files and depends on feature `FB`, then each file will in turn depend on `FB`.

This section showed what the file presence conditions (i.e., the build space constraints) extracted from the build system by MAKEX look like, and the information these constraints are based on. We next evaluate the coverage rate of MAKEX and its performance.

3.3.2 Evaluation of MAKEX

Evaluation Strategy. To evaluate if MAKEX extracts all the necessary file presence conditions from KBUILD, we use three coverage metrics: (1) *Makefile coverage* which is the percentage of Makefiles

[2]Note that when writing out the constraints, we use the .c extension of the file name.

MAKEX analyzes, (2) *Source file coverage* which is the percentage of .c files MAKEX finds presence conditions for, and (3) *Feature coverage* which is the percentage of KCONFIG features used in KBUILD which also appear in the presence conditions extracted by MAKEX. These metrics are based on the assumption that all Makefiles and source code files are both read and compiled. Since this does not necessarily happen in practice as some source files or directories may be just left behind without being used, the values of these metrics show the lower bound of our coverage. Besides measuring coverage, we also measure the performance of MAKEX.

Coverage Results. We find that MAKEX achieves a 75% Makefile coverage rate which means that it analyzes 75% of the Makefiles present in the kernel. For source files, MAKEX has a 85% source file coverage rate which means that it is able to find presence conditions for 85% of the source files present in Linux. Finally, in terms of the configuration features used in KBUILD, MAKEX has a feature coverage rate of 93%. This means that MAKEX is able to see the effect of 93% of the KCONFIG features appearing in the Makefiles.

Performance Results. MAKEX is implemented in Java and runs in a single thread starting from the Makefile in the root of the kernel's source code directory and recursively reads nested Makefiles as needed. Analyzing all architectures in a single release of the Linux kernel runs in approximately 51s.

Discussion and Limitations. The Linux kernel Makefiles are difficult to analyze statically [1] since they use specialized and complicated syntax to represent special cases and are not consistently structured. Analyzing KBUILD statically while considering the variability implementation in it is even more difficult. The previous section has illustrated the most common entries and patterns responsible for source file compilation in KBUILD. There are other more specialized patterns scattered throughout the Makefiles in KBUILD. For each pattern we recognize, we need to add a new pattern matching function in MAKEX's implementation. However, there are some aspects that are difficult to handle using pattern detection. For example, we do not handle definitions or redefinitions of general Makefile variables using `#define` for example and do not execute external scripts which may be called from the Makefiles. From what we have seen, by through manual understanding, the frequency of such cases in the parts of the Makefiles that deal with KCONFIG features are low. However, the limitations of not handling all KBUILD's notation explains why we do not achieve 100% coverage rates. Makefiles we miss may be a result of them being added through a syntax we do not support. Source files we miss may be a result of not analyzing some of the Makefiles or because the source file is added to the list of compiled files using a syntax we do not support. Additionally, some source files get indirectly included rather than explicitly compiled in the Makefile (discussed more in Chapter 6). Although we do not get 100% coverage rate, we have manually verified several of the extracted file presence conditions to make sure they are correct which gives us confidence in the correctness of the extracted constraints.

The fact that MAKEX's implementation is very simple allowing it to analyze a system as large as the Linux kernel in under a minute suggests that it can scale well. However, as discussed, its limitations lie in not being a complete Makefile parser and the fact that it is only able to detect certain KBUILD patterns. However, our purpose is to explore and clarify the role of the build system in the variability implementation in Linux, and not to provide a comprehensive tool which makes MAKEX a suitable for our purposes especially since it provides reasonable coverage rates.

3.4 KBUILD Variability Metrics

In this section, we investigate the extent of the role played by KBUILD in the variability implementation in Linux. To do so, we need to measure the variability in KBUILD and compare it to the rest of the system, where applicable. In this section, we explain some of the metrics we develop to quantify the variability in KBUILD.

To the best of our knowledge, there are no standard metrics to measure variability and its complexity. Some metrics were introduced by Liebig et al. [69] to measure CPP variability in code. We adapt some of these metrics (NOF, SD, TD, and GRAN) for measuring variability in KBUILD and also introduce some of our own (POF, POCCF, POCCD) as follows.

3.4.1 Number of Features (NOF) and Percentage of Features (POF)

The variability in Linux arises from its configuration features. The set of features (*K*) are defined in the KCONFIG files and control the final compiled kernel in one of two ways: in the code space (*C*) through cpp directives or in the build space (*B*) to control source file compilation. Some features may be used in neither or in both spaces. Figure 3.3 shows how four categories of feature uses arise from this setup as follows.

1. *Code Space Only*: This is the set of KCONFIG features that only appear in the code space.
2. *Build Space Only*: This is the set of KCONFIG features that only appear in the build space.
3. *Code & Build Spaces*: This is the set of KCONFIG features that appear in *both* the code space and the build space.
4. *Configuration Space Only*: This is the set of features that are not used in either the code nor the build spaces, but are used internally within KCONFIG to support dependency constraints between other features

We use the metrics *Number of Features* (NOF) and *Percentage of Features* (POF) to measure number and percentage of features in each of these four categories as follows:

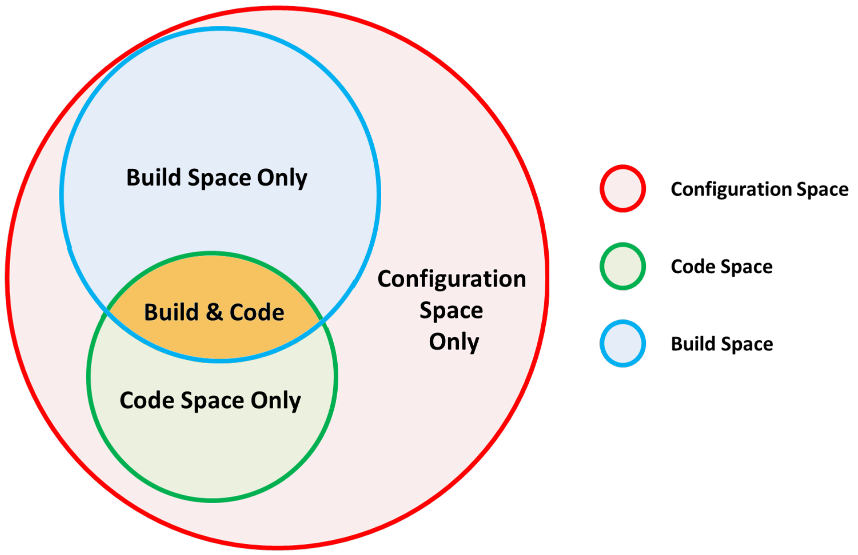


Figure 3.3: Four categories of usage of the features defined in the configuration space (i.e., KCONFIG files)

1. **NOF**$_{K-C-B}$ (or **POF**$_{K-C-B}$): number (percentage) of features that are defined in KCONFIG and *only* used there.
2. **NOF**$_{C-B}$ (or **POF**$_{C-B}$): number (percentage) of KCONFIG features *only* used in code space.
3. **NOF**$_{C\cap B}$ (or **POF**$_{C\cap B}$): number (percentage) of KCONFIG features used in *both* code and build spaces.
4. **NOF**$_{B-C}$ (or **POF**$_{B-C}$): number (percentage) of KCONFIG features *only* used in build space.

3.4.2 Percentage of Conditionally Compiled Files (POCCF) and Percentage of Conditionally Compiled Directories (POCCD)

The POCCF and POCCD metrics measure the percentage of files and directories, respectively, in KBUILD that are conditionally compiled based on some feature selection. Such a metric illustrates the level of variability present in the build system in terms of what fraction of files are conditionally compiled versus being compiled by default in every variant.

3.4.3 Scattering Degree (SD) and Tangling Degree (TD)

We use the *scattering degree* (SD) and the *tangling degree* (TD) to quantify feature usage in KBUILD. The scattering degree of a feature is its number of occurrences in different file presence conditions in KBUILD.

For example, if feature `FOO` controls the compilation of two different files, `foo.c` and `foo2.c`, then it appears in two different file presence conditions and its scattering degree is two. Conversely, the tangling degree is the number of different features that occur in a file presence condition. For example, given a file presence condition `FOO` & `BAR` & `FOOBAR`, then the tangling degree of this presence condition is three since it has three distinct features controlling the file. When examining a kernel release, we record a single SD or TD which calculates the average for that release.

3.4.4 Granularity (GRAN)

KCONFIG features used in KBUILD control the compilation of specific source code files as well as whole directories. We consider two levels of granularity of control. At a high level of granularity, a feature controls a directory which generally contains several source files implementing some related functionality, e.g., sound or USB support. We define GRAN_{dir} as the percentage of features used in KBUILD that control directories. For example in Figure 3.2a, feature `VT` on Line 4 is controlling a directory so it will count towards GRAN_{dir}. At a low level of granularity, a feature controls only the compilation of source code files that generally implement a specific part of this functionality. We use GRAN_{file} to measure the percentage of features used in KBUILD that control *only* source code files. For example, feature `FB_CG6` on Line 7 of Figure 3.2a only controls the file `sbuslib.c` so it counts towards GRAN_{file}. Note that high-level granularity features still appear in the presence conditions of source code files. This is because source code files in a directory will not be compiled unless their containing directory is compiled (see Section 3.3).

3.5 Quantifying KBUILD Variability in Linux

We apply the metrics above to 10 recent versions of the Linux kernel (v2.6.37 – v3.6) spanning a period of around one year and nine months. Examining several releases ensures that conclusions we draw are not just specific to one release. It also provides an evolutionary view of KBUILD variability. We divide our results into four questions as follows.

- Q1: How many KCONFIG features does each space use?
- Q2: How many files and directories are conditionally compiled in KBUILD?
- Q3: What granularity do features mostly control in KBUILD?
- Q4: How complex are the build-space constraints?

We present the results of these four questions below. All numbers reported (unless otherwise specified) are the average of the metric being measured over all releases examined

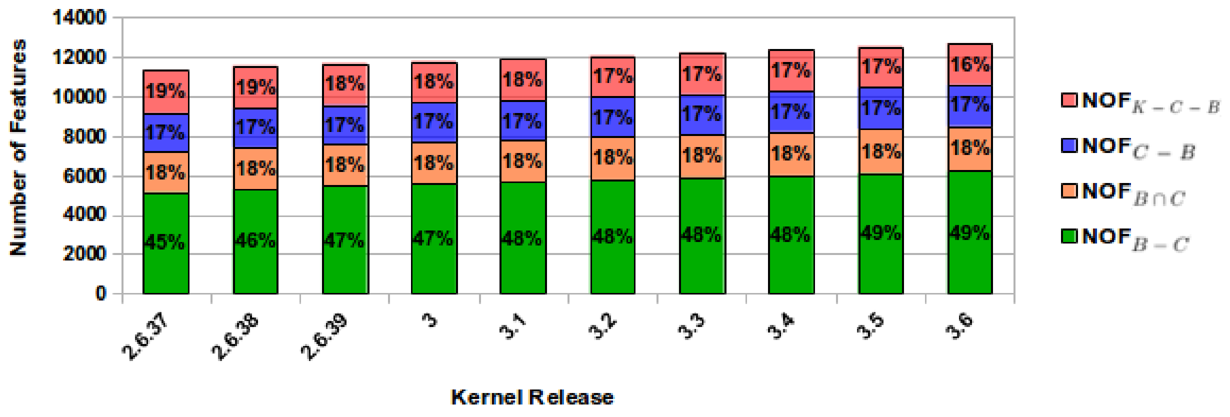


Figure 3.4: Feature usage across Linux releases. NOF is number of features used in: configuration space only ($K-C-B$), code space only ($C-B$), build and code spaces ($B \cap C$), and build space only ($B-C$).

3.5.1 Q1: How Many KCONFIG Features Does Each Space Use?

We use NOF and POF to measure the variability in each space in terms of the number and percentage of features used. Figure 3.4 presents our findings in terms of how KCONFIG features are used according to the four categories illustrated by Figure 3.3 above. Note that the total column height shown in Figure 3.4 represents the total number of features defined in KCONFIG, NOF$_K$. The figure shows that this number is growing in each release.

We now look at the different categories of feature usage shown in Figure 3.4 to understand which part of the system uses most of these features. We find that the percentage of features used in the build space, POF$_B$[3], is 63% versus 35% used in the code space (POF$_C$[4]) which suggests that more configuration control takes place in KBUILD. Over the 10 releases examined, POF$_{B-C}$ is 48% while POF$_{C-B}$ is 17%. This means that a higher percentage of features are *only* used in KBUILD to control whole source file compilation rather than being used in the code to control code block compilation. Additionally, if we look at the percentages shown in each category over the releases, we can see that POF$_{B-C}$ is growing while POF$_{C-B}$ and POF$_{C\cap B}$ represent about the same percentage in all examined releases.

Finding 1: *The majority of* KCONFIG *features are used in the build space. 46% of* KCONFIG *features are used* only *in the build space, and this percentage is growing over time.*

[3]POF$_B$ is obtained by adding POF$_{B-C}$ and POF$_{C\cap B}$ in Figure 3.4

[4]POF$_C$ is obtained by adding POF$_{C-B}$ and POF$_{C\cap B}$ in Figure 3.4

3.5.2 Q2: How Many Files and Directories are Conditionally Compiled in KBUILD?

Our analysis shows that throughout these 10 kernel releases, the percentage of conditionally compiled files (POCCF) is 92%. This means that 92% of the source code files' compilation depends on one or more KCONFIG features. Similarly, the percentage of conditionally compiled directories (POCCD) is 84%. These numbers indicate that most of the source files within Linux are controlled by the user's selection of configuration features and are not compiled by default.

Finding 2: *92% of source files and 84% of source directories are conditionally compiled.*

3.5.3 Q3: What Granularity do Features Mostly Control in KBUILD?

For all 10 kernel releases, we find a GRAN_{file} of 88% which means that 88% of the features used in KBUILD control the compilation of source files only (i.e., low granularity), while GRAN_{dir} is the remaining 12% controlling both files and directories (i.e., high granularity). We use v3.3 in Figure 3.5 to show the number of directories and source files controlled by each configuration feature we find. Each dot on the graph corresponds to a particular configuration feature that appears in KBUILD as found by MAKEX (total of 7,543 features), and shows the number of directories and files it controls. The distribution of feature usage is skewed towards the bottom left which indicates that most of the variability in KBUILD is at a low level of granularity. We find that on average, a feature controls 0.2 directories and 3 source files, and that around 78% of these features control exactly one source file.

To illustrate how this control works, consider the `SCSI` feature (circled on the graph). Its corresponding flag `CONFIG_SCSI` controls 30 directories and 303 files that support the `SCSI` driver. Directories controlled are an example of high-level granularity. Now we will consider the specific bus types within the `SCSI` driver. These are at a lower level of granularity and represent more specific functionalities governed by additional features besides `SCSI`. For example, file `in2000.c` in `SCSI`'s directory implements an `ISA SCSI` host adapter which is only compiled if both `SCSI` and `SCSI_IN2000` flags are turned on. `SCSI_IN2000` is an example of a low granularity feature.

Figure 3.5 also shows a few outliers in the right half of the graph. Two main outliers shown on the top right are features `STAGING` and `SND`. `STAGING` controls the `drivers/staging` directory which contains code that is still under development and has not been finalized for full integration into the kernel yet. The fact that feature `STAGING` is an outlier comes at no surprise since there are 62 directories directly under the `/drivers/staging/` directory apart from the sub directories under each of those. The `staging` directory itself would not be visited unless the `STAGING` feature is selected. The need for the `STAGING` feature to be selected would then be propagated to all subdirectories and files underneath it which results in `STAGING` controlling 84 directories and 531 files. The same applies to `SND` which controls the `SOUND` directory which has 21 main directories.

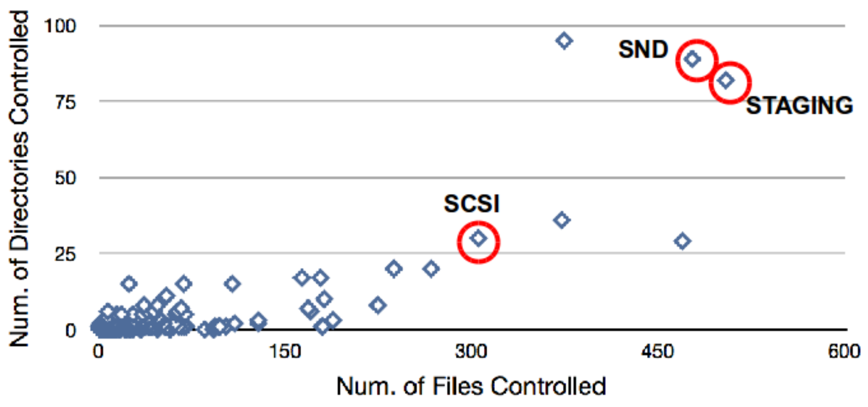


Figure 3.5: Granularity of control of features in KBUILD v3.3. Each point represents a KCONFIG feature used in the Makefiles.

Finding 3: *88% of the features used in* KBUILD *have low level granularity control while 78% control exactly one source file.*

3.5.4 Q4: How Complex are the Build-space Constraints?

We next examine particular aspects of the presence conditions of the source files in Linux to determine how many features usually control the compilation of a source file. We find that although more than half of the KCONFIG features are used in KBUILD, the presence conditions of files are not complex. We find that the tangling degree (TD) of features in the presence conditions is 2. This means that conditionally compiled files have an average of only two configuration features in their presence conditions. These two features are usually the feature controlling the directory (e.g., `SCSI` in the previous example), and then the feature controlling the specific lower level functionality (e.g., `SCSI_IN2000` in the previous example). If we only consider the features that directly control a file (i.e., apart from the feature(s) that control the file's directory), we find that 76% of source files have only one feature in their presence condition. We also found that the scattering degree (SD) of a feature used in KBUILD is 2 which means that on average, a feature appears in two different presence conditions.

Finding 4: *Presence conditions of files in* KBUILD *are not complex. The build space constraints have a tangling degree of 2 features, and features used in* KBUILD *also have a scattering degree of 2. Additionally, 76% of source files have only one feature in their presence condition (apart from the directory control feature).*

3.6 Insights: Variability in KBUILD

The number of features used by the build space shown in Figure 3.4 suggests that almost half of the configuration features defined are used to only control variability in the build system. This shows that when analyzing configurable software, whether to detect inconsistencies or model its configurability, we should not ignore the build system. With the exception of the work by Berger et al. [17], previous work has usually focused on studying the configuration or code space in isolation or studying them together while ignoring the build space [19, 69, 72, 118]. Our quantification of variability here shows that the build system plays an important role in implementing variability and should always be included in such analyses. Additionally, Finding 1 implies that variability in the build space is growing in terms of its usage of KCONFIG features with respect to the rest of the system. We explain this phenomenon as follows. Each time a new source file is added to the Linux kernel source code, an entry must be created in KBUILD so that the file compiles. The majority of new kernel code are device driver implementations, and drivers are usually conditionally compiled since they differ from one platform or machine to another. This means that each time a new driver is added, a feature is added for it in KCONFIG so that the user can select it, and this feature will control the compilation of the implementation source file in KBUILD.

Given Finding 2 above that 88% of source code files are conditionally compiled, we can safely say, that in most cases, whenever a new file is added, a new configuration feature will be used to control it in KBUILD. However, the same does not apply for `#ifdef` variability. A new file may be added with an entry in KBUILD, but this file may not contain any `#ifdef` blocks. Thus, most files will have a conditional compilation entry in KBUILD, but not necessarily conditionally compiled blocks.

If we look at Findings 3 and 4 together, we can deduce that there is commonly a one to one mapping between a feature and the source file it controls. This means that most of the time, a file depends on a single feature, and this feature only controls this file. This is an interesting characteristic of the Linux kernel since it suggests that the user's selection usually directly controls the compilation of whole source files.

Finally, our experience with extracting variability from KBUILD, as well as experiences of more recent work [34], suggests that analyzing build systems to extract file presence conditions is difficult and that more research in that direction is needed to obtain more sound and precise results.

3.7 Related Work

It is our understanding that Berger et al. [17] were the first to discuss variability in the Linux kernel's build system. They showed that the extraction of presence conditions of source code files from Makefiles is feasible where they extracted them for the x86 architecture in Linux and for all of FreeBSD. Our analysis of KBUILD in this thesis is based on all Linux CPU architectures over a longitudinal study, and not solely on the x86 architecture. We also provide a quantification of the variability in KBUILD and show its role in the overall variability of the kernel.

Dietrich et al. [35] have also found that KBUILD v3.1 alone uses almost 50% of the KCONFIG features in Linux. While parts of our work was, unknowingly, done in parallel with theirs, our work is different in that we perform an evolutionary study with several releases of KBUILD. We also adapt previous metrics used to measure CPP variability to customize them for KBUILD. Such metrics allow for standardization of future quantitative analysis of variability. In our work, we also analyze the complexity of constraints and granularity of control within KBUILD. This provides a better overall picture of the variability in KBUILD and how it contributes to the configurability of the whole Linux kernel.

To the best of our knowledge, our work was the first to show the importance of analyzing variability in KBUILD. After that, Dietrich et al. [34] developed their own build system constraint extractor, GOLEM. Their variability extraction approach is based on running an actual kernel build using different configurations and probing it to see which files get built. The advantage of their approach versus a static parsing approach such as ours and that of Berger et al. [17] is that they avoid explicitly analyzing the complicated syntax and special cases that occur in KBUILD. Currently, GOLEM takes about 90 minutes to extract the constraints of a single CPU architecture in Linux versus about 51 seconds for all architectures by MAKEX. Arguably, performance is not everything but such a high running time may affect the practicality of the approach. It is also not clear yet if such a probing based approach would catch all the complex constraints in KBUILD. For example, it would not be able to correctly identify constraints containing negations (i.e., that a feature should not be present) since they rely on incremental addition of features to the feature set and then probing the build system on what files will be built. It is also difficult to correctly identify disjunctions in some cases depending on the order the features get probed by. Thus, it seems that both static and probing/dynamic based approaches have their limitations. It may be the case that other techniques such as symbolic execution [117] may be able to overcome such limitations. However, our goal here is not to determine the most accurate parsing for Makefiles, but rather to clarify the role of build systems in variability support so that it is recognized in future variability research.

3.8 Summary

This chapter explored the role of build systems in variability implementation by using the Linux kernel's build system, KBUILD, as a case study. We extracted the presence conditions of source files from KBUILD (build-space constraints) using our developed extractor, MAKEX, and used this information to provide a quantitative analysis of KBUILD variability. We showed that KBUILD plays a key role in Linux's variability implementation: 63% of configuration features in Linux are used by KBUILD and 92% of source files are conditionally compiled in KBUILD. We have also found that 76% of source files have only one feature in their presence condition, and that 78% of features control exactly one file. This suggests that in most cases, there is a one to one mapping between a user-selected configuration feature and the source file it controls.

In the next chapter, we study if the build-space constraints extracted here (i.e., the file presence conditions) are consistent with the constraints in the rest of the system. We do so by detecting variability anomalies in the form of dead and undead code blocks.

Chapter 4

Block Level Variability Anomalies

Chapter 3 showed that 63 % of the configuration features in Linux are used in the build space. This suggests that the build system plays a major role in the configuration process. Recall that variability in the Linux kernel is scattered across three distinct artifacts: source code files (*code space*), KCONFIG files (*configuration space*), and KBUILD Makefiles (*build space*). Keeping the information in the three spaces consistent is challenging, and conflicts between constraints may occur. Such conflicts may cause anomalies which may lead to decreased reliability and increased maintenance effort.

In the introduction of this dissertation, we briefly introduced how dead code can affect the configurability of the system. We look at dead and undead artifacts (i.e., those which always appear in every variant) in our work both at the code block and code file level. In this chapter, we focus on the code block level, and describe these anomalies in detail as well as how we detect them in the Linux kernel. We use the example in Figure 4.1 throughout this chapter to illustrate block level anomalies. A KCONFIG file is shown in Figure 4.1a. The related Makefile is shown in Figure 4.1b. The related source files are shown in Figure 4.1c, Figure 4.1d, and Figure 4.1e. Similar to the example shown in the introduction chapter, this example describes a cell phone that has several types of camera options.

Chapter Organization. We start by explaining how we detect anomalies at the code-block level in Section 4.1. Section 4.2 then presents the results of applying these detection techniques on the Linux kernel over several releases, and Section 4.3 discusses possible threats to the validity of our work. We provide a summary of the work discussed in this chapter in Section 4.4.

Related Publications. The work in this chapter has been described in the following publications.

Sarah Nadi and Ric Holt. “Mining Kbuild to detect variability anomalies in Linux”. In: *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. Los Alamitos, CA: IEEE Computer Society, 2012, pp. 107–116

Sarah Nadi and Ric Holt. “The Linux kernel: A case study of build system variability”. In: *Journal of Software: Evolution and Process* (2013). Early online view. `http://dx.doi.org/10.1002/smr.1595`

```
config  CAMERA
  bool "Phone has a camera"

config  DSLR_CAMERA
  bool "Camera has a DSLR option"
  depends on CAMERA
```

(a) Kconfig

```
obj−y += options.o
obj−$(CONFIG_DSLR_CAMERA) += dslr_camera.o
obj−$(CONFIG_FLASH) += flash.o
```

(b) Makefile

```
#ifdef CONFIG_CAMERA
 ...
#if !defined(CONFIG_CAMERA)
//Block B1 is dead
#endif
 ...
#if defined(CONFIG_CAMERA)
//Block B2 is undead
#endif
#endif
 ...
#if defined(CONFIG_DSLR_CAMERA)
   && !defined(CONFIG_CAMERA)
//Block B3 is dead
#endif
 ...
#ifdef CONFIG_FLASH
//Block B4 is dead
#endif
```

(c) options.c

```
#if !defined(CONFIG_DSLR_CAMERA)
//Block B5 is dead
#endif
 ...
#if defined(CONFIG_DSLR_CAMERA)
//Block B6 is undead
#endif

#if !defined(CONFIG_CAMERA)
//Block B7 is dead
#endif
```

(d) dslr_camera.c

```
#ifdef DSLR_CAMERA
//Block B8 is dead
#endif
```

(e) flash.c

Figure 4.1: Examples of block level variability anomalies.

4.1 Detecting Block Level Anomalies

Previous work by Tartler et al. [118] discovered anomalies in the Linux kernel by studying the constraints in the source-code files (code space) and KCONFIG files (configuration space) using their UNDERTAKER tool[1]. Their work focuses on finding anomalies at the level of the code blocks using a SAT (satisfiability) solver. However, they do not include the information in the Makefiles (build space) as part of their analysis. Chapter 3 has shown the importance of the build space and its role in implementing variability in the Linux kernel. Thus, we should also consider build-space constraints when detecting these anomalies. To accomplish this, we extend the UNDERTAKER tool to consider the constraints in the build space during the detection of anomalies. We focus on two types of anomalies: dead and undead code blocks. A *dead* code block is a conditional block that is never present in any variant of the system while an *undead* code block is one that is always present in every variant of the system. In our work, we look at a certain type of undead code where a code block is undead if it is always present in every variant whenever its parent code block is also present. This will be described in more detail later.

The UNDERTAKER tool already extracts the presence condition of each code block from the code space, as well as the presence condition of each feature from the configuration space. Recall that a presence condition is a set of constraints (encoded as a Boolean expression) that must be satisfied in order for an artifact to be selected or compiled. For example, the presence condition of Block B4 in Figure 4.1c is `CONFIG_FLASH`. We use our tool, MAKEX (described in Chapter 3) to extract the build-space constraints and add it to the analysis. These extracted constraints are then used in propositional formulas that detect dead and undead code blocks. A SAT solver is used to solve these formulas to detect anomalies. Figure 4.2 illustrates this process. We now explain the propositional formulas we use by first presenting how UNDERTAKER originally worked and then how we modify it.

4.1.1 Original UNDERTAKER Formula

The UNDERTAKER tool extracts preprocessor-based variability in the Linux source code in order to determine the presence condition of each code block [109], and encodes it as propositional logic. Additionally, it identifies the feature dependencies in the KCONFIG files and also represents them as propositional logic [108, 119]. To find anomalies, it combines the constraints from the code space (denoted by *C*) and those from the configuration space (denoted by *K*) for each code block.

Tartler et al. [118] use Formula 4.1 to define a code block, $Block_N$ (B_N), as dead if there is never a case where it can be selected. In terms of propositional logic, this means that we can never satisfy the combination of constraints imposed by the code and configuration space and have $Block_N$ present at the same time [118] as shown in Formula 4.1.

$$Dead_{B_N} = \neg sat(Block_N \wedge C \wedge K) \tag{4.1}$$

[1]http://vamos.informatik.uni-erlangen.de/trac/undertaker

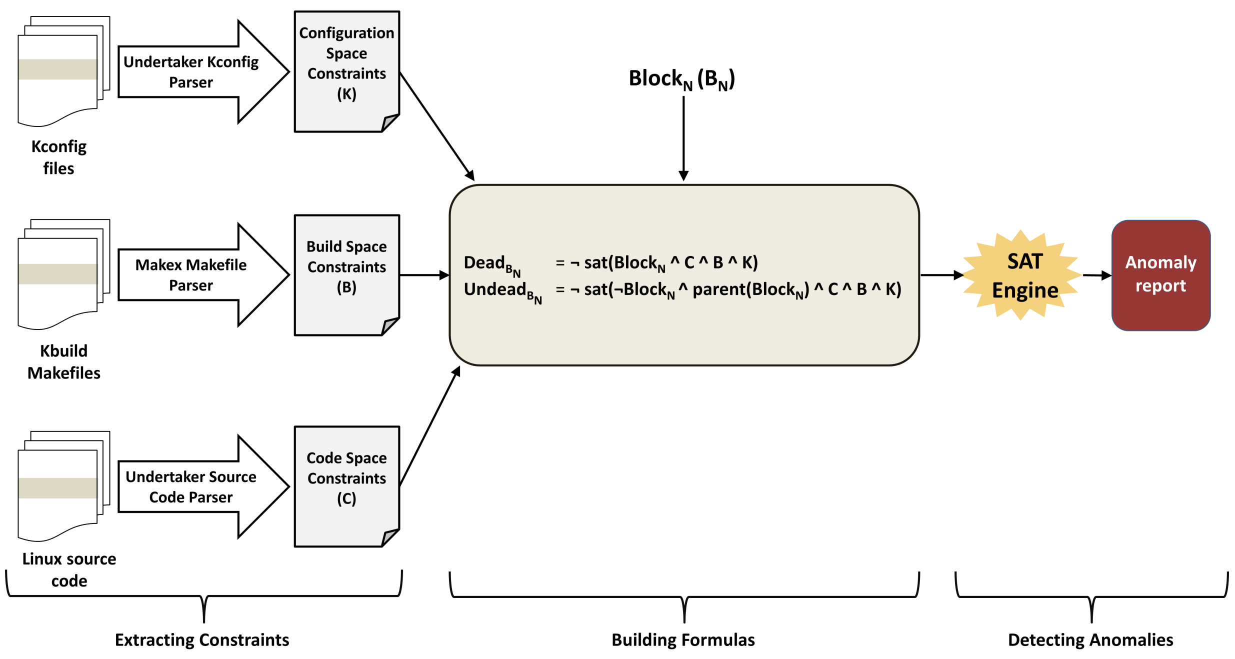

Kconfig files
Undertaker Kconfig Parser
Configuration Space Constraints (K)
Kbuild Makefiles
Makex Makefile Parser
Build Space Constraints (B)
Linux source code
Undertaker Source Code Parser
Code Space Constraints (C)
Block$_N$ (B$_N$)
Dead$_{B_N}$ = ¬ sat(Block$_N$ ^ C ^ B ^ K)
Undead$_{B_N}$ = ¬ sat(¬Block$_N$ ^ parent(Block$_N$) ^ C ^ B ^ K)
SAT Engine
Anomaly report
Extracting Constraints
Building Formulas
Detecting Anomalies

Figure 4.2: Block level anomaly detection process

Similarly, Formula 4.2 defines an undead code block, $Block_N$ (B_N), as one that is always present whenever its parent block is present [118]. If this is the case, then this code block's presence is not really variable since it always gets compiled if its parent block is compiled. This is shown in Formula 4.2 where a block is undead if it can never be deselected in the presence of its parent while satisfying the combination of constraints in the code and configuration space.

$$Undead_{B_N} = \neg sat(\neg Block_N \wedge parent(Block_N) \wedge C \wedge K) \tag{4.2}$$

We would like to point out that generally, dead and undead code can occur in any software system. However, in our context, we focus on parts of the code that should be *conditionally* compiled according to some feature selection, but end up never or always being a part of every variant. Thus, the anomaly arises from the conditional compilation being different from what is specified in the code by the developer.

To illustrate the use of the equations above, we refer to the example in Figure 4.1 and look at file `options.c` shown in Figure 4.1c. Based on Equation 4.1 above, Tartler et al.'s analysis would detect block B1 on Line 4 as dead because the code constraints themselves have a conflict since the parent block depends on feature `CAMERA` being *enabled*, while block B1 depends on feature `CAMERA` being *disabled*. This would cause Equation 4.1 to be satisfiable which means that block B1 is dead. On the other hand, based on Equation 4.2, block B2 on Line 8 is undead since it will always be compiled if its parent block is compiled since they both depend on the same feature `CAMERA`. This means that Equation 4.2 is satisfied since we can never find a solution where the parent is selected but the child (i.e., block B2) is not. Since the conflict in both examples happens in the code constraints only, we call this category of anomalies *code anomalies*.

Code anomalies might be easier to catch since the information is all in one place (i.e., the code file). Things become harder when information from another space is introduced. Looking again at file `options.c` shown in Figure 4.1c, we find that block B3 is also dead. However, this time it is not because of a direct conflict in the code. Instead, the conflict happens between the presence condition of the code block and that of feature `DSLR_CAMERA` in the KCONFIG file in Figure 4.1a. The configuration space specifies that `DSLR_CAMERA` depends on `CAMERA` (Line 6 of Figure 4.1a) while the presence condition of block B3 requires that `DSLR_CAMERA` is selected and `CAMERA` is not. Since this can never happen (i.e., it is prevented by the configuration space), this code block will always be dead. Since the conflict in Equation 4.1 will arise from both the constraints in the code space and those in the configuration space, we call this category of anomalies *code-configuration anomalies*. The same can be applied for undead code, but we do not show an example of it to avoid redundancy.

Finally, there are special types of conflicts in code-configuration anomalies that may arise from *missing* feature definitions. That is, a feature appears in the boolean formula, but has no definition in KCONFIG and can therefore never be selected resulting in it always being `false` in the formula. Line 18 in Figure 4.1c shows such an example. Here, block B4 depends on `FLASH`. We would normally then look for the dependencies of feature `FLASH` in KCONFIG. However, in this example, there is no definition of `FLASH` in the KCONFIG file in Figure 4.1a meaning that `FLASH` is always false (unselected). Therefore, a conflict happens because we need `FLASH` to be selected, but since it is not defined in KCONFIG, it is always deselected. We call this category of anomalies *code-configuration missing anomalies*.

4.1.2 Modified Formula with Build Space

The analysis at the code-block level in UNDERTAKER does not consider the constraints enforced in the Makefiles (the build-space constraints). In the examples we discussed above, this would not make a difference since file `options.c` is unconditionally compiled as shown in Line 1 of the Makefile in Figure 4.1b. However, if the file is conditionally compiled, which we know happens in 92% of the cases as shown in Section 3.5, then the situation might be different as we show below.

In our analysis, we add the build constraints, and modify Formulas 4.1 and 4.2 to those shown in Formulas 4.3 and 4.4. We denote the build-space constraints as B. By adding the build-space constraints, we use Formula 4.3 to define a code block as dead if it can never be present while satisfying the code and configuration constraints along with the build constraints.

$$Dead_{B_N} = \neg sat(Block_N \wedge C \wedge B \wedge K) \tag{4.3}$$

Listing 4 Boolean formula corresponding to Equation 4.3 to detect that block B7 in Figure 4.1 is dead.

```
1 B7 &&
2 B7 <-> !CONFIG_CAMERA &&
3 CONFIG_DSLR_CAMERA &&
4 CONFIG_DSLR_CAMERA -> CONFIG_CAMERA
```

Similarly, we define a block, $Block_N$ (B_N), as undead in Formula 4.4 if we can never find a case where $Block_N$ is not present, but its parent is present while still satisfying the constraints in all three spaces.

$$Undead_{B_N} = \neg sat(\neg Block_N \land parent(Block_N) \land C \land B \land K) \tag{4.4}$$

We refer back to the example in Figure 4.1 and look at file `dslr_camera.c` shown in Figure 4.1d while considering the new formulas we introduced here. In the original UNDERTAKER analysis, no anomalies would be detected for blocks B5 and B6 on Line 2 and Line 6 respectively. However, if we consider the presence condition of the `dslr_camera.c` shown in Figure 4.1b, we find that the whole file is compiled if feature `DSLR_CAMERA` is selected. Thus, blocks B5 and B6 are dead and undead respectively since there is no point in checking `DSLR_CAMERA` again. In this case, since the conflicts in Equation 4.3 and Equation 4.4 arise from looking at the block presence condition as well as the file's presence condition, we call this category of anomalies *code-build anomalies*.

Given the above example, we can imagine that things would even become more complicated, and harder to detect manually, when the conflict arises from all the three spaces. Block B7 on Line 10 in the same file shows such an example. Block B7 depends on feature `CAMERA` *not* being selected. However, given the build-space constraints, we know that file `dslr_camera.c` is only compiled if feature `DSLR_CAMERA` is selected (Line 2 in Figure 4.1b). Additionally, given the configuration space constraints, we also know that feature `DSLR_CAMERA` can only be selected if `CAMERA` is selected. This means that there is a conflict in block B7's presence conditions since if the file is compiled, we know that `CAMERA` is also selected which means that the presence condition of B7 can never be satisfied resulting in B7 being dead. We show an example of what the Boolean formula for detecting this anomaly would look like in Listing 4. Note here how the file presence condition (`CONFIG_DSLR_CAMERA`) is enforced in the formula on Line 3. Since this anomaly arises from conflicts between all three spaces, we call this category of anomalies *code-build-configuration anomalies*. The same can be applied for undead code, but we do not show an example of it to avoid redundancy.

Similar to the code-configuration missing category, missing features can cause a conflict in the code-build-configuration category. Block B8 on Line 2 of Figure 4.1e shows such an example. In this case, block B8 depends on `DSLR_CAMERA` from the code, but it also depends on `FLASH` from the Makefile in Figure 4.1b. As we have seen before, feature `FLASH` has no definition in KCONFIG, and is therefore always deselected. Thus, block B8 is dead because its dependencies cannot be satisfied according to Equation 4.3. We call this category of anomalies *code-build-configuration missing*.

	Category	Description
Logical	***code***	Conflicting code constraints.
	code-configuration	Code space constraints are in conflict with configuration space constraints.
	code-build	Code space constraints are in conflict with build-space constraints.
	code-build-configuration	The combination of constraints in the three spaces are conflicting.
Referential	***code-configuration missing***	Code space constraints are in conflict with configuration space constraints because certain features used in the code are not defined in the Kconfig files and are, therefore, always false.
	code-build-configuration missing	The combination of constraints in the three spaces are conflicting because certain features used in the build constraints are not defined in the KCONFIG files, and are therefore always false.

Table 4.1: Categories of code block variability anomalies.

In general, the term *referential anomalies* describes all anomalies caused by missing feature definitions while the term *logical anomalies* describes the anomalies caused by direct Boolean conflicts in the formulas [118]. To recap, Table 4.1 summarizes the three categories of anomalies we discussed in this subsection (code-build, code-build-configuration, and code-build-configuration missing) as well as the three discussed in Section 4.1.1 (code, code-configuration, and code-configuration missing). We refer back to these categories when presenting the results of detecting anomalies in the Linux kernel in the next section.

4.2 Results: Block Level Anomalies in the Linux Kernel

We now want to determine if adding the build-space constraints when detecting code block anomalies makes a difference. In other words, we explore whether using Equation 4.3 instead of Equation 4.1 and Equation 4.4 instead of Equation 4.2 makes a difference in the detected anomalies. To do so, we apply our analysis to 10 recent releases of the Linux kernel, v2.6.37 – v3.6. Analyzing multiple releases ensures that the results are not specific to one release. We analyze all the source code on all architectures of each release with the exception of the `staging` directory to avoid skewing our results. The `staging` directory contains code that is still under development and will thus likely contain more anomalies than that in the revised code in the other directories.

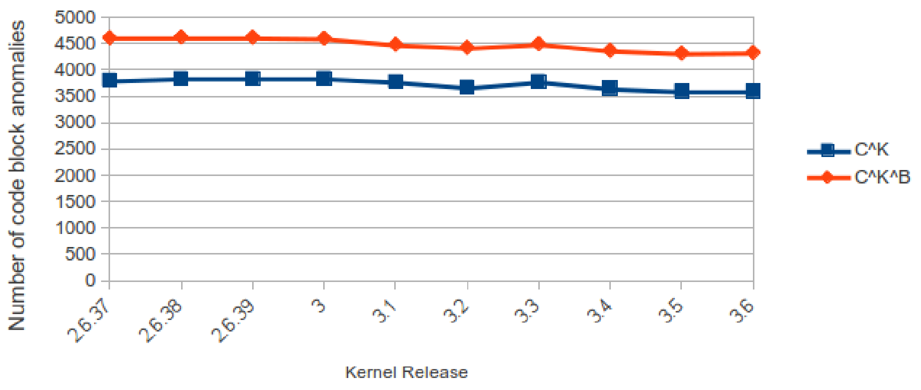


Figure 4.3: Total code-block anomalies detected with and without the build space (B)

We begin by running the original UNDERTAKER tool version 1.3[2] (i.e., using $C \wedge K$, but not B, as shown in Equations 4.1 and 4.2) over the ten releases examined. Then, we run our modified version of the tool (i.e., using $C \wedge K \wedge B$ as shown in Formulas 4.3 and 4.4) over the same 10 releases. We then compare the anomalies detected in each case and collect statistics about them. We provide these statistics below as well as some illustrative examples of the anomalies we detect. Note that we analyze all architectures in the kernel and report a block as dead or undead if it is problematic on all architectures.

Figure 4.3 shows the total number of code blocks having anomalies detected with and without considering the build space (B). We can see that with all three spaces, $C \wedge B \wedge K$, we detect more anomalies than with just $C \wedge K$. This suggests that the constraints in the build space are related to those in the other spaces (i.e., share configuration features and dependencies) and that conflicts often exist.

We are interested in examining the additional anomalies that are detected when the build-space constraints are added (i.e., the categories in Table 4.1 which involve the build space). The anomaly category can be identified since we incrementally add the constraints to the equation to determine when the conflict takes place. This can also be done through an UNSAT core although it may be difficult to identify the source of the problematic clause. Figure 4.4 focuses on these additional anomalies identified and shows the percentage of additional code block anomalies (both dead and undead) detected in each of the three categories involving build constraints (code-build, code-build-configuration, and code-build-configuration missing). The figure shows that when we enhance the block level analysis with build constraints, we detect an average of 20% additional anomalies when compared to just using the code and configuration constraints.

Throughout the 10 releases shown in Figure 4.4, we can see that the anomalies caused by conflicts between all three spaces (code-build-configuration) constitute most of the additional anomalies detected. Since detecting this category of anomalies requires solving a complex satisfiability formula, it suggests

[2]In our CSMR'12 paper describing this [85], we used UNDERTAKER 1.1, while in the extended journal version in JSEP [86], we used version 1.3. When using version 1.3, we do not use the GOLEM tool that was developed after our initial work.

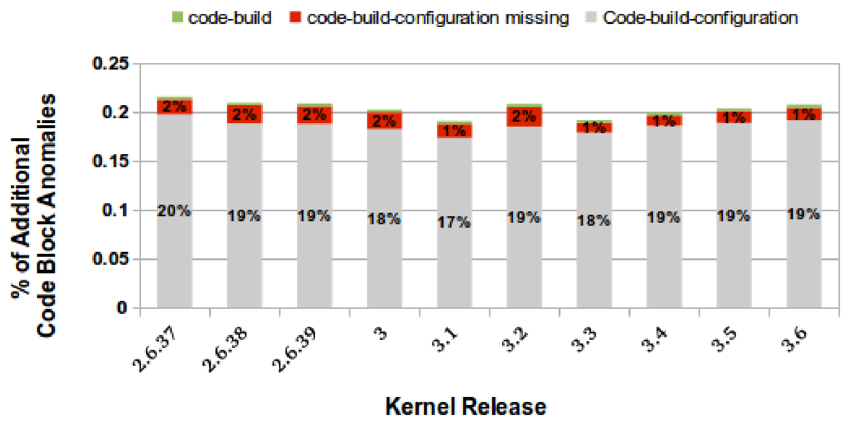


Figure 4.4: Percentage of additional anomalies caused by adding the build-space constraints to the analysis. The code-build-configuration represents the highest percentage of additional anomalies detected, while the code-build represents the smallest percentage of anomalies detected.

that these anomalies are hard to find manually by the developer and that having automated tools to detect them is important. In order to understand the nature of these additional anomalies, we provide illustrating examples from each category.

4.2.1 Analysis of Code-Build Anomalies

Code-build anomalies are a result of a direct conflict between the code constraints and the constraints in the Makefiles. Over the 10 releases examined, we find only 11 distinct code-build anomalies. We find that in some cases, these dead blocks are intentional by the developers so that they mark invalid feature configurations in the code. For example, two dead blocks contain code like `#error invalid SiByte UART configuration` and `#error unknown platform`. This means that the developers are aware that the feature combinations enabling these blocks should never happen and mark them with the `#error` to force the preprocessor to stop. The remaining dead and undead code-build block anomalies involve actual code.

We investigate two of these anomalies which occur in file `arch/sparc/kernel/jump_label.c`. According to the build constraints, this file is compiled only if `SPARC64` is selected. Within the file, there is a code block that does one thing if `SPARC64` is selected and another if it is not. However, since the file will not be compiled without this feature in the first place, then the block depending on it is always selected (i.e., undead), while the other block is never selected (i.e., dead). We submitted a patch[3] to

[3]`http://patchwork.ozlabs.org/patch/164254/`

```
config PCCARD
  bool "PCCard support"
  depends on HOTPLUG

config PCMCIA
  bool "16-bit PCMCIA support"
  depends on PCCARD
```

(a) drivers/pcmcia/Kconfig

```
obj-$(CONFIG_PCMCIA) += ds.o
```

(b) drivers/pcmcia/Makefile

```
#ifdef CONFIG_HOTPLUG
//B1
...
static int pcmcia_bus_uevent(...){...}
...
#else
...
//B2
static int pcmcia_bus_uevent(...){...}
#endif
```

(c) drivers/pcmcia/ds.c

```
B2 &&
( B1 <-> CONFIG_HOTPLUG ) &&
( B2 <-> ( !(B1) ) ) &&
( (CONFIG_PCMCIA) ) &&
( CONFIG_PCMCIA -> CONFIG_PCCARD) &&
( CONFIG_PCCARD -> CONFIG_HOTPLUG)
```

(d) Anomaly formula

Figure 4.5: Example of a code-build-configurationdead block anomaly found in the Linux kernel.

the Linux kernel developers to remove this unnecessary check, but one of the developers replied that the check is there so that 32-bit support is easy to add in the future if someone wants to do that. This response indicates that developers might intentionally leave dead/undead code behind for future anticipated maintenance. On the other hand, we submitted another patch[4] for a different dead code-build block in file `arch/m68k/sun3/prom/init.c`. One of the Linux developers accepted the patch stating that this dead code has been copied from elsewhere but it is actually not relevant to the functionality here and shall be removed [5].

4.2.2 Analysis of Code-Build-Configuration Anomalies

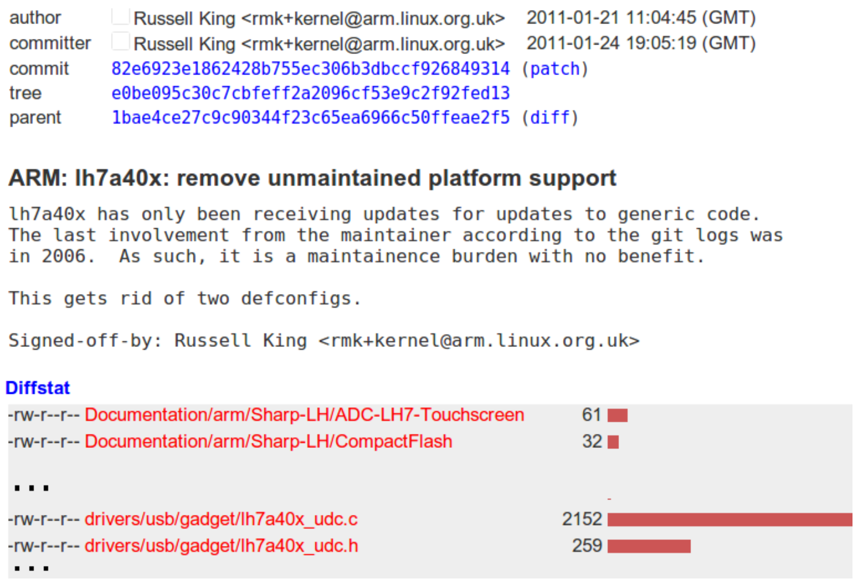
Anomalies in the code-build-configuration category are caused by a conflict involving all three spaces. This category differs from the previous one in that it is not caused by conflicts of direct dependencies in the code and build spaces, but conflicts caused by indirect dependencies that are exhibited in the configuration constraints.

Figure 4.5 provides an example of a code-build-configuration anomaly found in Linux[6] As shown in the Makefile in Figure 4.5b, the code file `ds.c` depends on `PCMCIA`. Code block $B1$ depends on `HOTPLUG`

[4]`https://lkml.org/lkml/2012/6/20/453`

[5]Change was pushed upstream on August 2012: `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=587a9e1f95794c05419d3bdb4c409a3274849f93`

[6]Snippets have been slightly modified for simplicity and better illustration.



```
author     Russell King <rmk+kernel@arm.linux.org.uk>   2011-01-21 11:04:45 (GMT)
committer  Russell King <rmk+kernel@arm.linux.org.uk>   2011-01-24 19:05:19 (GMT)
commit     82e6923e1862428b755ec306b3dbccf926849314 (patch)
tree       e0be095c30c7cbfeff2a2096cf53e9c2f92fed13
parent     1bae4ce27c9c90344f23c65ea6966c50ffeae2f5 (diff)
```

ARM: lh7a40x: remove unmaintained platform support

```
lh7a40x has only been receiving updates for updates to generic code.
The last involvement from the maintainer according to the git logs was
in 2006.  As such, it is a maintainence burden with no benefit.

This gets rid of two defconfigs.

Signed-off-by: Russell King <rmk+kernel@arm.linux.org.uk>
```

Diffstat

```
-rw-r--r-- Documentation/arm/Sharp-LH/ADC-LH7-Touchscreen     61
-rw-r--r-- Documentation/arm/Sharp-LH/CompactFlash            32

...
-rw-r--r-- drivers/usb/gadget/lh7a40x_udc.c                 2152
-rw-r--r-- drivers/usb/gadget/lh7a40x_udc.h                  259
...
```

Figure 4.6: Commit showing the removal of unused code to reduce the maintenance burden. This commit removes the file `drivers/usb/gadget/lh7a40x_udc.c` which had 11 dead code-build-configuration anomalies

while code block $B2$ depends on `!HOTPLUG` as shown in the code (Figure 4.5c). In the KCONFIG file in Figure 4.5a, we see that `PCMCIA` depends on `PCCARD` which in turn depends on `HOTPLUG`. This means that given the file is compiled, block $B2$ can never be selected since `HOTPLUG` will always be enabled for the file to compile. This is shown in the boolean formula illustrated in Figure 4.5d which is based on Equation 4.3.

When examining the anomalies, we notice that some of this dead code is just unused code. We give an example for such a case. The file `drivers/usb/gadget/lh7a40x_udc.c` has 11 dead code-build-configuration anomalies reported in v2.6.37 because of some conflict in the constraints among the three spaces. The commit in Figure 4.6 removes this file two releases later. The interesting part here is the developer's comments when removing the file suggesting that dead code left behind in the code are indeed a maintenance burden. This suggests that even if not necessarily exhibiting errors, automatically detecting these code anomalies is useful. This is especially true in cases where manual inspection of the code will not easily identify a code block as dead.

4.2.3 Analysis of Code-Build-Configuration Missing Anomalies

Missing features in this context are those that appear directly or indirectly from the presence condition of a code block or file, but have no definition in KCONFIG [118]. We can see in Figure 4.4 that only a few of the additional detected anomalies fall in this category. We provide one example here. In the file `drivers/spi/spi-stmp.c`, our analysis reports four dead blocks because of missing features (i.e., in the code-build-configuration missing category). The reasoning behind the anomalies can be explained as follows. The build constraints indicate that `SPI_STMP3XXX` needs to be defined for the source file to compile. However, `SPI_STMP3XXX` depends on another feature `ARCH_STMP3XXX` which has no KCONFIG definition. Thus, all the conditional code blocks in the file are reported as code-build-configuration missing since the file itself will never be compiled due to the missing feature definition.

We note that when adding the build space to the analysis, there are only a few *additional* anomalies caused by missing feature definitions (an average of around 1.5 %, see Figure 4.4). However, the block level anomalies that are caused by missing feature definitions in general account for an average of 48% of the detected block level anomalies. This suggests that developers use the correct defined features to guard the compilation of source files in the build system, but this is not always the case for the features used inside the code.

4.3 Threats to Validity

We now discuss the possible threats to the validity of the work presented in this chapter.

Internal Validity. Since we are extending the UNDERTAKER tool, any shortcomings in the original analysis will be reflected in our analysis. The UNDERTAKER tool is an ongoing work, and its authors are constantly updating it. Therefore, running the analysis with a different version could possibly yield a different number of anomalies but should not change the conclusions drawn.

Any problems with our extraction of the build-space constraints will also affect the results. There are certain parts of the Makefiles that are hard to parse using textual pattern detection as discussed in Chapter 3. In our work so far, there are some of these aspects which we ignore such as `#define` used with the Makefiles to define additional variables that are later used. We also do not execute external scripts called from within the Makefiles. Additionally, since we only analyze 75% of the Makefiles (See Section 3.3.2), we may miss clauses (disjunctions or conjunctions) in the extracted constraints resulting from the unanalyzed files.

Some of the anomalies we discover may not necessarily reflect errors. We choose to use the term *anomaly* precisely for this reason which is similar to the idea of *bad code smells* [46]. A dead artifact may exist due to bad maintenance, and an undead artifact may be used as a form of double checking that certain conditions actually hold. In both cases, we believe that developers should still be aware of such anomalies since they are potential sources of errors and undesired behavior. However, in order to address the fact that developers may intentionally leave dead or undead artifacts behind, a whitelist approach can be adopted to allow developers to remove certain files from consideration.

External Validity. We only examine one software system, the Linux kernel. However, the Linux kernel is the largest configurable open source software system available, supporting build-time variability. Our results, which conclude that considering the build constraints can lead to additional detected anomalies, do not necessarily apply to other systems. Linux's build system, KBUILD, is complex and unique in terms of the customized notation it uses. However, there are many other configurable systems that use a similar structure for their build systems (e.g., BusyBox and BuildRoot). Although we do not attempt to generalize our results beyond Linux, we believe that our work provides interesting findings which can be used to guide the study of variability in other build systems.

4.4 Summary

In this chapter, we have shown how adding the build-space constraints, extracted as file presence conditions, can affect the detection of variability anomalies. We have focused here on block-level anomalies where we detect dead and undead code blocks. Our results show that additional code block anomalies can be detected when the build-space constraints are considered suggesting that the constraints in the three spaces may not always be consistent.

Chapter 5

Evolution of Block Level Variability Anomalies

In Chapter 4, we showed that block-level anomalies exist due to conflicts between the constraints in the three spaces. Tools that detect these anomalies, such as UNDERTAKER, are therefore useful to improve maintenance. However, in order to avoid these anomalies from happening in the first place, we need to analyze how they get introduced and how they get fixed. In this chapter, we study the origin of block-level variability anomalies in the Linux kernel and how they get fixed.

To study the evolution of block-level anomalies, we focus on referential anomalies (those with missing feature definitions as discussed in Chapter 4) as they represent almost half of the detected variability anomalies (see Section 4.2.3) and their evolution can also be automatically analyzed. As a reminder of what a referential anomaly is, consider the following code:

```
#ifdef CONFIG_USB_SUPPORT
  // Block B1
#endif
```

Let us look at the code block B1 between the `#ifdef` and the `#endif` CPP statements. This block is guarded by the feature `CONFIG_USB_SUPPORT`, so it will not be compiled unless this feature is selected resulting in the following clause.

$$B1 \leftrightarrow CONFIG\_USB\_SUPPORT$$

Now, assume there is no definition for feature `CONFIG_USB_SUPPORT` in the KCONFIG files. This means that this feature can never be selected. A clause indicating that this feature is always undefined is then added to the Boolean formula as follows.

$$(\text{B1} \leftrightarrow \text{CONFIG\_USB\_SUPPORT})$$
$$\wedge\ (\neg\ \text{CONFIG\_USB\_SUPPORT})$$

Since this formula is not satisfiable, UNDERTAKER detects an anomaly here indicating that block B1 is dead. Based on the result of the SAT checker, UNDERTAKER generates an anomaly report which contains the boolean formula above where the last line contains the missing feature(s) as shown. This is an example where the missing feature is the one the code block directly depends on. It could also be the case that one of the features the block indirectly depends on is missing (see example in Section 4.2.3). For brevity, we use the terms *anomaly* or *variability anomaly* throughout the rest of the chapter to refer to *referential anomalies*. The rest of this chapter focuses on finding how such anomalies get introduced and fixed.

To determine which patterns to look for when identifying causes and fixes of variability anomalies, we first start with an exploratory case study [39]. In this exploratory case study, we analyze an existing set of 106 patches that fix variability anomalies. This set of patches has been submitted to Linux developers by Tartler et al. [118]. The patches received considerable feedback, with over 50% of them being accepted. As a result of studying the responses of developers to these patches, we are able to recognize some patterns causing these anomalies which allows us to develop four research questions which we answer in a confirmatory case study. We analyze several releases of the Linux kernel to determine if the patterns we find generalize to all the anomalies detected in the kernel.

Chapter Organization. We first describe the exploratory case study in Section 5.1. We then describe the procedure followed in the confirmatory case study in Section 5.2 and present its results in Section 5.3. In Section 5.4, we discuss the interpretations of our findings and their implications. We present the threats to the validity of our work in Section 5.5 and conclude with a summary of the work presented in this chapter in Section 5.6.

Related Publications. The work presented in this chapter has also been published in the following paper.

Sarah Nadi, Christian Dietrich, Reinhard Tartler, Richard C. Holt, and Daniel Lohmann. "Linux variability anomalies: What causes them and how do they get fixed?" In: *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. MSR '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 111–120

5.1 Exploratory Case Study

The aim of the exploratory case study is to determine patterns for how anomalies get introduced and fixed. Identifying such patterns allows us to automate the process of searching for the causes and fixes of any detected anomalies. To do so, we need a set of detected anomalies for which we have developer feedback commenting on how they got introduced or how they plan to fix them. Previous work by Tartler et al. [118] has used UNDERTAKER version 1.1 to detect variability anomalies in the Linux kernel v2.6.35 and earlier. As part of that work, they randomly chose 337 referential variability anomalies from those they detected and manually created patches to the Linux kernel to fix these anomalies (see Section 2.1.5 for a description of a patch). This resulted in 106 patches submitted to the Linux mailing lists to solve these 337 anomalies (some patches fixed more than one anomaly). The submitted patches were then reviewed by developers and a response was provided on the mailing list about whether each patch was accepted or not. Thus, this data set fits well with the objectives of our study.

In their work, Tartler et al. [118] did not perform an in depth analysis of the developers' responses they received or the causes behind these anomalies. To determine what causes variability anomalies and how developers fix them, we make use of this unanalyzed data they collected. This exploratory dataset we use consists of the proposed patches as well as the email conversations with developers that followed in response to these patches.

5.1.1 Description of Exploratory Dataset

To explore the dataset, we first determine the proposed fix of each submitted patch. We manually identify the change each patch proposes and classify them into five categories shown in Table 5.1. We now discuss these five categories. The first line in Table 5.1 shows that the majority of the proposed patches (90%) remove dead code caused by the missing features. Patches that remove dead code are important since they remove bad code smells [46] and tend to make the code easier to read and more maintainable. The second type of fix shows that 6 of the proposed patches (6%) rename the missing feature used in the CPP condition in the code to one that is defined in KCONFIG. Two of the proposed patches (2%) remove dead code as well as dead features from KCONFIG. The dead features depend on an undefined feature and could therefore never be selected. For two other patches (2%), the proposed fix removes `#ifdef` checks either because they are redundant checks (i.e., same condition is checked twice) or because the guarding feature is not defined, but the code inside the block is needed. Finally, one patch (1%) removes a dead block that depends on a missing feature while also removing this missing feature from dependency clauses in KCONFIG.

The *status* of a submitted patch describes the developers' response to the proposed patch. We manually study the response of developers to each submitted patch to determine its status as shown in Table 5.2. We find that the proposed patches are either accepted, acknowledged, receive no reply, or rejected. We now discuss each of these cases. The first entry in Table 5.2 shows that roughly half of the 106 patches (51%) were accepted by developers as is. The second entry shows that for 27 of the proposed patches, developers

Table 5.1: Categorization of proposed fixes of 106 submitted patches in the exploratory dataset

Type of Proposed Fix	**Description**	**Count (%)**
Remove Dead Code	Completely delete the dead `#ifdef` guarded code block	95 (90%)
Rename Feature Used in Code	Remove the undefined feature used in the `#ifdef` and use a different, defined feature	6 (6%)
Remove Dead Code & Dead KCONFIG Features	Remove dead code (see above) as well as features from KCONFIG which depend on the undefined feature	2 (2%)
Remove Redundant Checks	Remove `#ifdef` checks which are redundant with respect to the parent `#ifdef` check(s)	2 (2%)
Remove Dead Code & Edit KCONFIG Dependencies	Remove dead code (see above) as well as remove the missing feature from dependency clauses in KCONFIG	1 (1%)
Σ Total		106 (100%)

acknowledge that the code being fixed was indeed dead or undead, but the proposed patch was not applied for one of three reasons: 1) a fix is already being prepared for this problem, either in terms of a scheduled merge or a patch under progress (12 patches), 2) developers want a different fix than the one originally provided in the patch (11 patches), and 3) developers would like to keep the code block as it is because it is used in out of tree development or for reference purposes (4 patches). The third entry in Table 5.2 shows that there were 21 patches (20%) that received no reply, while the last entry shows that only a few patches (4%) were rejected. We classify a patch as rejected if developers do not acknowledge that a problem exists (i.e., that the code is actually dead or undead). This happens in rare cases where although the missing configuration feature(s) causing the anomaly are not defined in KCONFIG, developers set their values by hand to be able to use the corresponding code.

5.1.2 Observations about Exploratory Dataset

From our manual analysis of the set of 106 patches and the responses of the developers to each patch, we make two main observations.

Table 5.2: Categorization of the status of 106 submitted patches in the exploratory data set

Status	Description	Count	(%)
Accepted	Proposed patch accepted and applied as is	54	(51%)
Acknowledged	Acknowledgement of dead/undead code, but patch not applied as is because one of the following:	27	(25%)
Under progress	A fix is already being prepared for this problem, either in terms of a scheduled merge or a patch under progress (12)		
Different fix suggested	Developers want a different fix than the one originally provided in the patch (11)		
Keep code	Developers would like to keep the code block as it is because it is used in out of tree development or for reference purposes (4)		
No Reply	Developers did not respond to submitted patch	21	(20%)
Rejected	Developers do not acknowledge that a problem exists	4	(4%)
	Σ Total	106	(100%)

Observation 1: Feature Names. There are 6 proposed patches that change the name of the feature being used in the CPP code. Out of these patches, four were accepted. Recall that for some of the proposed changes, developers either suggested a different patch to fix the code or indicated that they already have a fix being prepared. If we look at these cases, we find that there are 9 patches that proposed removing the dead code caused by the missing feature(s), but where developers suggested leaving the code in, and guarding it with a different feature that is defined in KCONFIG. Both cases (accepted patches changing the feature name and suggesting a different feature to use) indicate that the dead CPP code block is not useless, but has been mistakenly guarded by an undefined feature causing it to be dead. In four of these cases, we could tell from the developer's comments and the name of the suggested feature that the undefined features being replaced were caused by a *misspelling* or *typo* such as using `CONFIG_CPU_S3C24XX` instead of `CONFIG_CPU_S3C244X` where the X is mistakenly typed instead of the 4. We use the terms misspelling and typo interchangeably throughout this chapter.

Observation 2: Incomplete Patches. In the comments of one of the patches for which developers suggested using a different feature in the CPP condition, their response indicates that the missing feature got renamed in KCONFIG, but developers forgot to rename it in the code. The developers' responses to three other accepted patches removing dead code also suggest that the missing features were retired in previous patches, but the code was not updated to reflect that. These observations lead us to suspect that incomplete patches may be a common cause for variability anomalies. We use the term *incomplete* to indicate that the change was not completely propagated throughout the system.

To confirm this, we look at two types of the proposed patches: those where the problem was acknowledged, but a different patch was suggested and those in which the original patch was renaming the feature being used. These result in 15 patches. We do an in-depth analysis of these 15 patches where we manually study the history of the corresponding anomalies to understand how they got introduced. We find that 8 patches fix anomalies that have existed since the related code block was introduced (i.e., code was dead since inception). On the other hand, 7 of the patches fix anomalies caused by incompletely propagated changes. That is, developers change or remove a feature definition from KCONFIG, but do not correctly propagate the change to the rest of the code.

We provide an example for such a case. One of these anomalies is a code block which is dead because feature `CONFIG_MTD_NAND_AT91_BUSWIDTH_16` is not defined in KCONFIG. After some investigation, we find that there is a previous patch that renames this feature to `CONFIG_MTD_NAND_ATMEL_BUSWIDTH_16` in KCONFIG but does not rename it in the CPP condition, resulting in the code block being dead because it uses an undefined feature. Although we cannot conclude that incomplete patches are a major source of variability anomalies from this small data sample of 15 patches, the data does provide indication for that. This gives us motivation to further examine this in the whole Linux kernel which we do in Section 5.2.

5.1.3 Research Questions

Based on the patterns we observe in the dataset described above, we develop four research questions about how Linux variability anomalies are introduced and fixed.

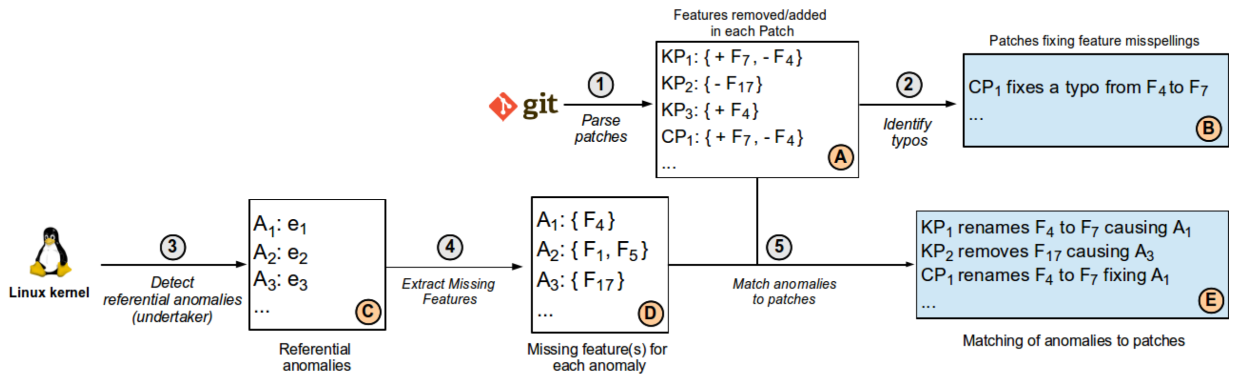
Our first research question, RQ1, is based on Observation 1 which leads us to conjecture that misspellings may cause missing features that in turn cause referential anomalies.

RQ1: *Are misspellings a common cause of variability anomalies?*

Our second research question is based on Observation 2 that several anomalies are caused by previous incomplete patches. Specifically, a patch renames or removes a feature in KCONFIG without renaming/removing all its uses in the rest of the kernel. We call these *incomplete* KCONFIG *patches*.

RQ2: *Are incomplete* KCONFIG *patches a common cause of variability anomalies?*

Our observations from the exploratory dataset are mainly concerned with what causes the anomalies. However, it is also important to know how they eventually get fixed. To explore this, we need to analyze



Features removed/added in each Patch
KP1: { + F7 , - F4}
KP2: { - F17}
KP3: { + F4}
CP1: { + F7 , - F4}
...
A
git
1
Parse patches
2
Identify typos
Patches fixing feature misspellings
CP1 fixes a typo from F4 to F7
...
B
Linux kernel
3
Detect referential anomalies (undertaker)
A1: e1
A2: e2
A3: e3
...
C
Referential anomalies
4
Extract Missing Features
A1: { F4}
A2: { F1, F5}
A3: { F17}
...
D
Missing feature(s) for each anomaly
5
Match anomalies to patches
KP1 renames F4 to F7 causing A1
KP2 removes F17 causing A3
CP1 renames F4 to F7 fixing A1
...
E
Matching of anomalies to patches

Figure 5.1: Protocol for confirmatory case study to identify potential causes and fixes for referential variability anomalies. KP_n: KCONFIG patches, CP_n: CPP patches, A_n: Anomalies, e_n: Boolean expressions, F_n: KCONFIG features.

how long anomalies last in the Linux kernel, and how they get fixed. We therefore raise the following additional questions.

RQ3: *How are variability anomalies fixed?*

RQ4: *How long do variability anomalies remain unfixed in Linux?*

We answer these four research questions in a confirmatory case study in which we examine if the observations from the exploratory case study hold on a larger scale. We present the procedure we follow in the next section.

5.2 Confirmatory Case Study

The goal of our confirmatory case study is to answer the four research questions from Section 5.1.3. In this section, we explain the protocol we use to answer these research questions. We describe the steps we perform to find typos as well as how we match anomalies to existing patches to identify potential causes and fixes as illustrated in Figure 5.1. We analyze the variability anomalies in 10 recent releases of the Linux kernel (v2.6.37–v3.6) which spans a period of almost 1 year and nine months.

Our analysis is mainly implemented through several Python scripts which we run on a machine with two quad-core Intel Xeon 2.67GHz CPUs and 16GB RAM.

5.2.1 Step 1: Extract and Parse Patches

Step 1 identifies and analyzes the patches stored in the GIT repository that are relevant to our analysis. Since most referential anomalies are caused by inconsistencies between the code and configuration spaces (Section 4.2.3), we focus on two types of patches here. *CPP patches* are those that change the CPP guards of conditional blocks. We also define *Kconfig patches* as those that change feature definitions in KCONFIG files. Note that a patch can affect both Kconfig and CPP code at the same time in which case the patch can be classified as both.

Box A in Figure 5.1 shows a representation of our extracted patches where KCONFIG patches are denoted by KP_n and CPP patches by CP_n. For each patch, we identify the features added or removed found through the + and – notation described in Section 2.1.5. For example, the notation $KP_1\{+F_7, -F_4\}$ in Box A means that KP_1 removes Feature F_4 and adds Feature F_7. Similarly, KP_2 removes F_{17} and KP_3 adds F_4 while CP_1 adds F_7 and removes F_4 from some CPP conditions.

It is straightforward to parse KCONFIG patches. This is done by locating feature declarations that are inserted or deleted and transcribing them into plus and minus notation (see Box A). It is more complicated to parse CPP patches. This is done by searching for CPP guarded code blocks (blocks surrounded by `#ifdef`, `#ifndef`, etc.) and identifying each feature F_i used in the CPP condition (the CPP guard). If patch CP_k deletes the line containing the condition, then F_i is also considered to be deleted, so we produce output such as $CP_k : -F_i$. Conversely, if CP_k adds the condition, we produce output such as $CP_k : +F_i$. For CPP patches, we also record the source file in which the CPP condition was added or removed.

If a patch removes one feature F_i and then adds another F_j in the next step, we assume that F_j *renames* F_i. For example, patch KP_1 in Box A renames F_4 to F_7.

The extraction and parsing of patches in Step 1 is performed only once on release v3.7[1] to extract the whole history of the ten Linux releases examined (v2.6.37–v3.6). We extract the KCONFIG patches and CPP patches separately. It takes approximately 39 minutes to extract KCONFIG patches from GIT, and 44 minutes to extract the CPP patches. The results are saved to be queried in Steps 2 and 5.

5.2.2 Step 2: Identify Misspelled Features

Step 2 determines if typos (misspelled features) are a common cause of variability anomalies (RQ1). It does this by locating CPP patches that rename the feature used in the CPP condition. Based on these renames, we develop heuristics to automatically compare the names of the old and new features to determine if the change is apparently correcting a typo.

We consider that the renaming is correcting a misspelling if the first name of the pair is within one or two edit distances from the second name (i.e., a difference of one or two characters) or if the first name is a permutation of the words separated by underscores in the second name (e.g., `CONFIG_USB_SUPPORT`

[1] v3.7 is used to ensure that any fixes that happen to anomalies found in v3.6 are also caught.

vs `CONFIG_SUPPORT_USB`). This automatic classification provides us with a set of CPP patches that could potentially be correcting misspelled features. For example, consider patch CP_1 in Box A. If feature F_4 is classified as an apparent typo of feature F_7, then we classify CP_1 as correcting a misspelling as shown in Box B.

We then manually verify each of these identified patches by looking at the developer's commit messages to judge if the change was actually correcting a spelling mistake or not. This manual verification is necessary to avoid false positives because there are features in Linux which have similar names (e.g., `X86_32` and `X86_64`), but are implementing different functionalities, and so a replacement of one feature with another one may be an intentional logic change and not a typo. We choose to design our analysis this way and not to match missing features in the anomalies to possible misspellings in all defined KCONFIG features, because there would be no way to verify if the identified features are indeed typos or not.

Since this step, Step 2, only depends on the extracted CPP patches, we perform it only once after extracting Linux's history. This takes 1 hr 38 minutes to run. The performance bottleneck here is the algorithms used to detect similar words.

The following steps (3, 4 and 5) are then performed for each kernel release examined to answer RQ2 and RQ3 dealing with patches causing and fixing anomalies.

5.2.3 Step 3: Detect Referential Anomalies Using UNDERTAKER

In Step 3, we use UNDERTAKER version 1.3 to detect referential variability anomalies in the ten releases of the Linux kernel we examine. These referential anomalies are shown in Box C in Figure 5.1 where each anomaly A_k has a boolean expression e_k which is the boolean formula that was not satisfied. Detecting anomalies on the whole Linux kernel using UNDERTAKER takes approximately 45 min for each kernel release using 4 parallel threads.

5.2.4 Step 4: Extract Missing Features

In Step 4, we analyze the boolean formula e_k for each referential anomaly extracted to automatically identify the missing feature causing the anomaly. This allows us to have a list with each anomaly and its corresponding missing features as shown in Box D. As mentioned before, missing features are negated at the end of the formula. In the example given at the beginning of this chapter, we identify `USB_SUPPORT` as the missing feature. Box D in Figure 5.1 shows the information extracted in this step which takes about 3 seconds for each release.

5.2.5 Step 5: Match Anomalies to Patches

Step 5 correlates anomalies with their potential causes and fixes; see Box E in Figure 5.1. The GIT repository for the Linux kernel contains thousands of commits. We, therefore, need to develop heuristics to automatically identify potential commits that may be the cause or fix for an anomaly. We now describe these heuristics.

Our heuristics identify two causes of anomalies: *renaming* and *removal* of KCONFIG features without reflecting these changes in the code. The first two lines of Box E in Figure 5.1 illustrate these two cases. In Box D, Anomaly A_1 is due to the missing feature F_4. Patch KP_1 renames feature F_4 to F_7 in KCONFIG, thus removing the definition of F_4 from KCONFIG, and causing anomaly A_1. Similarly, anomaly A_3 in Box D is due to the undefined feature F_{17}. Patch KP_2 removes F_{17} from KCONFIG making F_{17} a missing feature and causing anomaly A_3. In more general terms, there are two kinds of causes of an anomaly A_k that are due to missing feature F_j:

1. Patch KP_i *removes* feature F_j from KCONFIG.
2. Patch KP_i *renames* feature F_j in KCONFIG.

In both cases, patch KP_i must occur *before* anomaly A_k. If more than one matched patch occurs before the anomaly, we choose the patch closest to the date of the occurrence of A_k.

We also identify four ways that patches can fix anomalies. One of these is illustrated by the third line of Box E in Figure 5.1 which specifies that patch CP_1 renames feature F_4 to F_7 in the CPP condition thus fixing anomaly A_1. A_1 is caused by the undefined feature F_4. CP_1 fixes this by using feature F_7 instead of F_4 in the CPP condition. Recall that the incomplete patch KP_1 renames F_4 to F_7 in KCONFIG without reflecting the change in the code. Thus, CP_1 completes this rename in the code by using F_7 instead of F_4 which fixes anomaly A_1.

Specifically, given anomaly A_k caused by missing feature F_j, we identify four cases where a patch can fix this anomaly as follows (the discussed example is the fourth kind):

1. Patch KP_i *adds* F_j to KCONFIG.
2. Patch KP_i *renames* another feature in KCONFIG to F_j.
3. Patch CP_i *removes* the CPP condition containing F_j.
4. Patch CP_i *renames* F_j in the CPP condition.

All types of fixes essentially aim to ensure that the features used in the CPP conditions have a corresponding definition in KCONFIG. In each of the four kinds of fixes, the matched patch must occur *after* the anomaly A_k. If more than one such fixing patch occurs after the anomaly, we choose the patch closest to the date of the anomaly. Matching anomalies to KCONFIG patches takes approximately 13 seconds for each release, and matching anomalies to CPP patches takes approximately 47 seconds for each release.

5.3 Results of Confirmatory Case Study

We follow the procedure explained in the previous section (see Figure 5.1) to analyze the variability anomalies in releases v2.6.37 to v3.6. We apply Step 1 on release 3.7 to extract all the Linux history until the latest release examined. We extract 10,263 KCONFIG patches and 25,410 CPP patches from the GIT repository. We report the results of our analysis in this section. We structure our results to answer the four research questions and then provide interpretation of our findings.

5.3.1 RQ1: Are Misspellings a Common Cause of Variability Anomalies?

Out of the 25,410 extracted CPP patches, only 1,412 patches rename features in CPP conditions (i.e., the patch changes the feature being used in the condition). From these patches, we use our spelling checker heuristics (Section 5.2, Step 2) to automatically find 203 patches (14%) where the replacement feature seems to be correcting a misspelling of the original feature. We manually verify all 203 patches by checking developers' commit messages to judge if this patch is indeed correcting a misspelling. We are able to confirm that 54 out of the 203 patches (27%) are indeed correcting misspellings (the high number of false positives is due to similar features like `X86_32` and `X86_64` as explained in Step 2 of Section 5.2). This means that only 4% (54 out of 1,412) of CPP patches renaming features are dealing with misspelled features.

Finding 1: *Misspellings are not a common cause of variability anomalies. Only 4% of* CPP *patches changing the feature used in the* CPP *condition are correcting misspellings.*

5.3.2 RQ2: Are Incomplete KCONFIG Patches a Common Cause of Variability Anomalies?

Table 5.3 summarizes the results for the matched anomalies in each release studied. In each release, the table shows the number of anomalies matched to causing and fixing patches. The second column of the table shows the number of referential anomalies detected by UNDERTAKER in each release. The third column shows the number of anomalies which we are able to automatically match to a causing KCONFIG patch in the GIT history. That is, a patch that occurs before the date of this release removes or renames the missing feature in KCONFIG without reflecting this change in the anomalous file. Since a referential anomaly can be due to more than one missing feature, the same anomaly may be matched to several historic patches based on the different missing features. However, we only count the unique number of anomalies for which we could find a causing patch. We note that an anomaly may span multiple releases of the Linux kernel. Since we count the number of matches in each release, and not throughout all release, we avoid multiple countings of the same anomaly.

Table 5.3: Number of referential anomalies in each release that are caused by incomplete KCONFIG patches as well as those fixed by KCONFIG and CPP patches. Percentages are shown in parenthesis.

Release	Referential Anomalies	Anomalies caused by incomplete KCONFIG patches	Anomalies fixed by KCONFIG Patches	Anomalies fixed by CPP Patches
2.6.37	706	56 (8%)	22 (3%)	383 (54%)
2.6.38	688	62 (9%)	21 (3%)	354 (51%)
2.6.39	658	61 (9%)	28 (4%)	317 (48%)
3.0	618	67 (11%)	12 (2%)	193 (31%)
3.1	528	96 (18%)	12 (2%)	129 (24%)
3.2	478	74 (15%)	12 (3%)	99 (21%)
3.3	490	73 (15%)	42 (9%)	84 (17%)
3.4	485	86 (18%)	4 (1%)	39 (8%)
3.5	425	87 (20%)	5 (1%)	21 (5%)
3.6	420	83 (20%)	0 (0%)	3 (1%)
	Mean	75 (14%)	16 (3%)	162 (26%)
	Median	74 (15%)	12 (3%)	114 (23%)

Column 3 of Table 5.3 shows that a mean of 14% of the referential anomalies in each release are caused by incomplete KCONFIG patches with two releases having values as high as 20%. We could not automatically match the rest of the anomalies to a causing KCONFIG patch. A quick manual analysis of these unmatched anomalies suggests that many of these code blocks have been anomalous since their inception in the code which suggests that the code block has always been dead (or undead). Although we cannot conclude that incomplete KCONFIG patches are the only cause of referential anomalies, our results suggest that they are a common cause.

Finding 2: *Incomplete* KCONFIG *patches often cause referential anomalies. An average of 14% of referential anomalies are caused by changes to* KCONFIG *that are not completely propagated to the source code.*

5.3.3 RQ3: How are Variability Anomalies Fixed?

We now study patches that fix referential anomalies in order to answer RQ3. A referential anomaly is caused by a feature that appears in the boolean formula of the code block, but has no definition in KCONFIG. Therefore, such an anomaly would be fixed by either (1) adding the feature's definition in the KCONFIG files (either by adding a new feature or renaming another feature), or (2) removing that feature from the code block (either by deleting the whole code block or using another defined feature instead).

With respect to the first possibility, the fourth column in Table 5.3 shows the number of anomalies in each release that are fixed by KCONFIG patches. We can see that a very small percentage of referential anomalies (average of 3%) get fixed by future KCONFIG patches. This suggests that although changes to KCONFIG introduce these anomalies, not many of them also get fixed by changes to KCONFIG.

We now look at the second possibility of future CPP patches fixing referential anomalies. The last column in Table 5.3 shows the number of anomalies in each release that are fixed by CPP patches. As shown, an average of 26% of the anomalies are fixed by CPP patches. In some releases (2.6.37-2.6.39), these percentages are as high as 48%-54%. When we analyze those fixes, we find that the majority of them remove the dead code block itself or the CPP condition from around undead code blocks. This indicates that these code blocks should have been originally removed in previous patches (i.e., the incomplete ones) since this is indeed how they got fixed later on. On the other hand, only a few CPP fixes rename the features used in the code to be consistent with those in KCONFIG.

The last two columns in Table 5.3 show that the percentage of anomalies matched to potential fixes is decreasing over time. This is because with more recent releases, there is not much history beyond that release to be able to identify potential fixes. This suggests that such anomalies may stay a while in the kernel before getting fixed. The only exception is the number of KCONFIG patches fixing anomalies (Column 3) in release 3.3. The reason for the higher number of matches is due to a patch that renamed `CONFIG_SPI_BFIN` to `CONFIG_SPI_BFIN5XX` in KCONFIG. This simultaneously fixed 31 anomalies that were due to the missing feature `CONFIG_SPI_BFIN5XX` in different files under the `blackfin` architecture in Linux.

Finding 3: *Referential anomalies are commonly (26% of the time) fixed by* CPP *patches.*

5.3.4 RQ4: How Long do Variability Anomalies Remain in Linux?

We have identified that referential anomalies are commonly fixed through CPP patches. We now look at how long it usually takes for developers to fix these anomalies. Since the location of a code block may change over time (thus changing the location of the anomaly), we need a method to track the anomaly's location as it changes. We use Herodotos [92] to accomplish that. Herodotos tracks bugs over different versions of a software system by considering the lines added and removed in patches such that it can find the location of a particular code block in a different version of the system. Using Herodotos, we track the referential anomalies detected to determine when they are no longer detected in the system. We identify the version that introduces an anomaly and the version that fixes it.

We find that on average, referential anomalies remain in Linux for 6 releases (approx. 10 months). Since some anomalies are still not fixed in the last release examined, we consider the minimum lifetime for those anomalies (i.e., 1 release). The standard deviation of the lifetime of an anomaly is 3.

Finding 4: *Referential anomalies remain unfixed in Linux for an average of 6 releases.*

5.4 Discussion

5.4.1 Interpretation of Our Findings

Finding 4 suggests that on average referential anomalies remain unfixed in Linux for an average of 6 releases. A standard deviation of 3 releases for the anomaly lifetime suggests that there are anomalies that are easier to find and fix than others or that developers care more about. Fixed anomalies provide us insight to how Linux developers address such problems. We attribute the low number of corrections that fix misspellings (4%) found in Finding 1 to the strict Linux review process each change has to undergo before its integration. Anomalies not caused by misspellings are harder to catch by developers during their review process since other places such as KCONFIG files may also need to be checked, which explains the higher percentage of anomalies (14%) caused by incomplete KCONFIG changes found in Finding 2.

There are two explanations for Finding 3 which suggests that Linux developers tend to fix referential anomalies on the variability implementation (the source code) and rather seldom on the variability declaration side (KCONFIG). First, changes to the variability declaration occur less often than code additions because they are less often necessary. This is also seen in the smaller number of KCONFIG patches (10,263) in Linux's repository (i.e., those changing feature definitions in KCONFIG) when compared to the number CPP patches changing the features used in CPP conditions (25,410). Second, changes to KCONFIG have (potentially) wide cross-cutting effects on the Linux code base. Previous work by Eaddy et al. [38] has shown that a high amount of cross-cutting concerns in a software system increases the number of defects. Keeping in mind that Linux is a very large collaborative project, a developer that edits a KCONFIG feature definition potentially changes the behavior of some code that he does not know about, which introduces the anomaly. It, therefore, makes sense that changes to CPP code are later necessary to fix this anomaly, and make it consistent with the KCONFIG change. This makes the understanding of KCONFIG changes, and the required manual code review, much harder than the more focused changes in C source files.

These findings along with the observation that many of these anomalies have existed since the code was created provide an indication that tools to aid programmers in understanding the mapping from feature declaration to variability implementation are necessary. Running such tools when making changes in Linux can help make sure that variability information is kept consistent. However, further investigation into what difficulties developers have in maintaining this consistency are also needed. This can include surveys or interviews of developers to better identify the problems they face with maintaining variability in order to further improve the tools researchers provide them.

5.4.2 Beyond Referential Anomalies and Linux

When studying the evolution of block-level anomalies in this chapter, we focused on referential variability anomalies since they are very common and finding their causes and fixes can be automated. The generated boolean formulas are usually very long and complicated which makes them infeasible to study manually.

The challenge with studying logical anomalies not caused by missing features (see Chapter 4) is that when a boolean formula fails, it is not easy to automatically identify the conflict which caused the failure. Even when such a conflict is identified, it is often difficult to identify the right fix needed to remove the conflict. Let us take the following simple formula for a dead block as an example.

$$(\text{B1} \leftrightarrow \text{CONFIG\_X}) \wedge (\text{CONFIG\_X} \rightarrow \text{CONFIG\_Y} \wedge \text{CONFIG\_Z}) \\ \wedge(\text{CONFIG\_Z} \rightarrow \neg\text{CONFIG\_Y} \wedge \text{CONFIG\_W})$$

B1 is dead because the formula is not satisfiable since CONFIG_Y cannot be defined and undefined at the same time. Several changes can be made in order to fix this anomaly. In CONFIG_X's KCONFIG definition, the dependency on CONFIG_Z can be removed which will allow the formula to be satisfiable. Alternatively, the dependencies in the KCONFIG definition of CONFIG_Z can be changed by either removing the negation of CONFIG_Y or removing the dependency on CONFIG_Y altogether. Such solutions are difficult and expensive to capture automatically which is why analyzing logical anomalies may require additional manual effort.

A possible solution for this is to follow a technique similar to that proposed by Śliwerski et al. [111] where you can identify the release that fixed the problem (i.e., a release where the anomaly no longer appears in), and then analyze the changes that occurred between these releases. The challenge here is since we are dealing with multiple artifacts, a change that introduces or fixes the anomaly may not necessarily be in the code but may be in a related KCONFIG or Makefile. Heuristics can be applied to limit the search space a bit and existing techniques to debug conflicts [126, 127] can be a starting point. However, more investigation in this direction is needed.

Although our study is limited to Linux, we believe our techniques and results can be applied to other systems. Inconsistencies arise from scattered information, and changes that are not properly propagated to related parts of the system. Many systems use CPP to control variability, and also use KCONFIG as a variability modeling notation (e.g., BusyBox, uClibc). Since these systems are similarly structured to Linux, it seems likely that the same observations may apply. Other systems which implement variability differently (e.g., eCos) may also have inconsistencies caused by incomplete changes since variability information is still divided among more than one place.

5.5 Threats to Validity

5.5.1 Internal Validity

Mining GIT. Our work relies on mining the GIT repository in Linux. We only analyze the master repository maintained by Linux Torvalds. This ensures that the commits we analyze have been thoroughly reviewed and that we avoid many of the perils of GIT branching discussed by Bird et al. [21].

Matching Accuracy. Our results rely on the automatic matching scheme we have developed. Since we focus on investigating our raised research questions, and not on providing any tools to be directly used by Linux developers, we develop conservative heuristics to avoid false positives in our matching of anomalies to patches. This explains the large number of unmatched anomalies. However, we manually verify many of the detected matchings to confirm they are correct. Since removed and added features in each patch can be accurately identified from the + and – diff notation, we believe our matching are reasonably accurate.

Misspellings. We conclude that misspellings are not a common cause of variability anomalies. This is based on the fixes we analyzed. However, due to the conservative way we designed our analysis, we would miss any anomalies caused by misspellings but which have not yet been fixed. Thus, our result is the lower bound for the percentage of anomalies caused by misspellings.

5.5.2 Construct Validity

The dataset we use to develop our research questions has been created by Tartler et al. [118] in their previous work. Since this work is a collaboration with them, we avoid researcher expectancies or over familiarity with the data by having the author of this thesis, who was not involved in the original work, study this dataset to develop the research questions.

Since our study focuses on referential anomalies, the fact that the exploratory dataset we examine only contains referential anomalies does not bias our results. We study all 106 patches solving referential anomalies in the exploratory dataset, and thus avoid the need to do any data sampling. The 337 anomalies for which the patches have been created have been randomly sampled from all detected referential anomalies, and thus, the dataset does not suffer from any sampling bias.

5.5.3 External Validity

This work provides a case study of a single software system, the Linux kernel. We do not generalize our results to other software systems. However, Linux is one of the largest and commonly studied open source software systems that supports software variability, and has also been previously studied in terms of variability anomalies. We believe that this case study provides a methodology which can be followed to study causes and fixes of variability anomalies in other systems implementing variability through CPP directives and feature selection such as those discussed by Liebig et al. [69] and Spinellis [113] (e.g., Apache, FreeBSD).

5.6 Summary

In this chapter, we investigated the causes and fixes of referential anomalies (i.e., those caused by undefined KCONFIG features) in the Linux kernel. We found that variability anomalies typically stay in Linux for an average of 6 releases before they get fixed which suggests that detecting these anomalies is not trivial and that fixing them as soon as they are introduced is important. Our findings showed that referential anomalies are often (14% of the time) introduced by incomplete patches which change KCONFIG files without fully reflecting these changes in the corresponding source code. This indicates that automated anomaly detection, such as that provided by UNDERTAKER, should be incorporated into the change process to detect these inconsistencies as soon as the patches are applied. We also found that 26% of the time, these anomalies get fixed by CPP patches that remove the defective code block or rename the undefined feature being used in it. This indicates that KCONFIG changes often have wide cross-cutting effects on the code that are not detected till later and must be fixed through code changes. The patterns for anomaly causes and fixes we found can help developers avoid such problems in the future. They also allow consistency-checking tool designers to automatically identify causes of their detected anomalies and possibly provide suggestions to fix them.

Chapter 6

File Level Variability Anomalies

In Chapter 4, we analyzed the effect of build-space constraints on anomalies at the block level. We showed that considering the build space results in additional anomalies being detected. We also studied the evolution of certain types of block-level anomalies in Chapter 5. However, we observe that, generally, the number of block-level anomalies detected is quite large (Figure 4.3), that it may be hard to analyze complicated boolean formulas to understand the cause of the conflict (Section 5.4.2), and may be difficult to analyze their evolution. In an attempt to provide more manageable results to a developer, we now look at a higher granularity level of anomalies and focus on the file level in this chapter. Since Makefiles deal with code files rather than code blocks, it makes sense to also consider anomalies at the file level when dealing with the build-space constraints. We again use the Linux kernel as our case study, and find ways to ensure the following.

1. Source files are correctly used within KBUILD
2. Constraints in KBUILD are consistent with the rest of the system such that all files can be compiled without conflicts

For the first, we develop rules for detecting files that are not compiled because they are either not used in KBUILD or used incorrectly in KBUILD. In that sense, we are not doing any SAT-based reasoning on the file presence conditions in any way which means we do not detect constraint conflicts. Therefore, we call anomalies detected in this way *non-conflict anomalies*. For the second, we change the propositional formulas discussed in Chapter 4 to work at the file level, and detect conflicts that may lead to file-level anomalies. We call anomalies detected from the second case *conflict anomalies*.

The idea for both techniques is that we want to make sure that all files in the kernel get compiled on at least one variant. Files may not be compiled either because they are not used correctly in the Makefiles or because their constraints can never be satisfied. We discuss such cases in this chapter, and show how we can detect them.

For continuity, we again use a working example of a configurable cellphone system to illustrate the different types of anomalies. Figure 6.1 shows the example we use with a focus on file-level anomalies where Figure 6.1a shows a list of files in a directory, Figure 6.1b shows the related KCONFIG file, and Figure 6.1c shows the related Makefile.

Chapter Organization. We organize this chapter as follows. We first start by describing how we detect non-conflict anomalies in Section 6.1. Section 6.2 then presents how we adapt the propositional formulas discussed in Chapter 4 to work at the file level to detect conflict anomalies. In Section 6.3, we show the results we get when we apply our detection techniques on the Linux kernel. To further examine file-level anomalies, we discuss their evolution in Section 6.4. We then present a discussion of our results in Section 6.5 and how they compare to the block level. We provide a summary of the chapter in Section 6.6.

Related Publications. The work described in this chapter is partially described in the following publications.

Sarah Nadi and Ric Holt. "Make it or break it: Mining anomalies in Linux Kbuild". In: *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. 2011, pp. 315–324

Sarah Nadi and Ric Holt. "Mining Kbuild to detect variability anomalies in Linux". In: *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. Los Alamitos, CA: IEEE Computer Society, 2012, pp. 107–116

Sarah Nadi and Ric Holt. "The Linux kernel: A case study of build system variability". In: *Journal of Software: Evolution and Process* (2013). Early online view. `http://dx.doi.org/10.1002/smr.1595`

```
touch_supp.c  music_supp.c  handsfree.c  speaker_supp.c  unlock.c  mp3.c
```

(a) Files in Directory

```
1 config CODE_UNLOCK
2   bool "Phone will be ulocked by a code"
3
4 config FINGERPRINT_UNLOCK
5   bool "Phone will be unlocked by  fingerprint  detection"
6   depends on !CODE_UNLOCK
7
8 config MP3
9   depends on MUSIC
```

(b) Kconfig

```
1 obj-$(CONFIG_MUSIC) += music_supp.o
2 handsfree-y += handsfree.o speaker_supp.o
3 obj-$(CONFIG_MP3) += mp3.c
4 ifeq ($(CONFIG_CODE_UNLOCK),y)
5   obj-$(CONFIG_FINGERPRINT_UNLOCK) += unlock.o
6 endif
```

(c) Makefile

Figure 6.1: Example of several types of file-level variability anomalies.

6.1 File-level Non-conflict Anomalies

To begin our analysis at the file level, we first choose a simple, inexpensive technique to detect non-conflict anomalies. Non-conflict anomalies are those we can detect without reasoning about any constraints. In general, anomalies at the file level are in the form of files that can never be compiled in any variant (i.e., *dead files*) or conditional files that are always compiled in every variant (i.e., *undead files*). Based on how conditional compilation works in KBUILD, we develop three simple rules that detect dead files. These rules ensure that source files are correctly used in KBUILD (i.e., the Makefiles).

We start with the example in Figure 6.1 to show the types of anomalies we catch, and then formalize the rules we use. Let us look at the source files in Figure 6.1a. Ideally, each file should be compiled through a corresponding entry in the Makefile. However, in the example, we can see that file `touch_supp.c` does not appear anywhere in the Makefile shown in Figure 6.1c. In this case, the whole file is dead and will never be compiled. We call this a *File Not Used* anomaly. If we look at the next file in the directory, `music_supp.c`, we find that it is used on Line 1 of the Makefile in Figure 6.1c. However, the entry is conditioned on feature `MUSIC` which does not appear in the KCONFIG file in Figure 6.1b. Thus, the feature can never be selected and the file will never be compiled. We call this, a *Feature Not Defined* anomaly. If we look at the next two files, `handsfree.c` and `speaker_supp.c`, we find that they both appear on Line 2 of the Makefile as part of the composite object `handsfree` (see Chapter 3 for a description of composite objects). This means that both files will be compiled in the `handsfree.o` object. However, we can see that this object is not

Anomaly	**Description**
File Not Used	A .c file exists in the directory but is not used in the Makefile of that directory.
Feature Not Defined	A .c file is referenced in the Makefile, and its presence is conditioned on a Kconfig feature being defined. However, this feature is not defined in any of the Kconfig files.
Variable Not Used	A .c file is referenced in the Makefile as part of a composite variable definition, but this variable is never used.

Table 6.1: Types of non-conflict file-level anomalies

included in the `obj-y` list anywhere in the Makefile which means that the source files included in it do not actually get compiled into the final Linux variant. We call this a *Variable Not Used* anomaly.

We now formalize the three rules we use to detect the three types of anomalies described above. The violation of these rules results in the three types anomalies described above, and summarized in Table 6.1.

1. *Rule 1:* In each directory, every `fileName.c` file should have a corresponding `fileName.o` entry in the Makefile of that directory.
2. *Rule 2:* If `fileName.o` is dependent (directly or indirectly through a composite object) on some configuration feature, then there should be a corresponding entry defined for this feature in one of the KCONFIG files.
3. *Rule 3:* If `fileName.o` is part of a composite object definition, then we must make sure that this composite object gets used somewhere in that same Makefile.

We believe that the Variable Not Used and Feature Not Defined anomalies indicate errors. For both these types of anomalies, it seems clear that the developer intended for this file to be included in the build process, but due to some error has not setup things correctly and forgot to use the composite variable in the first case or forgot to define the KCONFIG feature in the second case. However, the File Not Referenced anomaly may not necessarily indicate an error. Instead, it can be caused by a developer intentionally not using a file because this file is no longer needed but is kept in the directory for reference purposes.

6.2 File-level Conflict Anomalies

The rules described above in Section 6.1 do not take into consideration the dependencies in KCONFIG and do not reason about constraints in any way. In this section, we discuss file-level conflict anomalies which result because of a conflict between the build-space constraints (in the form of file presence conditions) and the configuration space constraints (in the form of feature dependencies). Note that we do not discuss the code space constraints here since we are working at the file level and not at the block level anymore.

Recall that in Chapter 4, we presented the original UNDERTAKER Boolean logic formulas which detect dead (Formula 4.1) and undead (Formula 4.2) code blocks by finding conflicts between the configuration space and the code space. In that chapter, we modified these formulas to include the build space. In this chapter, to work on the file level, we again modify these formulas, but in this case we replace the block with the code file and then add the build constraints. This is shown in Formulas 6.1 and 6.2.

In Formula 6.1, we define a file as dead if it can never be present (i.e., will never get compiled) while satisfying the combination of constraints in the build space B and the configuration space K.

$$Dead_{F_N} = \neg sat(File_N \wedge B \wedge K) \tag{6.1}$$

Similarly, Formula 6.2 defines a file as undead if we cannot find a case where it does not get compiled while satisfying the build space and configuration space constraints.

$$Undead_{F_N} = \neg sat(\neg File_N \wedge B \wedge K) \tag{6.2}$$

To illustrate how these formulas work, we refer back to the example in Figure 6.1. On Line 5 of the Makefile shown in Figure 6.1c, we can see that file `unlock.c` is compiled if both features `CODE_UNLOCK` and `FINGERPRINT_UNLOCK` are selected. However, we can see that the phone specifications in the KCONFIG file in Figure 6.1b state that having `FINGERPRINT_UNLOCK` depends on not having `CODE_UNLOCK` (for added security reasons for example). Thus, the constraints needed for the file to compile can never be satisfied and `unlock.c` is dead. We call this category of anomalies *build-configuration anomalies*.

Since there are only two spaces involved in these formulas, anomalies can arise either because of direct conflicts between the two spaces as the previous example shows or because of conflicts due to missing feature definitions. These are *build-configuration missing anomalies*, which we now provide an example for. Line 3 of Figure 6.1c shows that file `mp3.c` will be compiled if feature `MP3` is selected. However, we can see that `MP3` depends on feature `MUSIC` in the KCONFIG file in Figure 6.1b, while there is no feature definition for `MUSIC` there. This means that `mp3.c` is dead according to Equation 6.1 since feature `MUSIC` will always be false, and thus the equation cannot be satisfied.

6.3 Detecting File-level Anomalies in the Linux Kernel

In this section, we discuss the results of applying both techniques for finding file-level anomalies in the Linux kernel. First, we use the three rules described in Section 6.1 to detect anomalies in the entire Linux source tree across 10 recent releases, v2.6.37-v3.6. We then use the formulas described in Section 6.2 to detect dead and undead files due to conflicting constraints. We apply these formulas to the same ten releases. Note that we ignore the staging directory to avoid skewing the results (similar to the block-level anomalies).

6.3.1 Results: Non-conflict Technique

When using the three rules described in Section 6.1, the anomaly detection process on each release takes an average of 1.95 minutes on a Core i7 2.67 GHz with 8GB RAM. However, our technique can also be applied on a per directory basis. That is, a developer can detect anomalies only in the directory they are responsible for. In that case, the analysis will just run in a few seconds and the developer does not have to wait for the full analysis of the whole source tree. This will be convenient for developers to use before committing their work.

Figure 6.2 shows the number of non-conflict file-level anomalies detected in each release. We omit the Variable Not Used anomaly from the figure for better visualization since there are only three detected instances of this type of anomaly throughout all the releases studied (discussed later). The figure shows that the File Not Used anomaly occurs more frequently than the Feature Not Defined anomaly. We now discuss the three categories below.

File Not Used. We can see that File Not Used anomalies occur in each of the examined releases in Figure 6.2. On average, there are 48 File Not Used anomalies in each of the releases we examined, i.e., 48 .c files not mentioned in any of the Makefiles. The majority of these files are in the `arch` and `drivers` directories which is consistent with related work on the Linux kernel which finds that the `drivers` directory contains most of the code, as well as many inconsistencies, errors, and clones [47, 52, 71, 118].

Detecting the right number of files that have not been used in the Makefiles was challenging at first. Taking v2.6.39 as an example, we originally discovered over 266 File Not Used anomalies in this category. We later realized that Linux does not follow the recommended practice of not `#include`-ing .c files. Therefore, before reporting a file as not used, we have to first check if it is included in any other .c file which does not have an anomaly. If it is, then we do not report this file as having an anomaly since it will be indirectly included in the final built variant through the `#include` directive. Based on that, we report 49 File Not Used anomalies instead of 266 in v2.6.39, for example.

While examining the files reported as not used, we observe that several files seem to be left behind for reference purposes or for ongoing maintenance. For example, some of the files reported as not used

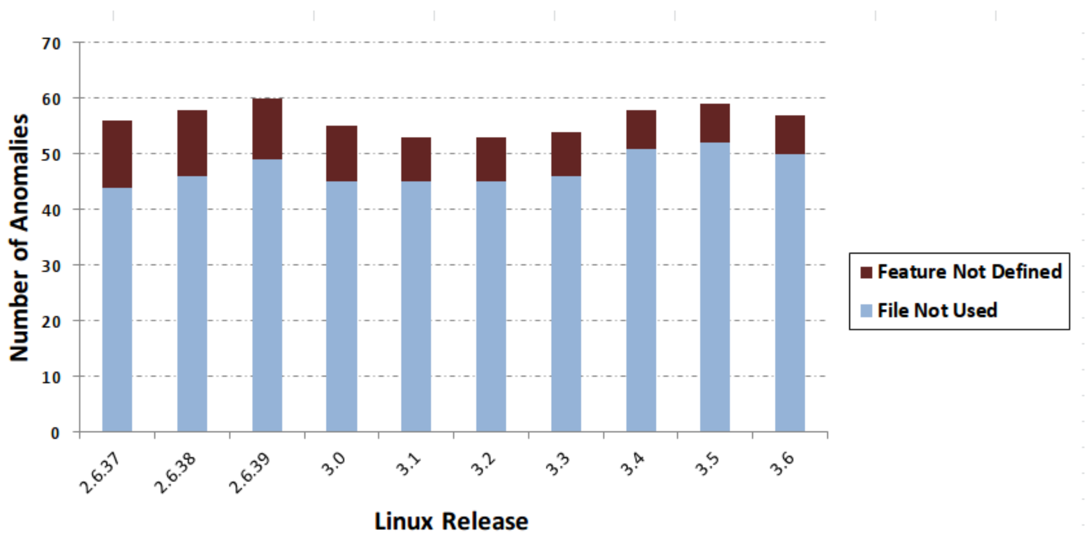


Figure 6.2: Number of non-conflict file-level anomalies detected in each Linux kernel release examined.

include `old_checksum.c`, `dummy.c`, and `test.c`. Their names indicate that they are probably there for test purposes or as a copy of some previously existing code.

Variable Not Used. We find that the Variable Not Used anomaly is rare. Over the ten releases examined, we only find three distinct occurrences of this case. We discuss one of these occurrences here. In the Makefile in directory `arch/cris/arch-v32/mach-fs/`, file `vcs_hook.c` is found in the following line:

```
bj-$(CONFIG_ETRAX_VCS_SIM) += vcs_hook.o
```

According to our rules, this means that there should be a variable called `bj` used somewhere in the Makefile (as `bj.o`). However, no such occurrence was found, and so it was reported as a Variable Not Used anomaly. However, on closer inspection, this looks like a typo where the line was intended to be as follows, but the o was forgotten:

```
obj-$(CONFIG_ETRAX_VCS_SIM) += vcs_hook.o
```

Detecting this category of anomalies can help catch such spelling mistakes. This anomaly remained in the kernel from 2007 to 2011 when we reported it[1] where it got pushed upstream in v3.0. It is surprising that this has remained undetected since 2007 (according to the developer we communicated with). This

[1]https://lkml.org/lkml/2011/5/17/239

indicates that these types of anomalies may be hard to detect manually especially when they do not break the system. However, they may cause some functionality to be missing. In this particular case, the code in the file `vcs_hook.c` was only used for development purposes, and is not actually used in the "real world".

For one of the other Variable Not Used anomalies, we find another typo where `Obj-` was used instead of `obj-`. This anomaly was fixed right away[2] just lasting for one release which suggests that it was an error that got fixed right away. The other Variable Not Used anomaly we found does not seem to be a typo and got introduced in v3.5 remaining there even in the last release we examined (v3.6).

Feature Not Defined. On average, we find 8 Feature Not Defined anomalies in each release examined, i.e., 8 features that are used in the Makefiles, but are not defined in any of the KCONFIG files. This means that for each release, there are an average of 8 cases where some code is expected to compile when a certain feature is selected but this code is actually never compiled because the feature is not defined

It is also interesting that one of these undefined features, `CPU_S3C2400`, appears in `default` and `depends on` clauses in a KCONFIG file. When there is no definition for a feature in KCONFIG, then this feature can never be selected. Additionally, any other feature depending on it in any way will not be visible to the user for selection in the configuration process, and will thus, in turn, never be itself selected as well. This means that some of the intended variability for this feature as well as all features depending on it can never actually be used. It is worth noting that the dependencies on this feature as well as the Makefile entry and the corresponding file got removed in release v3.1[3]. According to the developer's comments, the functionality for `CPU_S3C2400` was never completed.

6.3.2 Results: Detecting Conflict Anomalies

Using the formulas described in Section 6.2, we are able to detect several dead files because of constraint conflicts in the Linux kernel, but no undead files. The dead files we detect are all in the build-configuration missing category which means they are caused by missing KCONFIG definitions.

Table 6.2 shows the number of dead files detected in each release examined. On average, there are 56 dead files in each of the releases examined. Over the ten releases examined, there are 88 unique dead files. We discuss one example here. Listing 5[4] shows the propositional formula for file `drivers/spi/spi-stmp.c`, which is dead because of the missing feature `ARCH_STMP3XXX`. Recall that the last line lists the feature(s) that are missing (one feature in this case). Since this feature is not defined and can thus never be selected, the missing feature is defaulted to false in the boolean formula to see if the formula can still be satisfied when this feature is not selected. This also allows the developer to know the missing features from the anomaly reports. Note that this same missing feature caused four dead code

[2]http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=d5ef642355bdd9b383ff5c18cbc6102a06eecbaf
[3]http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=632b7cf6c056a355fe920c5165c4d7772393b817
[4]formula has been simplified for better illustration.

Release	build-configuration missing
2.6.37	54
2.6.38	60
2.6.39	59
3.0	62
3.1	57
3.2	51
3.3	49
3.4	52
3.5	56
3.6	59

Table 6.2: Dead files found due to a conflict between the build and configuration constraints caused by a missing feature definition.

blocks on the block level anomaly analysis. It is also worth noting that this anomaly was detected in all the kernel releases we examined which suggests that some of these anomalies are not easily detected.

6.4 Evolution of File-level Anomalies

After determining that file-level anomalies exist in Linux, our next step is to determine whether such anomalies persistently appear in the system and whether they get resolved or not. Finding the evolution of anomalies gives an indication of the seriousness of the detected anomalies. If developers invest time in fixing the detected anomalies between releases, then this suggests that these anomalies are important. It also illustrates whether the same anomaly is occurring in all releases (i.e., it is just ignored or left behind) or if new anomalies also get introduced. Fortunately, tracking the evolution of file-level anomalies is easier than tracking those on the block level. We, therefore, do not need elaborate techniques as those described in Chapter 5 for the block level. We can simply track whether an anomaly for the same file path exists in each release. Again, we investigate the evolution of non-conflict and conflict based anomalies separately and describe the results of each analysis.

6.4.1 Evolution of Non-conflict Anomalies

Figure 6.3 shows the number of anomalies fixed in each examined release as opposed to those introduced. The evolution plot starts at release 2.6.38 to be able to correctly identify introduced and fixed anomalies in each release by comparing them to the previous release. Additionally, Figure 6.3 breaks down the number of introduced and fixed anomalies by type. The figure shows that only a couple of non-conflict file-level anomalies get introduced and fixed in each release. In order to understand how developers address the

Listing 5 Example of a build-configuration missing dead anomaly in file drivers/spi/spi-stmp.c. The last line shows the the missing features causing the conflict.

```
drivers/spi/spi-stmp.c
&&
drivers/spi/spi-stmp.c <-> CONFIG_SPI && CONFIG_SPI_STMP3XXX
&&
CONFIG_SPI_STMP3XXX -> CONFIG_SPI && CONFIG_ARCH_STMP3XXX && CONFIG_SPI_MASTER
&&
! CONFIG_ARCH_STMP3XXX
```

anomalies we discover and how serious they are considered, we manually study each of the fixed anomalies. We now present our findings for the File Not Used and Feature Not Defined anomalies.

File Not Used anomaly. We find that while File Not Used anomalies represent on average 83% of the total non-conflict anomalies in each release, only an average of 5% of the File Not Used anomalies in one release are fixed in the next release. This suggests that although there are many anomalies of this type, most of these anomalies are not particularly urgent since only 5% of them actually get addressed in the next release. Nonetheless, we cannot conclude that the File Not Used anomaly is totally insignificant. In the releases we examined, there is a total of 65 distinct File Not Used anomalies. By the last release we examined (v3.6), 15 of these anomalies were actually addressed (i.e., approximately 23%). This shows that although this type of anomaly does not get immediately fixed, many of them eventually get addressed.

We find that in fourteen out of the fifteen File Not Used anomalies that got fixed, the fix was simply removing the file from the source tree. This suggests that most of the File Not Used anomalies are indeed caused by lax maintenance where code that is no longer needed is still left in the source tree. However, developers still invest time in cleanup up the code. In the remaining case, an entry was actually added for the unused source file in the Makefile suggesting that this is useful code. Note that since we are ignoring the staging directory, we are analyzing stable code which has passed testing and inspection. The staging directory, on the other hand, would contain many more anomalies that need more serious fixes since it is still code that has not been fully tested.

Feature Not Defined anomaly. We find that Feature Not Defined anomalies represent on average 15% of the total anomalies in each release. On average, 9% of the Feature Not Defined anomalies in one release are addressed in the next release. This suggests that although there are fewer anomalies of type Feature Not Defined, a higher percentage of them actually get fixed immediately (i.e., within one release) when compared to the File Not Used anomalies. Over the course of the ten releases we studied, we have observed seven fixes for Feature Not Defined anomalies. Out of these, 3 (43%) simply removed the source file from the directory, 2 (29%) added a definition for the corresponding missing feature in KCONFIG, and 2 (29%) changed the feature this file depends on in KBUILD. In the two latter cases, the change suggests that the file should indeed be compiled and that the missing feature definition was an error.

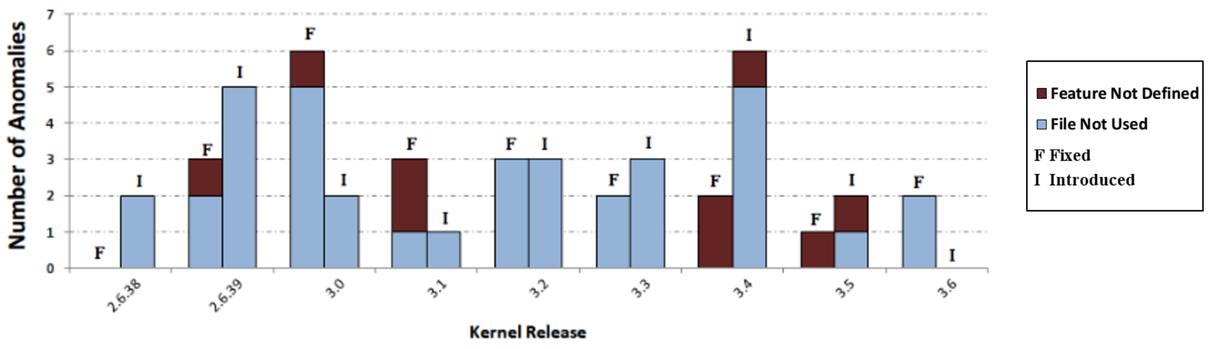


Figure 6.3: Breakdown of non-conflict fixed and introduced anomalies by type over Linux releases. For each release, the left column shows fixed anomalies and the right column shows introduced anomalies.

Our results support our suspicions that the File Not Used anomaly may not necessarily indicate an error while the Feature Not Defined anomaly is more likely to be causing a problem. This is because in the second type of anomaly, developers have created an entry for the compilation of the file in the Makefile showing their intention of compiling it while for the first type of anomaly, we do not know if this file is doing anything important or not.

6.4.2 Evolution of Conflict Variability Anomalies

Figure 6.4 shows the evolution of dead files caused by conflicts due to missing features. The figure shows the number of distinct anomalies fixed (dark bars on left) and introduced (light bars on right) in each release. Again, we can see that in each release, some anomalies get fixed while others get introduced. In total, we could observe the fixes for 26 build-configuration missing anomalies. Out of these, 10 (39%) fixes involved keeping the file and adding a definition for the missing feature or removing the undefined feature from the dependencies in KCONFIG. This indicates that these were real errors and that the dead file should get compiled.

Figure 6.5 shows an example of a commit which fixes a build-configuration missing anomaly in release v3.1. In this particular case, file `drivers/rtc/rtc-stmp3xx.c` depended on feature `RTC_DRV_STMP` (not shown in the commit). However, in KCONFIG, `RTC_DRV_STMP` depended on the undefined feature `ARCH_STMP3XXX` (highlighted in Figure 6.5) which means that the dependency of `RTC_DRV_STMP` can never be satisfied causing the file to never be built. The patch shown in Figure 6.5 fixes this by changing the dependency of `RTC_DRV_STMP` in KCONFIG from the undefined feature `ARCH_STMP3XXX` to the defined

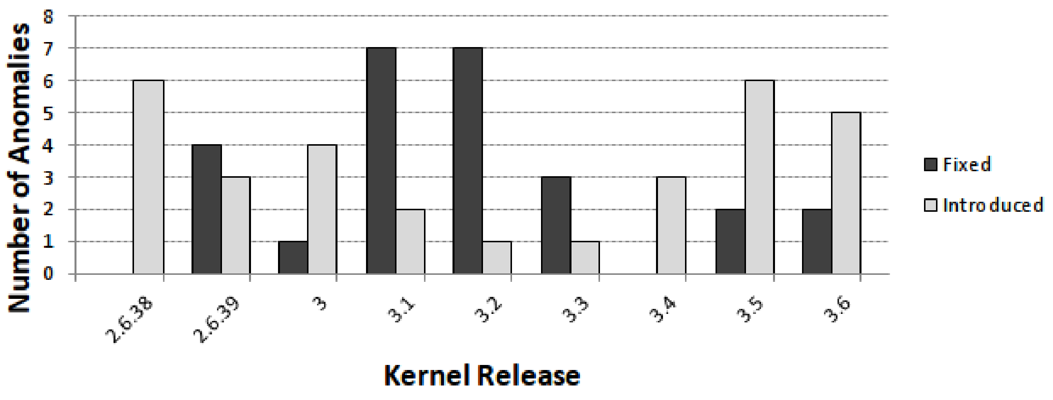


Figure 6.4: Evolution of dead code files caused by constraint conflicts due to missing features.

feature `ARCH_MXS` (highlighted). Note that this same anomaly would be reflected on the code-block level, but instead of just one anomaly reported for the file, an anomaly would be reported for each affected code block in the file. In this case, there were two such code blocks.

Another similar case is the dead file `drivers/crypto/picoxcell_crypto.c` caused by an indirect dependency on the undefined feature `ARCH_PICOXCELL`. In the file-level analysis, one build-configuration missing anomaly was reported for it while three dead blocks were reported on the code block level. The anomaly was fixed in v3.2 by adding a feature definition for the missing feature `ARCH_PICOXCELL` in KCONFIG.

6.5 Discussion

We have shown that both non-conflict and conflict-finding techniques detect file-level anomalies. There are both similarities and differences between both techniques in terms of the anomalies they detect. For example, the non-conflict technique detects files that are dead because they are not used in the Makefiles, while the conflict technique can only detect dead files that are already used in the Makefiles. On the other hand, both techniques would detect files that are dead because the feature they directly depend on in the Makefile does not have a definition. However, the conflict technique can also detect indirect missing features while the non-conflict technique cannot. For example, if we refer back to the `mp3.c` file case in Figure 6.1c, which we discussed before, we will find that the non-conflict technique would not detect a problem with the file because the feature it directly depends on, i.e., `MP3` is defined. On the other hand, the conflict technique would detect a problem, because it also looks at the dependencies of `MP3` and tries to satisfy Equation 6.1. This difference explains why in the same release, the conflict technique detects more build-configuration missing anomalies than those detected as Feature Not Defined by the non-conflict technique.

Although working at the block level is more detailed, and is a natural extension to previous work [119], working at a higher level, namely the file level, has provided us several advantages. A first advantage is that it yields more manageable results which is beneficial to both the developer and to us when analyzing the results. Developers do not want to be overwhelmed by hundreds of anomalies if they can reach the same conclusion with less information. For example, in terms of missing anomalies, the same missing feature `ARCH_STMP3XXX` in the block-level results is also discovered during our analysis at the file level. At the file level, only two dead file are caused by this problem while at the block level, 6 code blocks (in the two different files) are reported as dead because of the same problem. This suggests that working at the file level may end up solving the same problems, but with less information provided to the developer. We believe developers may find it easier to deal with the file level since the number of reported anomalies for the same problem are more manageable. However, there is of course the risk that other anomalies are unique only to the block level and might be missed at the file level. Thus, it might be beneficial for developers to start the analysis at the file level, and solve the issues there, which in turn will remove many of the block-level anomalies, and then they may move onto block-level analysis to solve any remaining issues.

Another advantage is that we are able to track the evolution of anomalies. This is difficult to do when working at the code block level, as described in Chapter 5, because code blocks are identified by their line numbers in a code file. Unfortunately, these line numbers may change from one release to the other as more patches and changes are being applied to the kernel. In many situations, we thought that certain code block anomalies are fixed in a particular release, but later discover that the anomaly still exists. The anomalous piece of code had only been moved to somewhere else in the file which changes the block number. Thus, tracking code-block anomalies by the block number, anomaly description, or line numbers can lead to inaccurate analysis while this is easier to handle on the file level.

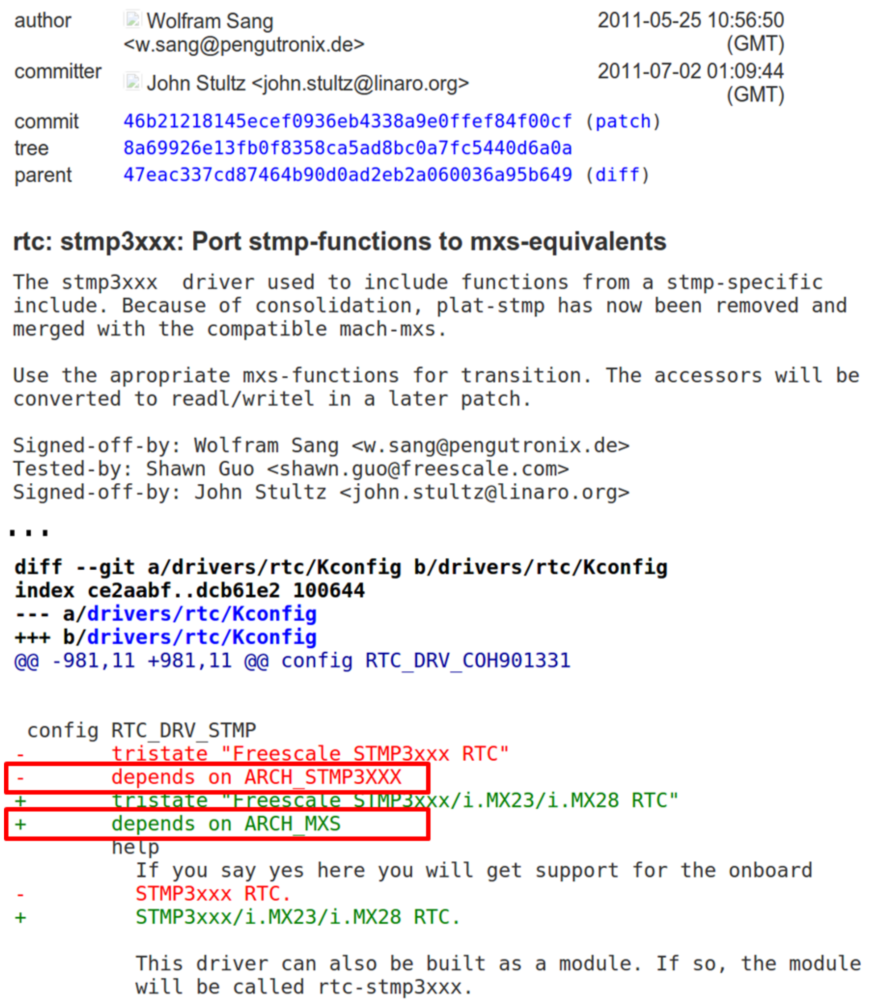
The results of the evolution of the file-level anomalies suggest that while many of the dead files are unused code that is left behind, there are still some dead files that should get compiled. However, given the number of file-level anomalies detected and the fact that the dead files are mainly due to missing features, it seems developers find it easier to maintain the consistency between the build and configuration spaces. It seems that the challenge is keeping this consistent with the code space which seems to introduce more anomalies since many indirect conflicts may arise.

Since we manually verified several of the detected file-level anomalies during the evolution analysis, we believe our results are accurate. That said, the same threats to validity as those at the block level discussed in Chapter 4 apply here. Additionally, we do not claim that the types of non-conflict file-level anomalies we detect is a comprehensive list. There may be other incorrect setups in the Makefiles that may lead to other types of anomalies.

6.6 Summary

In this chapter, we have studied file-level variability anomalies in the Linux kernel. We detected files that are dead on every variant because they are not used in the Makefiles or because the configuration feature controlling their compilation is not defined in the KCONFIG files. Additionally, we detected dead files which are due to indirectly undefined features which can only be discovered when the build-space constraints and the configuration-space constraints are combined. This is because the undefined feature(s) can only be identified when the features appearing in the file presence conditions (i.e., the build-space constraints) are considered. Our results show that anomalies occur more rarely on the file level than on the block level. However, since the same file-level anomaly can be manifested as multiple block-level anomalies, working on the file level can provide more manageable results.

[6]`http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=46b21218`



```
author     Wolfram Sang                                  2011-05-25 10:56:50
           <w.sang@pengutronix.de>                                    (GMT)
committer  John Stultz <john.stultz@linaro.org>          2011-07-02 01:09:44
                                                                      (GMT)
commit     46b21218145ecef0936eb4338a9e0ffef84f00cf (patch)
tree       8a69926e13fb0f8358ca5ad8bc0a7fc5440d6a0a
parent     47eac337cd87464b90d0ad2eb2a060036a95b649 (diff)
```

rtc: stmp3xxx: Port stmp-functions to mxs-equivalents

```
The stmp3xxx  driver used to include functions from a stmp-specific
include. Because of consolidation, plat-stmp has now been removed and
merged with the compatible mach-mxs.

Use the apropriate mxs-functions for transition. The accessors will be
converted to readl/writel in a later patch.

Signed-off-by: Wolfram Sang <w.sang@pengutronix.de>
Tested-by: Shawn Guo <shawn.guo@freescale.com>
Signed-off-by: John Stultz <john.stultz@linaro.org>
...
diff --git a/drivers/rtc/Kconfig b/drivers/rtc/Kconfig
index ce2aabf..dcb61e2 100644
--- a/drivers/rtc/Kconfig
+++ b/drivers/rtc/Kconfig
@@ -981,11 +981,11 @@ config RTC_DRV_COH901331


 config RTC_DRV_STMP
-       tristate "Freescale STMP3xxx RTC"
-       depends on ARCH_STMP3XXX
+       tristate "Freescale STMP3xxx/i.MX23/i.MX28 RTC"
+       depends on ARCH_MXS
        help
          If you say yes here you will get support for the onboard
-         STMP3xxx RTC.
+         STMP3xxx/i.MX23/i.MX28 RTC.

          This driver can also be built as a module. If so, the module
          will be called rtc-stmp3xxx.
```

Figure 6.5: This commit[6] replaces the dependency of RTC_DRV_STMP on the undefined feature ARCH_STMP3XXX by a dependency on a defined feature ARCH_MXS which resolves a build-configuration missing anomaly in file `drivers/rtc/rtcstmp3xx.c`

Chapter 7

Using Anomalies to Extract Configuration Constraints

Recall that in Section 2.5.2, we defined the term variability anomaly as a general umbrella for all unexpected behavior related to the configurability of the system. In the previous chapters, we discussed dead and undead code blocks and files as forms of such variability anomalies. We showed how to detect such anomalies and how they evolve over time. In this chapter, we look at variability anomalies in the form of build-time errors and a variation of dead code as sources of configuration constraints.

In previous chapters, when detecting variability anomalies, we used knowledge of the existing constraints in the configuration space (i.e., the variability model) to determine the valid configurations. In other words, we detect only anomalies that occur under valid configurations. *Valid configurations* are feature selections that satisfy all the constraints in the configuration space. We do this because we do not care about anomalies that exist under configurations not allowed by the configuration space in the first place. Such *invalid configurations* should not occur by definition. In this chapter, instead of detecting anomalies that occur in valid configurations using knowledge from the variability model, we use anomaly detection mechanisms to identify the constraints that should be enforced in the variability model. This is based on the assumption that the variability model should prevent the configurations under which anomalies would occur. We assume no knowledge of the variability model and try to reverse engineer the configuration constraints which should be enforced in it from the implementation.

In order to reverse engineer the variability model from the solution space (i.e., the code space and build space), we need to identify the sources of configuration constraints it enforces. We suspect that many of these configuration constraints are enforced in the code which is why we develop static-analysis techniques (presented in this chapter) to extract them. To determine if this holds in practice, we conduct an empirical study on four large open-source systems with existing variability models: uClibc, BusyBox, eCos, and the Linux kernel to compare the constraints we automatically extract to those in the existing models. We are

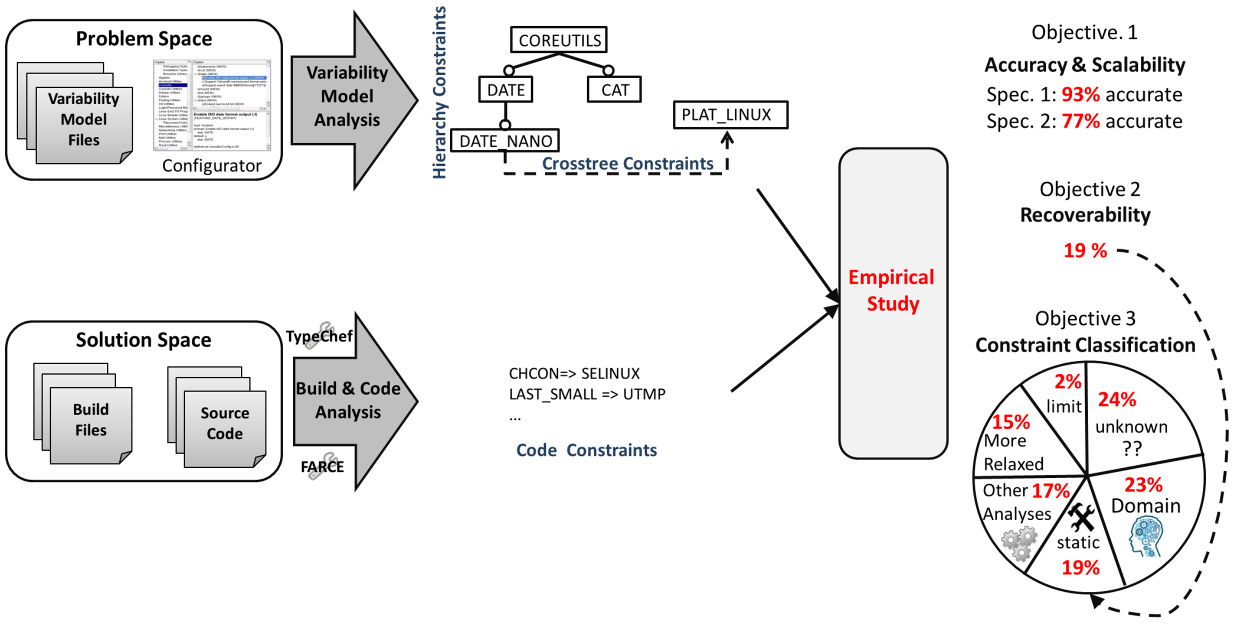


Figure 7.1: Overview of proposed approach and empirical study

interested in understanding the different types of configuration constraints defined in the problem space and how many of these are statically reflected in the solution space. Specifically, our empirical study has the following three objectives: (1) evaluate the accuracy and scalability of our constraint-extraction methodology, (2) evaluate the recoverability of existing variability-model constraints using our approach, and (3) classify existing variability-model constraints. Figure 7.1 shows an overview of the approach we follow in this chapter as well as our empirical evaluation.

As shown in Figure 7.1, to reach our objectives, we need to extract the configuration constraints from the solution space and compare them to the existing constraints in the problem space. Recall from Chapter 2 that there has been much research to extract constraints from existing variability models within the problem space [16, 102, 118]. Such extractors can interpret the semantics of different variability-modeling languages to extract both hierarchy and cross-tree constraints, as shown in Figure 7.1. Thus, extracting the configuration constraints from the solution space is the remaining challenge. We address this challenge in this chapter and show how we extract configuration constraints from the solution space and later compare them to the problem space.

Chapter Organization. Section 7.1 first discusses the sources of configuration constraints in the solution space that we use in our extraction. In Section 7.2, we summarize the problem statement we are addressing in this chapter. Section 7.3 then explains how we use static-analysis techniques, based on the identified sources, to extract constraints from the solution space. In Section 7.4, we present the results of our empirical study and then present the threats to the validity of our work in Section 7.6. Section 7.7 discusses how this work is different from other related work. We then provide a summary of this chapter in Section 7.8.

Related Publications. The work described in this chapter has been previously published in the following paper.

> Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. "Mining configuration constraints: static analyses and empirical results". In: *Proceedings of the International Conference Software Engineering (ICSE)*. Hyderabad, India, 2014, pp. 140–151

7.1 Sources of Solution-space Constraints

As explained in Chapter 2, the solution space consists of build and code files (i.e., the build space and the code space). Our focus is on C-based systems that realize configurability with a build system and the C preprocessor. The build system determines the source files and the preprocessor determines the code fragments to be compiled. The latter is realized using conditional-compilation preprocessor directives such as `#ifdefs`.

To compare constraints in the variability model to those in the code, we must find ways to extract global configuration constraints from the code. Since the majority of configuration features are directly used in code or build files through conditional compilation as shown in Chapter 3, we expect that many of the configuration constraints are reflected in the solution space. Note that in the previous chapters, our code space constraints consisted of localized code-block presence conditions as extracted by Tartler et al. [118]). However, in this context, we need to look at the system globally rather than at each code block or file individually. We assume that there is a solution-space constraint (i.e., spanning both code and build spaces) if any configuration violating this constraint is ill-defined by some rule. There may be several sources of constraints that fit such a description. In this thesis, we concentrate on two tractable sources of constraints: (i) those resulting from build-time errors and (ii) those resulting from the effect of features on build files and on the structure of the code (i.e., `#ifdef` usage). We now explain the justification behind the two rules.

7.1.1 Build-time Errors

Every valid configuration needs to build correctly. In C projects, various types of errors can occur during the build process: preprocessor errors, parsing errors, type errors, and linker errors. We assume that if a specific configuration leads to a build-time error, then there should be a constraint in the variability model preventing this configuration from being valid. Our goal here is to determine the configuration constraints that prevent such build errors. Thus, we derive configuration constraints from the following rule:

> ***Rule 1: No Build Errors.*** *Each valid configuration of the system must not cause build-time errors that would prevent it from being successfully preprocessed, parsed, type checked, and linked.*

```
1 #ifndef Y
2 void foo() { ... }
3 #endif
4
5 #ifdef X
6 void bar() { foo(); }
7 #endif
```

(a) type error

```
1 #if defined(Z)&&defined(X)
2 ...
3 #ifdef W
4 ...
5 #endif
6 ...
7 #endif
```

(b) feature effect

Figure 7.2: Examples of constraint sources

A naive, but not scalable, approach to extract these constraints would be to build and analyze every single configuration in isolation. If every configuration with feature X compiles except when feature Y is selected, we could infer a constraint $\mathrm{X} \rightarrow \neg\mathrm{Y}$. For instance, in Figure 7.2a, the code will not compile in some configurations due to a type error in Line 6: The function `foo()` is called under condition X while it is only defined under condition $\neg\mathrm{Y}$ (Line 2). Therefore, the constraint $\mathrm{X} \rightarrow \neg\mathrm{Y}$ must always hold. The problem space should enforce this constraint to prevent invalid configurations that break the compilation. However, already in a medium-sized system such as BusyBox with 881 Boolean features, this results in more than 2^{881} configurations to analyze, which is more than the number of atoms in the universe. We show how this can be avoided in Section 7.3.

7.1.2 Feature Effect

Ideally, variability models should also prevent meaningless configurations, such as redundant feature selections that do not change the solution space. That is, if feature *A* is selected in a configuration, then we expect that *A* adds or changes some code functionality that was not previously present. However, it may also be the case that a feature does not change the code functionality unless other features are selected (or deselected). Such a dependency may be reflected in the variability model such that a configurator may hide or disable this feature when the other features are not selected to simplify the configuration process.

Determining if two variants of a program are equivalent is difficult (even undecidable). We approximate this by comparing whether the programs differ in their source code at all. We define the *feature effect* as the effect of the feature on the compiled code. That is, when a feature is selected, which parts of the code get added or removed. If two different configurations yield the same code, this suggests some anomaly in the model. We extract constraints that prevent such anomalies. We use the following rule as a simplified, conservative approximation of our second source of constraints:

Rule 2: Lexically Distinct Variants. *Each valid configuration of the system should yield a lexically distinct program.*

The use of features within the build system and the preprocessor directives for conditional compilation provides information about the context under which selecting a feature makes a difference in the final product. In the code fragment in Figure 7.2b, selecting W without selecting Z and X will not break the system. However, only selecting W will not affect the compiled code since the surrounding block will not be compiled without Z and X also being selected. Thus, $W \rightarrow Z \wedge X$ is a feature-effect constraint that should likely be in the model, even though violating it will not break the compilation.

It is worth noting that this is similar to detecting dead blocks in the code (see Chapter 4). However, the difference here is that to detect if a block is dead or not, you only consider the block's presence condition as well as that of the file it is part of. On the other hand, to find the effect of a feature, you have to consider *all* the code blocks it appears in throughout the whole program as will be explained in Section 7.3.3.

7.2 Problem Statement

In Section 2.1.2, we mentioned that variability-model constraints arise from different sources. These included technical low-level dependencies discoverable from the code. We discussed two examples of such sources in Section 7.1.1 and Section 7.1.2 above where the constraints exist for technical reasons discoverable from the code. Our goal in this chapter is to automatically extract such constraints. However, it is not clear if other sources of constraints exist beyond implementation artifacts and how prevalent they are. We, therefore, also strive to identify the sources of any non-recovered constraints.

Improving empirical understanding of constraints in real systems is important, especially since several studies emphasize configuration and implementation challenges for developers and users due to complex constraints [15, 19, 50, 72]. Such knowledge not only allows us to understand which parts of a variability model can be reverse-engineered and consistency-checked from code, and to what extent; but also how much manual effort, such as interviewing developers or domain experts, would be necessary to achieve a complete model. For example, a key challenge when reverse-engineering a variability model from constraints is to disambiguate the hierarchy [103]. Thus, this process could be supplemented by knowing which sources of constraints relate to hierarchy information in the model.

We focus on the sources of constraints described in both rules above since such constraints can be extracted using *decidable* and *scalable* static-analysis techniques. There are, of course, also other possible kinds of constraints in the code resulting from errors or other rules (e.g., buffer overflows or null-pointer dereference). However, many of these require looking at multiple runs of a program (which does not scale well or depends on imprecise sampling) or produce imprecise or unsound results when extracted statically.

7.3 Extracting Solution-space Constraints

As explained in the introduction of this chapter, one of our goals here is to determine if the configuration constraints enforced in variability models can be automatically extracted from the solution space. Thus, we need to develop techniques to extract these constraints from the solution space in order to later compare

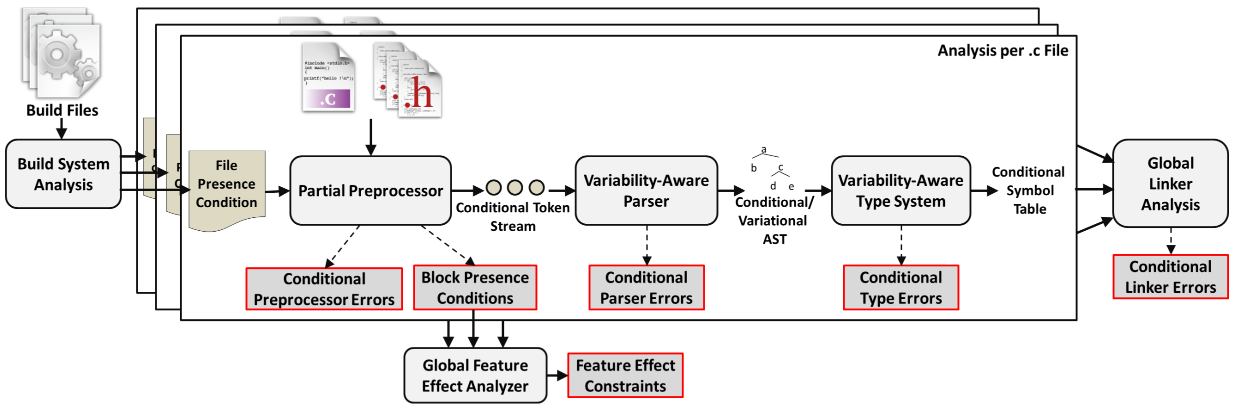

Build Files
Build System Analysis
File Presence Condition
Analysis per .c File
.c
.h
Partial Preprocessor
Conditional Token Stream
Variability-Aware Parser
Conditional/ Variational AST
Variability-Aware Type System
Conditional Symbol Table
Global Linker Analysis
Conditional Preprocessor Errors
Block Presence Conditions
Conditional Parser Errors
Conditional Type Errors
Conditional Linker Errors
Global Feature Effect Analyzer
Feature Effect Constraints

Figure 7.3: Variability-aware approach based on the TypeChef infrastructure to extract configuration constraints from code

them to the ones existing in the problem space. To do so, we use the two rules described in Section 7.1 to extract code constraints from *preprocessor errors*, *parser errors*, *type errors*, *linker errors*, and *feature effect*. Figure 7.3 shows an overview of the approach we use while Table 7.1 summarizes what is done in each of these steps, both of which we explain in detail in this section.

As shown in Figure 7.3, before analyzing the code in a specific C file, we first determine under which conditions the build system includes this file to be able to accurately derive constraints. Recall that we use the term *presence condition (PC)* to refer to a propositional expression over features that determines when a certain code artifact is compiled. For example, a file with presence condition HUSH $\vee$ ASH is compiled and linked if and only if the features HUSH or ASH are selected.

To avoid an intractable brute-force approach of analyzing every possible configuration, we build on the recent research infrastructure, *TypeChef*, to analyze the entire configuration space of C code with build-time variability at once [57, 60, 61]. Our overall strategy for extracting code constraints is based on parsing C code without evaluating conditional-compilation directives. We extend and instrument TypeChef to accomplish this.

TypeChef[1] is a variability-aware infrastructure developed by Kästner et al. [60]. It aims to analyze C code containing `#ifdef` variability such that the code can be correctly parsed to allow for later tasks such as type checking. The challenge with parsing and analyzing C code containing `#ifdef` variability is that different expansions for the code can occur under different feature selections. To correctly parse such code, and detect any errors, the code must be parsed separately for all possible feature combinations. Of course, such analysis is infeasible especially for systems with a large number of features. TypeChef solves this through its variability-aware analysis.

[1]https://github.com/ckaestne/TypeChef

Table 7.1: Summary of each analysis step. The definition of each step as well as where it is explained are described. Running times reported in Table 7.4

	Step	Definition	Described in
TypeChef	File PC Extraction	Analyzing the build system to extract file presence conditions.	Section 7.3, Chapter 3, and Berger et al. [17]
	Lexing	Partially preprocessing and lexing the file to produce a conditional token stream	Section 7.3 and detailed description by Kästner et al. [60]
	Parsing	Parsing the conditional token stream produced by the partial preprocessor and creating a conditional Abstract Syntax Tree (AST)	Section 7.3 and detailed description by Kästner et al. [60]
	Type checking	Traversing the conditional AST to detect conditional type errors	Example of type error in Section 7.3.1 and detailed description by Kästner et al. [60]
	Symbol Table creation	Creating a conditional symbol table for each parsed file	Example of conditional symbol table and linker error in Section 7.3.2 and detailed description by Kästner et al. [60]
FARCE	Feature effect - Build Constr.	Calculating the feature effect constraints by only considering the file presence conditions	See Section 7.1.2
	Feature effect Constr.	Calculating the full feature effect constraint based on the presence conditions appearing in the conditional token stream produced by TypeChef	See Section 7.1.2
	Preprocessor Constr.	Calculating the preprocessor constraints based on conditional preprocessor errors detected by TypeChef	See Section 7.3.1
	Parsing Constr.	Calculating the parsing constraints based on conditional parsing errors detected by TypeChef	See Section 7.3.1
	Type Checking Constr.	Calculating the type constraints based on conditional type errors detected by TypeChef	See Section 7.3.1
	Linker Constr.	Calculating the linker constraints based on the linker errors detected by FARCE using the conditional symbol tables produced by TypeChef	See Section 7.3.2

TypeChef consists of three variability-aware components [60] shown in Figure 7.3. The first is the *partial preprocessor* which includes all necessary header files and expands all macros, but preserves conditional-compilation directives. This allows each C file to be self-contained and analyzed individually. The partial preprocessor then produces a conditional token stream where each token is guarded by a corresponding presence condition (including the file presence condition). This conditional token stream is then passed to a *variability-aware parser* which tries to optimize the parsing process through splitting and combining different token streams according to their presence conditions. This allows the whole conditional token stream to be parsed in a single pass, and a single conditional abstract syntax tree (AST) to be produced. The conditional AST can then be processed by the *variability-aware type system* to detect type errors. This variability-aware analysis is conceptually sound and complete with regards to a brute-force approach of analyzing all configurations separately. However, it is much faster since it does the analysis in a single step and exploits similarities among implementations of different configurations; see [57, 60, 61] for more detail.

In previous research with TypeChef, it was typically called with a given variability model such that it only emits error messages for parser or type problems that can occur in valid configurations—discarding all implementation problems that are already excluded by the variability model. This is the typical approach to find consistency errors (See Section 2.5.2) which a user can subsequently fix in the implementation or in the variability model [29, 121, 122]. Since we need to extract all constraints without knowledge of valid configurations, we use TypeChef in a different context where we run it *without* a variability model and process all reported problems in all configurations.

We extend and instrument TypeChef, and implement a new framework FARCE (FeAtuRe Constraint Extractor) [43], which analyzes the output of TypeChef and the structure of the codebase with respect to preprocessor-directive nesting, derives constraints according to our low-level rules, and provides an infrastructure to compare extracted constraints between a variability model and code. We now explain our design decisions and methodology using the C code adapted from BusyBox in Figure 7.4 as a running example. For simplicity, we show the file presence condition as an `#ifdef` spanning the entire file. However, all file presence conditions (extracted from the build system analysis) are included in the calculation of all constraints.

7.3.1 Preprocessor, Parser, and Type Constraints

Preprocessor errors, parser errors, and type errors are detected at different stages of analyzing a file. However, the post-processing used to extract constraints from them is similar which is why we discuss them together here. In contrast, linker errors require a global analysis over multiple files, which we discuss separately in Section 7.3.2.

Preprocessor Errors. A normal C preprocessor stops on `#error` directives, which are usually intentionally introduced by developers to avoid invalid feature combinations. We extend TypeChef's partial preprocessor to log `#error` directives with their corresponding presence condition and to continue with the rest of the file instead of stopping on the `#error` message. The rest of the file is then processed under the negated error condition (i.e., assuming this condition does not hold). The presence condition of an `#error` is extracted from the features used to guard the `#error` statement. In our example (Figure 7.4), Line 3

```
#ifdef ASH //represents the file presence condition

#ifdef NOMMU
#error "... ash will not run on NOMMU machine"
#endif

#ifdef EDITING
static line_input_t *line_input_state;

void init () {
   initEditing ()
  int maxlength = 1 *

  #ifdef MAX_LEN
    100;
  #endif //MAX_LEN
}
#endif //EDITING

int main() {
  #ifdef EDITING_VI
    #ifdef MAX_LEN
      line_input_state->flags |= 100
    #endif
  #endif
}
#endif //ASH
```

Figure 7.4: Example of C code with build-time errors (adapted from `ash.c` in Busybox)

shows a `#error` directive that occurs under the condition $\text{ASH} \wedge \text{NOMMU}$. Thus, the partial preprocessor can output a list of conditional preprocessor errors as shown in Figure 7.3.

Parser Errors. Similarly, a normal C parser stops on syntax errors such as mismatched parentheses. The TypeChef variability-aware parser reports an error message together with a corresponding presence condition, but continues parsing for other configurations. The variability-aware parser will output a list of conditional parsing errors for each file processed as shown in Figure 7.3. For example, in Figure 7.4, a parser error occurs on Line 11 because of a missing semicolon if MAX_LEN is not selected. In this case, our analysis reports a parser error under condition $\text{ASH} \wedge \text{EDITING} \wedge \neg\text{MAX\_LEN}$.

Type Errors. Where a normal type checker reports type errors in a single configuration, TypeChef's variability-aware type checker [57, 61] reports each type error together with a corresponding presence condition. In Figure 7.4, TypeChef detects a type error in Line 22 if EDITING is not selected since `line_input_state` is only defined under condition $\text{ASH} \wedge \text{EDITING}$ on Line 7. TypeChef would, thus, report a type error under condition $\text{ASH} \wedge \text{EDITING\_VI} \wedge \text{MAX\_LEN} \wedge \neg\text{EDITING}$. Again, for each file processed, the variability-aware type system will output a list of conditional type errors as shown in Figure 7.3.

Constraints. Following Rule 1 (No Build Errors), we expect that each file should compile without errors. Every error message with a corresponding condition indicates part of a configuration that does not compile and should hence be excluded in the variability model. Thus, if an error occurs under condition `X`, we assume that the variability model should enforce that `X` should never occur. More formally, for each condition ϕ of an error, we extract a configuration constraint $\neg\phi$ which we believe should be enforced in the variability model. We calculate these constraints using FARCE. During the calculation, we rely on TypeChef's simplification of the error presence conditions to simplify the calculated constraints.

Referring back to our example in Figure 7.4, recall that there was a preprocessor error detected on Line 3 under condition $\text{ASH} \wedge \text{NOMMU}$. Thus, FARCE extracts a configuration constraint which ensures that the variability model does not allow features `ASH` and `NOMMU` to be simultaneously selected ($\neg\text{ASH} \vee \neg\text{NOMMU}$). Rewriting this to its equivalent implication, FARCE extracts the following preprocessor constraint from this error: $\text{ASH} \rightarrow \neg\text{NOMMU}$. Following the same logic for the parser and type error examples discussed above, we extract the following constraints: $\text{ASH} \rightarrow \neg\text{EDITING} \vee \text{MAX\_LEN}$ from the parser and $\text{ASH} \rightarrow \text{EDITING} \vee \neg\text{EDITING\_VI} \vee \neg\text{MAX\_LEN}$ from the type system.

7.3.2 Linker Constraints

To detect linker errors in configurable systems, TypeChef builds a conditional symbol table for each file during type checking (see Figure 7.3). The symbol table describes all non-static functions as exported symbols and all called but not defined functions as imports. All imports and exports are again guarded

Table 7.2: Example of two conditional symbol tables

file	**symbol**	**kind**	**presence condition**
Figure 7.4	`init`	export	ASH $\wedge$ EDITING
	`main`	export	ASH
	`initEditing`	import	ASH $\wedge$ EDITING
other file	`initEditing`	export	INIT

by corresponding presence conditions. We check linkage only within the application and discard all symbols defined in libraries (with additional analysis though, we could also model library symbols with corresponding presence conditions). Table 7.2 shows the conditional symbol table (without type information) corresponding to the code example in Figure 7.4, assuming that symbol `initEditing` is defined under presence condition INIT in some other file (not shown). For more detail on conditional symbol tables, see Aversano et al. [7] and Kästner et al. [61].

In contrast to the file-local preprocessor, parser, and type analyses, linker analysis is global across all files. Thus, we wait until TypeChef analyzes all source files and then use FARCE to perform the global post processing. From all conditional symbol tables, FARCE derives linker errors and corresponding constraints. A linker error arises when a module imports a symbol that is not exported by any other model (def/use) or when two modules export the same symbol (conflict). The *def/use* constraints ensure that a symbol cannot be used unless it is defined (similar to the idea of safe composition [121]). In other words, the presence condition of an import implies at least one presence condition of a corresponding export. *Conflict* constraints ensure mutual exclusion of the presence conditions of exports with the same function name. In other words, you cannot have multiple definitions for the same symbol.

More formally, we derive configuration constraints for each symbol s as follows:

$$\textit{def/use}(s) = \Big(\bigvee_{(f,\phi)\in imp(s)} \phi\Big) \rightarrow \Big(\bigvee_{(f,\psi)\in exp(s)} \psi\Big)$$

$$\textit{conflict}(s) = \bigwedge_{(f_1,\psi_1)\in exp(s);(f_2,\psi_2)\in exp(s);f_1\neq f_2} \neg(\psi_1 \wedge \psi_2),$$

where *imp*(s) and *exp*(s) look up all imports and exports of symbol s in all conditional symbol tables and return a set of tuples (f,ψ), each determining the file f in which s is imported/exported and presence condition ψ.

An overall linker constraint can be derived by conjoining all def/use and conflict constraints for each symbol in the set of all symbols S: $\bigwedge_{s\in S} \textit{def/use}(s) \wedge \textit{conflict}(s)$. Assuming that the two files shown in

Table 7.2 are the only files in our example, we would extract the constraint ASH $\wedge$ EDITING $\rightarrow$ INIT for symbol `initEditing`. In this case, this is a result of a def/use constraint.

7.3.3 Feature Effect

To ensure *Rule 2 (Lexically Distinct Variants)*, we detect the configurations under which a feature has no effect on the compiled code and create a constraint to disable the feature in those configurations.

The general idea is to detect nesting among `#ifdef`s: When a feature occurs only nested inside an `#ifdef` of another feature, such as EDITING that occurs only nested inside '`#ifdef` ASH' in the example in Figure 7.4, the nested feature does not have any effect when the outer feature is not selected. Hence, we would create a constraint that the nested feature should not be selected without the outer feature, because it would not have any effect: EDITING $\rightarrow$ ASH in this example.

Unfortunately, determining the feature effect is not easy. Extracting constraints directly from nesting among `#ifdef` directives produces inaccurate results, because features may occur in multiple locations inside multiple files, and `#if` directives allow complex conditions including disjunctions and negations. Hence, we develop the following novel and principled approach deriving a constraint for each feature's effect from presence conditions throughout the system.

First, we collect all unique presence conditions of all code fragments occurring in the entire system (in all files, including the corresponding file presence condition as usual). Technically, we inspect the conditional token stream produced by TypeChef's partial preprocessor and collect all *unique* presence conditions as shown in Figure 7.3 (note that this covers all conditional compilation directives, `#if`, `#ifdef`, `#else`, `#elif`, etc. including dynamic reconfigurations with `#define` and `#undef`).

To compute a feature's effect, we use the following insights: Given a set of presence conditions P found for code blocks anywhere in the project and the set of features of interest F, then we say a feature $f \in F$ has no effect in a presence condition $\phi \in P$ if $\phi[f \leftarrow True]$ is equivalent to $\phi[f \leftarrow False]$, where $X[f \leftarrow y]$ means substituting every occurrence of f in X by y. In other words, if enabling or disabling a feature does not affect the value of the presence condition, then the feature does not have an effect on selecting the corresponding code fragments. Furthermore, we can identify the exact condition when a feature f has an effect on a presence condition ϕ. This is done by identifying all configurations in which the result of substituting f is different (using xor: $\phi[f \leftarrow True] \oplus \phi[f \leftarrow False]$). This method is also known as *unique existential quantification*.

Putting the pieces together, to find the overall effect of a feature on the entire code in a project, we take the disjunction of all its effects on all presence conditions. We then assume that the feature should only be selected if it has an effect, resulting in the following constraint:

$$f \rightarrow \bigvee_{\phi \in P} \phi[f \leftarrow True] \oplus \phi[f \leftarrow False]$$

This means that we choose to disable a feature by default when it does not have an effect on the build. Alternatively, we could enable a feature by default and forbid disabling it when disabling it has no effect: We just need to negate f on the right side of the above formula. However, we assume the more natural setting where most features are disabled by default and so we look for the effect of *enabling* a feature.

In our example in Figure 7.4, we can identify six unique presence conditions (excluding tokens for spaces and line breaks): ASH (Line 0), $\mathrm{ASH} \wedge \mathrm{NOMMU}$ (Line 2), $\mathrm{ASH} \wedge \mathrm{EDITING}$ (Line 6), $\mathrm{ASH} \wedge \mathrm{EDITING} \wedge \mathrm{MAX\_LEN}$ (Line 13), $\mathrm{ASH} \wedge \mathrm{EDITING\_VI}$ (Line 20), and $\mathrm{ASH} \wedge \mathrm{EDITING\_VI} \wedge \mathrm{MAX\_LEN}$ (Line 21). To determine the effect of $\mathrm{MAX\_LEN}$, we would substitute it with `True` and `False` in each of these conditions, and create the following constraint (assuming that $\mathrm{MAX\_LEN}$ does not occur anywhere else in the code):

$$\begin{aligned}
&\mathrm{MAX\_LEN} \rightarrow \Big((\mathrm{ASH} \oplus \mathrm{ASH}) \vee \\
&\big((\mathrm{ASH} \wedge \mathrm{NOMMU}) \oplus (\mathrm{ASH} \wedge \mathrm{NOMMU})\big) \vee \\
&\big((\mathrm{ASH} \wedge \mathrm{EDITING}) \oplus (\mathrm{ASH} \wedge \mathrm{EDITING})\big) \vee \\
&\big((\mathrm{ASH} \wedge \mathrm{EDITING} \wedge \mathrm{True}) \oplus (\mathrm{ASH} \wedge \mathrm{EDITING} \wedge \mathrm{False})\big) \vee \\
&\big((\mathrm{ASH} \wedge \mathrm{EDITING\_VI}) \oplus (\mathrm{ASH} \wedge \mathrm{EDITING\_VI})\big) \\
&\big((\mathrm{ASH} \wedge \mathrm{EDITING\_VI} \wedge \mathrm{True}) \oplus (\mathrm{ASH} \wedge \mathrm{EDITING\_VI} \wedge \mathrm{False})\big)\Big) \\
\equiv &\mathrm{MAX\_LEN} \rightarrow \mathrm{ASH} \wedge (\mathrm{EDITING} \vee \mathrm{EDITING\_VI}),
\end{aligned}$$

This confirms that $\mathrm{MAX\_LEN}$ has an effect if and only if ASH and either $\mathrm{EDITING}$ or $\mathrm{EDITING\_VI}$ are selected. In all other cases, the constraint enforces that $\mathrm{MAX\_LEN}$ remains deselected. Additionally, to determine how many constraints the build system alone provides, we do the same analysis for file presence conditions instead of presence conditions of code blocks. Note that the feature-effect analysis on the build system alone is incomplete (since it does not include the code) and provides only a rough approximation.

7.4 Empirical Study Setup

We study four real-world systems with existing variability models. As shown in Figure 7.1, our objectives are:

- **Objective 1 (Accuracy and Scalability):** Evaluate the *accuracy* and *scalability* of our extraction approach. This is done by checking if the configuration constraints we extract from implementation are enforced in existing variability models.
- **Objective 2 (Recoverability):** Study the *recoverability* of variability-model constraints using our approach. Specifically, we are interested in how many of the existing model constraints reflect implementation specifics that can be automatically extracted.

- **Objective 3 (Constraint Classification):** *Classify* variability-model constraints. In other words, we want to understand which constraints are technically enforced and which constraints go beyond the implementation artifacts. This allows us to understand which reverse-engineering approaches to choose in practice.

For all three objectives, we report the key results here. Refer to our online appendix for full datasets, extracted formulas, and the raw qualitative results [90].

7.4.1 Subject Systems

We chose four highly-configurable open-source projects from the systems domain. All are large, industrial-strength projects that realize variability with the build system and the C preprocessor. Our selection reflects a broad range of variability-model and codebase sizes in the reported range of large commercial systems.

Our subjects comprise the following systems and variability-model sizes. The first three use the KCONFIG [132] and the last one uses the CDL [125] language and configurator infrastructure in the problem space. We chose systems with existing variability models to have a basis for comparison.

uClibc is an alternative, resource-optimized C library for embedded systems. We analyze the x86_64 architecture in uClibc v0.9.33.2, which has 1,628 C source files and 367 features described in a KCONFIG model. **BusyBox** is an implementation of 310 GNU shell tools (ls, cp, rm, mkdir, etc.) within one binary executable. We study BusyBox v1.21.0 with 535 C source files and 921 documented features described in a KCONFIG model. The **Linux kernel** is a general-purpose operating system kernel. We analyze the x86 architecture of v2.6.33.3, which has 7,691 C files and 6,559 features documented in a KCONFIG model. **eCos** is a highly configurable real-time operating system intended for deeply embedded applications. We study the i386PC architecture of eCos v3.0 which has 579 C source files and 1,254 features described in a CDL model.

In all four systems, the variability models have been created, maintained, and evolved by the original developers of the systems over periods of up to 13 years. Using them reduces experimenter bias in our study. Prior studies of the Linux kernel and BusyBox have also shown that their variability models, while not perfect, are reasonably well maintained [60, 61, 72, 94, 118]. In particular, eCos and Linux have two of the largest publicly available variability models today.

7.4.2 Methodology and Tool Infrastructure

We follow the methodology shown in Figure 7.1. We first extract hierarchy and cross-tree constraints from the variability models of our subject systems (problem space). We rely on previous analysis infrastructures LVAT [73] and CDLTools [25], which can interpret the semantics of KCONFIG and CDL respectively to extract such constraints and additionally produce a single propositional formula representing all enforced constraints (see [16, 102] for details).

We then run TypeChef on each system, and use our developed infrastructure FARCE to extract solution-space constraints based on Rule 1 (No Build Errors) and Rule 2 (Lexically Distinct Variants). As a prerequisite, we extract file presence conditions from build systems using the build system analysis tool *KBuildMiner* [62] for systems using KBUILD (BusyBox and Linux). We use KBuildMiner here since it had already been tested and used before with TypeChef on both versions of BusyBox and Linux which we use in this work. For the two other systems which do not use KBUILD, we use a semi-manual approach to extract the file presence conditions.

7.4.3 Evaluation Technique

After problem and solution-space constraints are extracted, we compare them according to our three objectives. To address **Objective 1** (accuracy and scalability), we verify whether extracted solution-space constraints hold in the propositional formula representing the variability model (problem space) of each system. We also measure the execution time of the involved analysis steps. For this objective, we assume the existing variability model as the ground truth since it reflects the system's configuration knowledge which developers have specified. To address **Objective 2** (recoverability of model constraints), we determine whether each existing variability-model constraint holds in the solution-space constraint formulas we extract. We use the term *recoverability* instead of *recall*, because we do not have a ground truth in terms of which constraints can be extracted from the code. Since no previous study has classified the kinds of constraints in variability models, we cannot expect that 100% of them are enforced in the code and can be automatically extracted. To address this gap and **Objective 3** (classification of variability model constraints), we show the types of constraints we could automatically recover, and manually investigate 144 randomly sampled non-recovered model constraints to characterize constraints that are not found by our analysis.

7.5 Empirical Study Results

In this section, we present the results of the three study objectives explained in Section 7.4. Note that averages and numbers reported across subjects are geometric means (unless otherwise specified) to account for the differences among the subject systems.

7.5.1 Objective 1: Accuracy and Scalability

We expect that all constraints extracted according to *Rule 1 (No Build Errors)* hold in the problem-space variability model as these prevent any failure in building a system. Constraints that do not hold either

Table 7.3: Constraints extracted with each rule per system and the percentage of those holding in the variability model (VM)

Code Analysis	**uClibc**		**BusyBox**		**eCos**		**Linux**	
	# extracted	% found in VM	# extracted	% found in VM	# extracted	% found in VM	# extracted	% found in VM
Rule 1 (No Build Errors)								
Preprocessor Constr.	158	100	3	100	162	81	12,780	81
Parser Constr.	60	100	23	100	133	91	8,443	100
Type Checking Constr.	958	96	54	100	139	82	256,510	97
Linker Constr.	314	63	38	100	7	100	19,654	90
Total	1,340	90	118	100	441	85	284,914	96
Rule 2 (Lexically Distinct Variants)								
Feature effect Constr.	55	75	359	93	263	62	2,961	95
Feature effect - Build Constr.	25	80	62	0	n/a	n/a	2,552	97
Total	80	76	421	79	263	62	5,513	96

indicate a false positive due to an inaccuracy of our implementation or an error in the variability model or implementation. In contrast, *Rule 2 (Lexically Distinct Variants)* prevents meaningless configurations that lead to duplicate systems. Thus, we expect a large number of corresponding constraints, but not all, to hold in the variability model.

Measurement. We measure accuracy as follows. We keep constraints extracted in the individual steps of our analysis separate. That is, for each build error (*Rule 1*) and each feature effect (*Rule 2*), we create a separate constraint ϕ_i. For each extracted constraint ϕ_i, we check whether it holds in the formula ψ representing all the problem-space constraints from the variability model with a SAT solver, by determining whether $\psi \Rightarrow \phi_i$ is a tautology (i.e., whether its negation is not satisfiable).

We record the execution time of each analysis step separately to measure the scalability of our approach. For all analysis steps performed by TypeChef, which can be parallelized, we report the average and the standard deviation of processing each file. In addition, we report the total processing time for the whole systems, assuming sequential execution of file analyses. For the derivation of constraints, which can not be parallelized, we report the total computation time per system.

Results. Table 7.3 shows the number of unique constraints extracted from each subject system in each analysis step and the percentage of those constraints found in the existing variability model. On average across all systems, constraints extracted with Rule 1 and Rule 2 are 93 % and 77 % accurate, respectively.

Recall that accuracy here means that the constraints we extract automatically are actually enforced in the existing variability models. Both results show that we achieve a high accuracy across all four systems. Rule 1, based on having no build-time errors, is a reliable source of constraints where our tooling produces few false positives (extracted constraints that do not hold in the model). Interestingly, a 77 % accuracy rate for Rule 2 (Syntactically Distinct Variants) suggests that variability models in fact prevent meaningless configurations to a high degree.

Table 7.4: Duration, in seconds unless otherwise noted, of each analysis step. Average time per file and standard deviation shown for analysis using TypeChef. Global analysis time shown for post-processing using FARCE. Description of what is performed in each step is shown in Table 7.1

		uClibc	**BusyBox**	**eCos**	**Linux**
TypeChef	File PC Extraction	manual	7	N/A	20
	Lexing	7 ± 3	9 ± 1	10 ± 6	25 ± 12
	Parsing	17 ± 7	20 ± 3	72 ± 1.6	108 ± 1.9
	Type checking	4 ± 3	5 ± 1	3 ± 5	41 ± 14
	Symbol Table creation	0.1 ± 0.1	0 ± 0.03	3 ± 20	2 ± 2
	Sum for all files (Sequential)	13hr	5hr	7hr	376hr
FARCE	Feature effect - Build Constr.	3	3	N/A	24
	Feature effect Constr.	20	8	1200	1.7hr
	Preprocessor Constr.	0.7	0.7	8	1hr
	Parsing Constr.	16	4	8	39min
	Type Checking Constr.	15	6	5	1.3hr
	Linker Constr.	120	60	840	5hr
	Total FARCE Time	3min	1.4min	34min	10hr

Table 7.4 shows execution times of our tools, which were executed on a server with two AMD Opteron processors (16 cores each) and 128GB RAM. Significant time is taken to parse files, which often explode after expanding all macros and `#include` preprocessor directives. Our results show that our analysis scales reasonably where a system as large as Linux can be analyzed in parallel within twelve hours on our hardware (e.g., using 30 parallel threads).

Accuracy Discussion. Our extraction approach is highly accurate given the complexity of our real-world subjects. While increasing accuracy further is conceptually possible, improving our prototypes into mature tools would require significant, industrial-scale engineering effort, beyond the scope of a research project.

Regarding false positives, we identify the following reasons. First, the variability model and the implementation have bugs. In fact, previous work found several errors in BusyBox and reported them to the developers [61]. In this work, we also found one and reported it in uClibc. Second, all steps involved in our analysis are nontrivial. For example, we reimplemented large parts of a type system for GNU C and reverse-engineered details of the KCONFIG and CDL languages, as well as the KBUILD build system. Little inaccuracies or incorrect abstractions are possible.

After investigating false positives in uClibc linker constraints, we found that many of these occur due to incorrectly (manually) extracted file presence conditions. In general, intricate details in Makefiles, such as shell calls, complicate their analysis [117]. Third, our subject systems each implement their own mechanisms for providing and generating header files at build-time, according to the configuration. We implemented emulations of these project-specific mechanisms to statically mimic their behavior, but such emulations are likely incomplete. We are currently investigating using symbolic execution of build systems [117] in order to accurately identify which header files need to be included under different configurations.

Scalability Discussion. Our evaluation shows that our approach scales, in particular to systems sharing the size and complexity of the Linux kernel. However, we face many scalability issues when combining complex constraint expressions into one formula, mainly in Linux and eCos. Feature-effect constraints were particularly problematic due to the unique existential quantification (see Section 7.3.3), which causes an explosion in the number of disjunctions in many expressions, thus adding complexity to the SAT solver. To overcome this, we omit expressions including more than ten features when aggregating the feature effect formula. This resulted in using only 17 % and 51 % of the feature-effect constraints in Linux and eCos, respectively. The threshold was chosen due to the intuition that larger constraints are too complex and likely not modeled by developers.

We faced similar problems in deriving other formulas, such as the type formula in Linux, but mainly due to the huge number of constraints and not their individual complexity. This required several workarounds and significant memory consumption in the conversion of the formula into conjunctive normal form, required by our SAT solver. Thus, we conclude that extracting constraints according to our rules scales, but can require workarounds or filtering expressions to deal with the explosion of constraint formulas. Refer to our online appendix [90] for more details.

7.5.2 Objective 2: Recoverability

We now investigate how many variability-model constraints can be automatically extracted from the code.

Measurement Strategy. While the extraction approach directly gives us individual constraints to count and compare, the situation is more challenging when measuring constraints from the variability model. Variability models in practice use different specification languages. For example, in our subject systems alone, we already have two languages used: KCONFIG and CDL.

Semantics of a variability model are typically expressed uniformly as a single large Boolean function expressed as a propositional formula describing the valid configurations. After experimenting with several slicing techniques for comparing these propositional formulas, we decide to exploit structural characteristics of variability models that are commonly found. In all analyzed models, we can identify child-parent relationships (*hierarchy constraints*), as well as inter-feature constraints (*cross-tree constraints*). This way, we count individual constraints as the developer modeled them which is intuitive to interpret and allows us to investigate the different types of model constraints. Note that we account only for binary constraints as they are most frequent, whereas accounting for n-ary constraints is an inherently hard combinatorial problem. Technically, we perform the inverse comparison to that in Section 7.5.1: we compare whether each individual problem-space constraint ψ_c holds in the conjunction of all extracted solution-space constraints ϕ_i in each code-analysis category, i.e., whether $(\bigwedge_i \phi_i) \Rightarrow \psi_c$ is a tautology.

	uClibc	**BusyBox**	**eCos**	**Linux**
# of VM Hierarchy Constraints	54	366	588	4,999
	Count (%) Recovered from code			
Rule 1 (No Build Errors)				
Preprocessor Constr.	0 (0 %)	0 (0 %)	0 (0 %)	1 (0 %)
Parser Constr.	0 (0 %)	0 (0 %)	3 (0 %)	1 (0 %)
Type Checking Constr.	0 (0 %)	1 (0 %)	0 (0 %)	0 (0 %)
Linker Constr.	0 (0 %)	1 (0 %)	1 (0 %)	1 (0 %)
Total (Unique)	0 (0 %)	2 (1 %)	4 (1 %)	3 (0 %)
Rule 2 (Lexically Distinct Variants)				
Feature effect Constr.	8 (15 %)	251 (69 %)	60 (10 %)	325 (7 %)
Feature effect - Build Constr.	4 (7 %)	0 (0 %)	-	1,337 (27 %)
Total (Unique)	9 (17 %)	251 (69 %)	60 (10 %)	1,661 (33 %)
Total Unique Constraints Recovered	9 (17 %)	253 (69 %)	64 (11 %)	1,664 (33 %)

(a) Hierarchy

	uClibc	**BusyBox**	**eCos**	**Linux**
# of VM Cross-tree Constraints	118	265	315	7,759
	Count (%) Recovered from code			
Rule 1 (No Build Errors)				
Preprocessor Constr.	2 (2 %)	1 (0 %)	5 (2 %)	6 (0 %)
Parser Constr.	0 (0 %)	0 (0 %)	9 (2 %)	2 (0 %)
Type Checking Constr.	8 (7 %)	15 (6 %)	1 (0 %)	3 (0 %)
Linker Constr.	12 (10 %)	21 (8 %)	1 (0 %)	19 (0 %)
Total (Unique)	16 (14 %)	37 (14 %)	15 (5 %)	28 (0 %)
Rule 2 (Lexically Distinct Variants)				
Feature effect Constr.	6 (5 %)	14 (5 %)	1 (0 %)	58 (1 %)
Feature effect - Build Constr.	3 (3 %)	0 (0 %)	-	316 (4 %)
Total (Unique)	7 (6 %)	14 (5 %)	1 (0 %)	374 (5 %)
Total Unique Constraints Recovered	22 (19%)	51 (19 %)	16 (5 %)	402 (5 %)

(b) Crosstree

Figure 7.5: Number (and percentage) of variability-model constraints recovered from each code analysis. Hierarchy constraints shown in (a) and cross-tree constraints shown in (b).

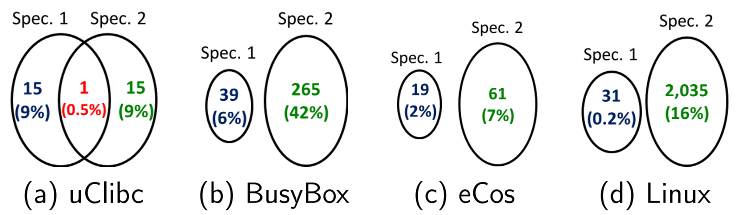


Figure 7.6: Overlap between rules 1 and 2 in recovering variability-model constraints. An overlap means that the same model constraint can be recovered by both rules

Results. In Figure 7.5, we show how many of the variability models' hierarchy (Figure 7.5a) and cross-tree (Figure 7.5b) constraints can be recovered automatically from code. Since the same constraint can be recovered by different analyses, we also show the total number of unique constraints for each rule and for each system. Across the four systems, we recover 26 % of hierarchy constraints and 10 % of cross-tree constraints.

To compare the two rules we use to extract solution-space constraints, we show the overlap between the total number of recovered variability-model constraints (both hierarchy and cross-tree) aggregated across both rules in the Venn diagrams in Figure 7.6. These illustrate that, in all systems, a higher percentage of the variability-model constraints reflects feature-effect constraints in the code (Rule 2). Overall, we can recover 19 % of variability-model constraints using both rules across the four systems where approximately 4 % are found by Rule 1 and 15 % are found by Rule 2.

Recoverability Discussion. We can see a pattern in terms of where variability-model hierarchy and cross-tree constraints are reflected in the code. As can be seen in Figure 7.5a, the structure of the variability model (hierarchy constraints) often mirrors the structure of the code. Rule 2 alone can extract an average 25 % of the hierarchy constraints. An interesting case is Linux where already 27 % of the hierarchy constraints are mirrored in the nested directory structure in the build system (i.e., file presence conditions). We conjecture that this results from the highly nested code structure, where most individual directories and files are controlled by a hierarchy of Makefiles (see Chapter 3) almost mimicking the variability model hierarchy. On the other hand, although harder to recover, cross-tree constraints seem to be scattered across different places in the code (e.g., linker and type information), and seem more related to preventing build errors than hierarchy constraints are. Interestingly, Figure 7.6 shows that there is no overlap (with the exception of one constraint in uClibc) between the two rules we use to recover constraints. This aligns with the different reasoning behind them: *one is based on avoiding build errors while the other ensures that product variants are different*. The fact that our static analysis of the code could recover only 19 % of the variability-model constraints suggests that many of the remaining constraints require different types of analysis or stem from sources other than the implementation. We look at this in more detail in our final objective in the next subsection.

7.5.3 Objective 3: Classification of Variability-model Constraints

To investigate which parts of a variability model can be automatically extracted, our aim is to understand the kinds of constraints that exist in variability models and the analyses and knowledge needed to identify them.

Measurement Strategy. To automate parts of the investigation, we use the recoverability results from Section 7.5.2 to automatically classify a large number of constraints as technical and statically discoverable, which reduces manual investigation to the remaining ones. To manually investigate the remaining constraints, we randomly sample 144 non-recovered constraints (18 hierarchy and 18 cross-tree constraints from each subject system). We then divide these constraints among three researchers involved in this study for manual investigation.

Results. From our manual investigation of 144 non-recovered constraints, we classify five cases in which constraints could not be statically detected from the code with our approach. In Figure 7.1, we summarize the overall classification of the sources of constraints including those automatically found through our static analysis. Note that the numbers and percentages discussed below are based on the sample of 144 constraints while the classification in Figure 7.1 is adjusted for the whole population.

Case 1. Additional Analyses Required: We find 30 (21 %) constraints where the relationship might have been recovered by using more expensive analysis, such as data-flow analysis or testing (11 %), more advanced build-system analysis (5 %), system-specific analysis such as the use of applets in BusyBox or the kernel module system in Linux (3 %), or assembly analysis (2 %).

Case 2. More Relaxed Code Constraints: For 27 (19 %) constraints, we recover constraints that relate two features, but not directly as they appear in the variability model. For example, our analysis would recover the following constraint in BusyBox, BLKID_TYPE $\rightarrow$ VOLUMEID_FAT $\vee$ BLKID while the variability model constraint is BLKID_TYPE $\rightarrow$ BLKID. This suggests that developers may use configuration features differently in the code than what they enforce in the model.

Case 3. Domain Knowledge: For 40 (28 %) constraints, at least one of the features is not used in the implementation. We find two cases where this occurs. The first is that the constraint is *configurator related*, where a feature is used only internally in the variability model to support its menu structure and constraint propagation in the configurator. For example, HAS_NETWORK_SUPPORT in uClibc is a *menuconfig* [104], which helps to organize networking features in the configurator into a menu format. This happens in 27 (19 %) constraints. From their domain knowledge, developers usually know which features are related and that are, thus, grouped together in the same menu. For the remaining constraints, we find that the unused feature represents some form of platform or hardware knowledge. For example, the Linux kernel has the following constraint in its variability model: SERIO_CT82C710 $\rightarrow$ X86_64. The first feature controls the port connection in that particular chip, but which seems to work only with an

X86_64 architecture. Such hardware dependencies are not statically detectable in the code and can be found only through testing the software on the different platforms. We believe that developers use their domain expertise (usually gained from previous testing experiences) to enforce such dependencies.

Case 4. Limitation in Extraction: In 5 (3 %) constraints, our analyses could not recover the constraint because it indirectly depends on some non-Boolean comparison which we do not handle or because it depends on C++ code which we do not analyze.

Case 5. Unknown. We could not determine the rationale behind the remaining 42 (29 %) constraints. First, this indicates that finding constraints manually is a very difficult and time-consuming process which enforces the need for automatic extraction techniques such as those we present here. Second, the fact that we could not manually extract the constraints that were not automatically recovered by our analysis gives us confidence in our results. It might be that such constraints also require additional analyses which we could not easily determine or that they rely on external developer knowledge.

Classification Discussion. Our classification shows that 19 % of the variability model constraints can be statically extracted with our approach (see Section 7.5.2). This seems motivating for automated extraction tools. We have seen that 15 % of constraints are reflected in the nesting structure and can be easily extracted using *Rule 2*, since it depends only on extracting the file presence conditions and lexing the files, which are cheaper steps in the analysis (see Table 7.4). However, our manual analysis of the remaining constraints also shows that many of them can be found only through more expensive analysis or through testing. Additionally, it seems that several constraints in the model are non-technical and are simply responsible for organizing the structure of the model for configuration purposes. We have also come across constraints that could stem only from domain knowledge. Both these facts suggest that additional developer and expert input may always be needed to create a complete model.

Finally, the constraints we find in Case 2 of our manual analysis explain why an analysis may produce accurate constraints and yet recover no variability-model constraints. For example, the type analysis in Linux extracts over 0.25 million constraints which are 97% accurate (Table 7.3), and yet only recovers 3 cross-tree constraints in Figure 7.5b. We plan to investigate the feasibility of alternative comparison techniques to investigate this.

7.6 Threats to Validity

We now discuss possible threats to the validity of the work presented in this chapter.

7.6.1 Internal Validity

Our analysis extracts solution-space constraints by statically finding configurations that produce build-time errors. Conceptually, our tools are sound and complete with regard to the underlying analyses (i.e., they should produce the same results achievable with a brute-force approach of compiling all configurations separately). Practically however, instead of well-designed academic prototypes, we deal with complex real-world artifacts written in several different, decades-old languages. Our tools support most language features, but do not cover all corner cases (e.g., some GNU C extensions, some unusual build-system patterns), leading to minor inaccuracies, which can have rippling effects on other constraints. We manually sample extracted constraints to confirm that inaccuracies reflect only a few corner cases that can be solved with additional engineering effort (which however exceeds the possibilities of a research prototype). We argue that the achieved accuracy, while not perfect, is sufficient to demonstrate feasibility and support our quantitative analysis.

In addition to sampling non-recovered constraints, we randomly sample 6 of the variability-model constraints which were recovered by code analysis in each system, and manually verify that they are indeed technical constraints in the code. Ideally, complementing our results with qualitative data from interviewing the creators of the variability models we study may provide additional insight to the usage of variability constraints.

Our static-analysis techniques currently exploit all possible sources of constraints addressing build-time errors. We are not aware of other classes of build-time errors checked by the GCC/CLANG infrastructure. We could also check for warnings/lint errors, but those are often ignored and would lead to many false positives. Other extensions could include looking for annotations or code comments inside the code, which may provide variability information. However, even in the best case, this is a semi-automatic process. Furthermore, dynamic-analysis techniques, test cases or more expensive static techniques, such as data-flow analysis, may also extract additional information. However, the benefit gained from performing such expensive analyses still needs investigation.

The percentage of recovered variability-model constraints in Linux and eCos may effectively be higher than that reported since we limit the number of constraints we use in the comparison due to scalability issues. Therefore, we can safely use the reported numbers as the worst performance of our tools in these settings. Additionally, we cannot analyze non-C codebases, which also decreases our ability to recover technical constraints in systems such as eCos, where 13% of the codebase comprises C++ and assembler code, which we excluded.

7.6.2 Construct Validity

Different transformations or interpretations of the variability model may lead to different comparison results than the ones achieved (e.g., additionally looking at ternary relationships in the model). Properly comparing constraints is a difficult problem, and we believe the comparison methods we chose provide meaningful results that can also be qualitatively analyzed. Additionally, this strategy allowed us to use the same interpretation of constraints in all subject systems.

7.6.3 External Validity

Due to the significant engineering effort for our extraction infrastructure, we limit our study to Boolean features and to one language: C code with preprocessor-based variability. We apply our analysis to four different systems that include the largest publicly available systems with explicit variability models. Although our systems vary in size and cover two different notations of variability models, all systems are open source, developed in C, and from the systems domain; thus, our results may not generalize beyond that setting.

7.7 Related Work

Our work on studying the sources of configuration constraints builds on, but significantly extends prior work. We reuse the existing TypeChef analysis infrastructure for analyzing `#ifdef`-based variability in C code with build-time variability [60, 61, 70]. However, we use it for a different purpose and extract constraints from various intermediate results in a novel way, including an entirely novel approach to extract constraints from a feature-effect heuristic. While most variability-aware work, including that using TypeChef before, has used the variability model to limit the number of combinations analyzed, we do not use any knowledge from the variability model while extracting constraints. Furthermore, we double the number of subject systems in contrast to prior work. The work is complementary to prior reverse-engineering approach for feature models [103] which showed how to get from constraints to a feature model suitable for end users and tools. Here, we focus on deriving the constraints in the first place.

Le et al. [67] attempt to reverse engineer a feature model from implementation and check if it is consistent with the existing feature model. While our goals are different, the work is closely related. However, the authors there mainly rely on extracting code-block presence conditions from code to determine code constraints. It is not clear if they have a complete parsing infrastructure that expands macros and considers redefinitions or not. Additionally, they require developer involvement to determine the relationship between the features used in the code while our approach is completely automated. We additionally also investigate what the sources of the configuration constraints are. While it is not clear if their comparison strategy is accurate, it is an interesting approach which we could investigate as an alternative comparison technique.

It might be the case that dynamic analysis such as that by Reisner et al. [97] which uses symbolic execution to identify interactions and constraints among configuration parameters by symbolically executing a system's test cases may extract additional constraints as discussed in Section 7.5.3. However, scalability of symbolic execution is limited to medium size systems (up to 14K lines of code with up to 30 options in [97]), whereas our build-time analysis scales to systems as large as the Linux kernel. We also avoid using techniques such as data-flow analysis [22, 24, 70] due to scalability issues. However, we plan to investigate if such techniques may be scaled. Previous work on data-flow analysis, symbolic execution, and testing tailored to variability [22, 70, 89, 97] are interesting starting points.

7.8 Summary

In this chapter, we engineered static analyses to extract configuration constraints and performed a large-scale study of constraints in four real-world systems. Our results raise four main conclusions.

- Automatically extracting accurate configuration constraints from large codebases is feasible to some degree. Our analyses scale to systems as large as Linux using the right infrastructure and approximations. We can recover constraints that in almost all (93%) cases assure a correct build process. In addition, our new feature-effect heuristic is surprisingly effective (77% accurate).
- However, variability models contain much more information than we can extract from code. Our scalable static analysis can only recover 19 % of the model constraints. Qualitative analysis shows additional types of constraints resulting from runtime or external dependencies (often already known by experts) or used for model structuring and configurator support.
- While cross-tree constraints in variability models mainly prevent build-time errors, several parts of the feature hierarchy (25%) can be found using our feature-effect heuristic. The feature hierarchy is one of the major benefits of using variability models. It helps users to configure, and developers to organize features. With our results, reverse engineering a feature hierarchy can be substantially supported.
- Manually extracting technical constraints is very hard for non-experts of the systems, even when they are experienced developers. We experienced this first-hand, giving a strong motivation for automating the task.

Chapter 8

Conclusions

Software product lines (SPLs) promise many benefits to software developers who need to generate multiple variants of their product. Software product lines provide *variability* by allowing the user to select the features they want to include in each generated product. In this dissertation, we focused on C based systems with build-time variability using a build system and C preprocessor. Such systems usually consist of a *configuration space* which describes the features provided and any dependencies between them (usually documented in a variability model), a *build space* which controls which source files get compiled according to the user's feature selection, and a *code space* which contains the implementation of the supported features. Despite their benefits, there are many challenges to adopting and maintaining software product lines.

We addressed two of these challenges in this dissertation. The first is related to the consistency of configuration constraints in a software product line. Any inconsistencies may lead to what we call *variability anomalies* which we detect in terms of dead and undead artifacts. The second is related to identifying the configuration constraints needed to create a variability model. To address this, we identified two other types of variability anomalies which allowed us to extract configuration constraints from existing implementation.

We first summarize our contributions and findings in Section 8.1. We then discuss the insights gained from our work in Section 8.2 and present directions for future work in Section 8.3

8.1 Summary of Thesis Contributions and Findings

We now provide a summary of our contributions and findings. We first discuss the contributions related to the consistency of configuration constraints.

Consistency of Configuration Constraints

1. **Exploring variability in build systems:** We argued that the build system plays an important role in the variability implementation of software product lines [86]. We have demonstrated this through a case study on the Linux kernel's build system, KBUILD, where we showed that 48% of the configurable features are only used in the build system to control source file compilation.

2. **Determining the consistency among variability spaces on the block level:** We have shown that the configuration constraints enforced in the three variability spaces described above often contain inconsistencies [85, 86]. We extended previous work [118] to include the build space constraints and conducted an empirical study on the Linux kernel which showed that many `#ifdef` guarded code blocks are dead or undead because of these inconsistencies.

3. **Determining the consistency among variability spaces on the file level:** We have shown that some conflicts occur at the file level as well causing whole files to be dead [84–86]. Such anomalies occur due to conflicts between the build space and the configuration space mainly caused by missing feature definitions. Our results indicate that these occur less frequently than the block-level anomalies suggesting that developers can manage the file level better but find difficulties keeping all the three spaces consistent.

4. **Studying the evolution of block-level anomalies:** We identified possible causes and fixes of block-level referential variability anomalies (i.e., dead or undead code blocks caused by missing feature definitions) [83]. Through studying the Linux kernel, we have shown that 14% of such referential anomalies are caused by incompletely propagated configuration changes. We also showed that 26% of these anomalies later get fixed on the code side by removing the dead block or guarding it with a different configuration feature.

5. **Studying the evolution of file-level anomalies:** We showed that file level anomalies are in the form of unused or dead files. By studying the Linux kernel, we have shown that such files are often removed from the source tree indicating that they are no longer useful [84–86]. The fact that developers invest time in cleaning them up suggests that leaving such code behind is a maintenance burden. However, we have also found that if the file is dead because of a missing feature definition, developers often go back and correct this which suggests that the file is doing something useful.

We now summarize our contributions related to the extraction of configuration constraints.

Extraction of Configuration Constraints

6. **Designing a methodology to extract configuration constraints:** We introduced a new methodology to extract configuration constraints from existing implementation based on identifying certain

types of variability anomalies [82]. We identified two types of variability anomalies which can be used as sources of configuration constraints: build-time errors and lexically similar product variants. We modified an existing variability-aware analysis infrastructure, TypeChef [60], to extract the information we need from C code and built a new infrastructure, FARCE, which extracts the configuration constraints.

7. **Conducting an empirical study on four real-world systems:** We tested our methodology on four systems with existing variability models: uClibc, BusyBox, eCos, and the Linux kernel [82]. We found that the configuration constraints we extract using the two types of variability anomalies above are 93 % and 77 % accurate, respectively. However, our static analysis techniques could only recover 19 % of the existing variability-model constraints.

8. **Analyzing and classifying sources of configuration constraints:** To understand what other sources of configuration constraints exist, we manually studied a sample of the existing variability model constraints our analysis could not recover [82]. Our qualitative analysis suggests that 23 % of the variability model constraints stem from domain knowledge. For example, developers are often familiar with hardware restrictions or relationships between different functionalities that may not discoverable from analyzing the code.

8.2 Insights

Based on our contributions and findings summarized above, we make the following remarks about the insights we have gained from this work.

1. **Scattering of variability information:** The fact that information contributing to the variability of the system is scattered in three different places increases the chance of inconsistencies. To address this, alternative design techniques which avoid such situations are needed. From a more pragmatic perspective, automated tools which support such situations are needed [54]. We have presented such tools and techniques in our work which help with the maintenance of scattered information.

2. **Granularity of variations points:** We observed that variability at the code-block level (e.g., through `#ifdef`) is harder to maintain than variability at the file level in the build system. Ideally, each file could implement a separate functionality and all variability control can be lifted to the build system. However, it seems that developers still find mechanisms such as using the preprocessor better suited to the level of granularity they need. New generation mechanisms which combine such flexibility but avoid problems such as the inconsistencies we observed (e.g., conflicts between `#ifdef` condition and the variability model) might be needed.

3. **Change support:** Our evolution studies which showed that incompletely propagated changes may cause variability anomalies suggest that integrated consistency-checking mechanisms should be in

place when applying such changes. When a developer has a patch in place, analyses such as those we presented here would inform the developer of any problems. However, finding the best way to integrate such analyses seamlessly such that developers actually use them is still needed.

4. **Support for automatically creating variability models:** Our work showed that while it is feasible to automatically extract configuration constraints by statically analyzing existing implementation, many of these constraints cannot be found this way. While more investigation is needed to identify additional automated techniques, our work provides a practical measure for expectations from automated tools. Companies planning to migrate their systems to software product lines should be aware that expert input will always be needed and that eliciting certain information from sources such as requirement documents, marketing departments, and hardware engineers might be necessary.

8.3 Future Work

Based on our contributions and insights highlighted above, we identify the following possible directions for future work.

1. **Improving build-system analysis:** We have highlighted some of the shortcomings with our build-system analysis as well as with other related techniques in Chapter 3 and Chapter 7. Thus, more complete and generic build-system analysis techniques are needed. We are currently exploring if symbolic execution of build systems might provide better solutions since it can capture information such as which compiler flags or which header files might be included in the compilation of a particular file.

2. **Identifying best practices:** Since many of the anomalies we observed seem to be related to the fact that information is scattered over three different places, designing software product lines differently might reduce this burden. For example, eCos does not have separate artifacts for the information in its configuration space and build space. Instead, it indicates the files that will get compiled along with any specific compiler options within the feature definition itself. The actual build code then gets generated from this information for each specific configuration. It would be interesting to see if such a setup would yield fewer anomalies or if it would affect the quality of the product line in any way. One way would be to correlate design information with the number of reported problems/bugs by users or developers in each system's bug repository. More generally, identifying best practices for implementing variability in software product lines can help avoid such problems in the future.

3. **Exploring additional analyses:** Our work on automatically extracting configuration constraints has used only certain types of static analyses (e.g., type checking). It would be interesting to explore if other analyses such as data-flow analysis can extract additional configuration constraints.

4. **Supporting other aspects of software product-line migration:** In general, our work on helping the creation of software product lines only touches the problem of extracting the configuration constraints. Exploring other aspects involved in product line migration such as feature location or the refactoring of the system are possible future directions.

5. **Expanding to other types of variability support:** We have focused on C-based systems with build-time variability in this thesis. We hope to expand our work to other types of variability support (e.g., load-time options, dynamic binding) as well as other programming languages (e.g., Java).

Overall, our work provides tools and techniques to help maintain software product lines by ensuring the consistency of variability constraints scattered across the system. We also developed automated techniques for extracting configuration constraints from implementation which can be used to reverse engineer variability models. The results of our work can help in the maintenance of existing software product lines as well as help in migrating legacy code into a software product line. Our work lays the foundation for additional future research such as identifying the best practices to maintain and create software product lines.

References

[1] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. “Design recovery and maintenance of build systems”. In: *Proceedings of the International Conference Software Maintenance (ICSM)*. Los Alamitos, CA: IEEE Computer Society, 2007, pp. 114–123.

[2] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. “The evolution of the Linux build system”. In: *Electronic communications of the easst* 8 (2008).

[3] Nele Andersen, Krzysztof Czarnecki, Steven She, and Andrzej Wąsowski. “Efficient synthesis of feature models”. In: *Proceedings of the International Software Product Line Conference (SPLC)*. Salvador, Brazil: ACM, 2012, pp. 106–115.

[4] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-oriented software product lines. Concepts and Implementation.* Springer Berlin Heidelberg, 2013, pp. 243–282.

[5] Sven Apel and Christian Kästner. “An overview of feature-oriented software development”. In: *Journal of Object Technology (JOT)* 8.5 (2009), pp. 49–84.

[6] Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kästner. “Language-independent reference checking in software product lines”. In: *Proceedings of the GPCE Workshop on Feature-Oriented Software Development (FOSD)*. Eindhoven, The Netherlands: ACM Press, 2010, pp. 65–71.

[7] L. Aversano, M. Di Penta, and I.D. Baxter. “Handling preprocessor-conditioned declarations”. In: *Proceedings of the International Workshop Source Code Analysis and Manipulation (SCAM)*. 2002, pp. 83–92.

[8] Lerina Aversano, Massimiliano Di Penta, and Ira. D. Baxter. “Handling preprocessor-conditioned declarations”. In: *Proceedings of the International Workshop Source Code Analysis and Manipulation (SCAM)*. Los Alamitos, CA: IEEE Computer Society, 2002, pp. 83–92.

[9] Don Batory. “Feature models, grammars, and propositional formulas”. In: *Proceedings of the International Software Product Line Conference (SPLC)*. Rennes, France: Springer, 2005, pp. 7–20.

[10] Don Batory. “Feature-oriented programming and the AHEAD tool suite”. In: *Proceedings of the International Conference Software Engineering (ICSE)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 702–703.

[11] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. “Scaling step-wise refinement”. In: *IEEE Transactions on Software Engineering (TSE)* 30.6 (2004), pp. 355–371.

[12] Ira Baxter and Michael Mehlich. “Preprocessor conditional removal by simple partial evaluation”. In: *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. Washington, DC: IEEE Computer Society, 2001, pp. 281–290.

[13] Joachim Bayer, Jean-François Girard, Martin Würthner, Jean-Marc DeBaud, and Martin Apel. “Transitioning legacy assets to a product line architecture”. In: *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. Berlin/Heidelberg: Springer, 1999, pp. 446–463.

[14] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. “Automated analysis of feature models 20 years later: a literature review”. In: *Information systems* 35.6 (2010), pp. 615 –636.

[15] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. “A survey of variability modeling in industrial practice”. In: *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. 2013, 7:1–7:8.

[16] Thorsten Berger and Steven She. “Formal semantics of the CDL language”. Technical Note. Available at `www.informatik.uni-leipzig.de/~berger/cdl_semantics.pdf`.

[17] Thorsten Berger, Steven She, Krzysztof Czarnecki, and Andrzej Wąsowski. “Feature-to-Code mapping in two large product lines”. In: *Proceedings of the International Software Product Line Conference (SPLC)*. 2010.

[18] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. “A study of variability models and languages in the systems software domain”. In: *Ieee transactions on software engineering* 39.12 (2013), pp. 1611–1640.

[19] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. “Variability modeling in the real: A perspective from the operating systems domain”. In: *Proceedings of the International Conference Automated Software Engineering (ASE)*. Antwerp, Belgium: ACM Press, 2010, pp. 73–82.

[20] John Bergey, Liam O’Brian, and Dennis Smith. *Mining existing assets for software product lines*. Tech. rep. CMU/SEI-2000-TN-008. Pittsburgh, PA: SEI, 2000.

[21] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. “Fair and balanced?: Bias in bug-fix datasets”. In: *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. Amsterdam, The Netherlands: ACM, 2009, pp. 121–130.

[22] Eric Bodden, Mira Mezini, Claus Brabrand, Társis Tolêdo, Márcio Ribeiro, and Paulo Borba. “SPLlift - Statically analyzing software product lines in minutes instead of years”. In: *Proceedings of the Conference Programming Language Design and Implementation (PLDI)*. ACM Press, 2013, pp. 355–364.

[23] Jan Bosch. *Design and use of software architecture: Adopting and evolving a product-line approach*. Harlow, England: Addison-Wesley, 2000.

[24] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, and Paulo Borba. "Intraprocedural dataflow analysis for software product lines". In: *Proceedings of the International Conference Aspect-Oriented Software Development (AOSD)*. Potsdam, Germany: ACM Press, 2012, pp. 13–24.

[25] *CDLTools*. `https://bitbucket.org/tberger/cdltools`.

[26] Paul Clements and Linda Northrop. *Software product lines: Practices and patterns*. Boston, MA: Addison-Wesley, 2001.

[27] Krzysztof Czarnecki. "Overview of generative software development". In: *Proceedings of Unconventional Programming Paradigms (UPP)*. Vol. 3566. LNCS. Springer-Verlag, 2004, pp. 313–328.

[28] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: Methods, tools, and applications*. Boston, MA: Addison-Wesley, 2000.

[29] Krzysztof Czarnecki and Krzysztof Pietroszek. "Verifying feature-based model templates against well-formedness OCL constraints". In: *Proceedings of the International Conference Generative Programming and Component Engineering (GPCE)*. Portland, Oregon: ACM Press, 2006, pp. 211–220.

[30] Krzysztof Czarnecki and Eisenecker Ulrich. "Components and Generative Programming". In: *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. Ed. by Oscar Nierstrasz and Michel Lemoine. Vol. 1687. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1999, pp. 2–19.

[31] Jean-Marc Davril, Edouard Delfosse, Negar Hariri, Mathieu Acher, Jane Cleland-Huang, and Patrick Heymans. "Feature model extraction from large collections of informal product descriptions". In: *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. Saint Petersburg, Russia: ACM Press, 2013, pp. 290–300.

[32] Deepak Dhungana and Paul Grünbacher. "Understanding decision-oriented variability modelling". In: *1st workshop on analyses of software product lines, in collocation with the 12th international software product line conference*. Limerick, Ireland, 2008.

[33] Deepak Dhungana, Paul Grünbacher, and Rick Rabiser. "The DOPLER meta-tool for decision-oriented variability modeling: A multiple case study". In: *Automated Software Engineering* 18.1 (2011), pp. 77–114.

[34] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "A robust approach for variability extraction from the Linux build system". In: *Proceedings of the International Software Product Line Conference (SPLC)*. Salvador, Brazil: ACM Press, 2012.

[35] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. “Understanding Linux feature distribution”. In: *Proceedings of the Workshop on Modularity In Systems Software (MISS)*. Potsdam, Germany, 2012.

[36] Slawomir Duszynski. “A scalable goal-oriented approach to software variability recovery”. In: *Proceedings of the International Software Product Line Conference (SPLC)*. Munich, Germany: ACM, 2011, pp. 1–8.

[37] Slawomir Duszynski, Jens Knodel, and Martin Becker. “Analyzing the source code of multiple software variants for reuse potential”. In: *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. Los Alamitos, CA: IEEE Computer Society, 2011, pp. 303–307.

[38] Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. “Do crosscutting concerns cause defects?” In: *IEEE Transactions on Software Engineering (TSE)* 34.4 (2008), pp. 497–515.

[39] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. “Selecting empirical methods for software engineering research”. In: *Guide to advanced empirical software engineering*. Ed. by Forrest Shull, Janice Singer, and Dag I. K. Sjøberg. Springer London, 2008, pp. 285–311.

[40] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. “Locating features in source code”. In: *IEEE Transactions on Software Engineering (TSE)* 29 (2003), pp. 210–224.

[41] Thomas Eisenbarth and Daniel Simon. “Guiding feature asset mining for software product line development”. In: *Proceedings of the International Workshop on Product Line Engineering - The Early Steps (PLEES)*. 2001, pp. 15–18.

[42] Michael Ernst, Greg Badros, and David Notkin. “An empirical analysis of C preprocessor use”. In: *IEEE Transactions on Software Engineering (TSE)* 28.12 (2002), pp. 1146–1170.

[43] *FARCE*. `https://bitbucket.org/tberger/farce`.

[44] D. Faust and C. Verhoef. “Software product line migration and deployment”. In: *Software: Practice and Experience* 33.10 (2003), pp. 933–955.

[45] Henry Ford. *My life and work*. Cosimo, Inc., 2007.

[46] Martin Fowler and Kent Beck. *Refactoring. Improving the design of existing code*. Boston, MA: Addison-Wesley, 1999.

[47] Michael W. Godfrey and Qiang Tu. “Evolution in Open Source Software: A case study”. In: *Proceedings of the International Conference Software Maintenance (ICSM)*. Los Alamitos, CA, USA: IEEE Computer Society, 2000.

[48] Negar Hariri, Carlos Castro-Herrera, Mehdi Mirakhorli, Jane Cleland-Huang, and Bamshad Mobasher. “Supporting domain analysis through mining and recommending features from online product listings”. In: *IEEE Transactions on Software Engineering (TSE)* 39.12 (2013), pp. 1736–1752.

[49] Richard C. Holt, Michael W. Godfrey, and Andrew J. Malton. "The build/comprehend pipelines". In: *Proceedings of the Second ASERC Workshop on Software Architecture*. 2003.

[50] Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. "A user survey of configuration challenges in Linux and eCos". In: *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. Leipzig, Germany: ACM Press, 2012, pp. 149–155.

[51] Hans Peter Jepsen, Jan Gaardsted Dall, and Danilo Beuche. "Minimally invasive migration to software product lines". In: *Proceedings of the International Software Product Line Conference (SPLC)*. Kyoto, Japan, 2007, pp. 203–211.

[52] Zhen Ming Jiang and Ahmed E. Hassan. "A framework for studying clones in large software systems". In: *Proceedings of the International Workshop Source Code Analysis and Manipulation (SCAM)*. Paris, France, 2007, pp. 203–212.

[53] Kyo Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-21. Pittsburgh, PA: SEI, 1990.

[54] Christian Kästner. "Virtual separation of concerns: Toward preprocessors 2.0". Logos Verlag Berlin, isbn 978-3-8325-2527-9. PhD thesis. Marburg, Germany: Department of Mathematics and Computer Science, Philipps University Marburg, May 2010.

[55] Christian Kästner, Sven Apel, and Martin Kuhlemann. "Granularity in software product lines". In: *Proceedings of the International Conference Software Engineering (ICSE)*. Leipzig: ACM Press, 2008, pp. 311–320.

[56] Christian Kästner, Sven Apel, Syed Saif ur Rahman, Marko Rosenmüller, Don Batory, and Gunter Saake. "On the impact of the optional feature problem: analysis and case studies". In: *Proceedings of the International Software Product Line Conference (SPLC)*. San Francisco, CA: ACM Press, 2009, pp. 181–190.

[57] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. "Type checking annotation-based product lines". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 21.3 (2012), Article 14.

[58] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. "Variability mining: Consistent semi-automatic detection of product-line features". In: *IEEE Transactions on Software Engineering (TSE)* 40.1 (Jan. 2014), pp. 67–82.

[59] Christian Kästner, Paolo G. Giarrusso, and Klaus Ostermann. "Partial preprocessing of C code for variability analysis". In: *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. New York: ACM Press, 2011, pp. 137–140.

[60] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. "Variability-aware parsing in the presence of lexical macros and conditional compilation". In: *Proceedings of the International Conference Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. Portland: ACM Press, Oct. 2011, pp. 805–824.

[61] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. “A variability-aware module system”. In: *Proceedings of the International Conference Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. New York: ACM Press, 2012.

[62] *KBuildMiner*. `http : / / code . google . com / p / variability / wiki / PresenceConditionsExtraction`.

[63] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. “TypeChef: Toward type checking #ifdef variability in C”. In: *Proceedings of the GPCE Workshop on Feature-Oriented Software Development (FOSD)*. Eindhoven, The Netherlands: ACM Press, 2010, pp. 25–32.

[64] Charles W. Krueger. “Software reuse”. In: *ACM Compututer Surveys* 24.2 (June 1992), pp. 131–183.

[65] Gary Kumfert and Tom Epperly. *Software in the DOE: The hidden overhead of “the build”*. Tech. rep. Lawrence Livermore National Lab., 2002.

[66] Germaschewski Kumfert and S. Ravnborg. “Kernel configuration and building in Linux 2.5”. In: *Linux symposium* (2003), pp. 197–212.

[67] DucMinh Le, Hyesun Lee, KyoChul Kang, and Lee Keun. “Validating consistency between a feature model and its implementation”. In: *Safe and Secure Software Reuse*. Ed. by John Favaro and Maurizio Morisio. Vol. 7925. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 1–16.

[68] Thomas Leich, Sven Apel, and Laura Marnitz. “Tool support for feature-oriented software development: FeatureIDE: an eclipse-based approach”. In: *Proceedings OOPSLA Workshop on Eclipse Technology eXchange (ETX)*. San Diego, CA: ACM Press, 2005, pp. 55–59.

[69] 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 International Conference Software Engineering (ICSE)*. Cape Town, South Africa: ACM Press, 2010, pp. 105–114.

[70] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. “Scalable analysis of variable software”. In: *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. Saint Petersburg, Russia: ACM Press, 2013, pp. 81–91.

[71] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. “CP-Miner: a tool for finding copy-paste and related bugs in operating system code”. In: *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*. San Francisco, CA: USENIX Association, 2004, pp. 20–20.

[72] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. “Evolution of the Linux kernel variability model”. In: *Proceedings of the International Software Product Line Conference (SPLC)*. Jeju Island, South Korea: Springer, 2010, pp. 136–150.

[73] *LVAT*. `http://code.google.com/p/linux-variability-analysis-tools`.

[74] Mike Mannion. “Using first-order logic for product line model validation.” In: *Proceedings of the International Software Product Line Conference (SPLC)*. Vol. 2379. Lecture Notes in Computer Science. San Diego, CA: Springer, 2002, pp. 176–187.

[75] Malcolm D. McIlroy. “Mass-produced software components”. In: *Software engineering: concepts and techniques* (1968), pp. 88–98.

[76] Shane McIntosh, Bram Adams, and Ahmed Hassan. “The evolution of Java build systems”. In: *Empirical software engineering* (2011), pp. 1–31.

[77] Shane McIntosh, Bram Adams, and Ahmed E. Hassan. “The evolution of ANT build systems”. In: *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. 2010, pp. 42 –51.

[78] Shane McIntosh, Bram Adams, Thanh H. D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. “An empirical study of build maintenance effort”. In: *Proceedings of the International Conference Software Engineering (ICSE)*. Honolulu, Hawaii, 2011, pp. 1167 –1169.

[79] Marcílio Mendonça, Andrzej Wąsowski, and Krzysztof Czarnecki. “SAT-based analysis of feature models is easy”. In: *Proceedings of the International Software Product Line Conference (SPLC)*. San Francisco, CA: ACM Press, 2009, pp. 231–240.

[80] Andreas Metzger, Klaus Pohl, Patrick Heymans, Pierre-Yves Schobbens, and Germain Saval. “Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis”. In: *Proceedings of the International Requirements Engineering Conference (RE)*. Los Alamitos, CA: IEEE Computer Society, 2007, pp. 243–253.

[81] Kai Germaschewski Sam Ravnborg Michael Elizabeth Chastain and Jan Engelhardt. *Linux Kernel Makefiles*. Tech. rep. Available at /Documentation/Kbuild/makefiles.txt. 2011.

[82] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. “Mining configuration constraints: static analyses and empirical results”. In: *Proceedings of the International Conference Software Engineering (ICSE)*. Hyderabad, India, 2014, pp. 140–151.

[83] Sarah Nadi, Christian Dietrich, Reinhard Tartler, Richard C. Holt, and Daniel Lohmann. “Linux variability anomalies: What causes them and how do they get fixed?” In: *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. MSR ’13. San Francisco, CA, USA: IEEE Press, 2013, pp. 111–120.

[84] Sarah Nadi and Ric Holt. “Make it or break it: Mining anomalies in Linux Kbuild”. In: *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. 2011, pp. 315–324.

[85] Sarah Nadi and Ric Holt. “Mining Kbuild to detect variability anomalies in Linux”. In: *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. Los Alamitos, CA: IEEE Computer Society, 2012, pp. 107–116.

[86] Sarah Nadi and Ric Holt. "The Linux kernel: A case study of build system variability". In: *Journal of Software: Evolution and Process* (2013). Early online view. `http://dx.doi.org/10.1002/smr.1595`.

[87] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. "Consistency management with repair actions". In: *Proceedings of the International Conference Software Engineering (ICSE)*. May 2003, pp. 455–464.

[88] George V. Neville-Neil. "Kode vicious system changes and side effects". In: *Communications of the ACM* 52.4 (2009), pp. 25–26.

[89] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. "Exploring variability-aware execution for testing plugin-based web applications". In: *Proceedings of the International Conference Software Engineering (ICSE)*. Hyderabad, India, 2014.

[90] *Online appendix*. `http://gsd.uwaterloo.ca/farce`.

[91] Yoann Padioleau. "Parsing C/C++ code without pre-processing". In: *Proceedings of the International Conference Compiler Construction (CC)*. York, UK: Springer, 2009, pp. 109–125.

[92] Nicolas Palix, Julia Lawall, and Gilles Muller. "Tracking code patterns over multiple software versions with Herodotos". In: *Proceedings of the International Conference Aspect-Oriented Software Development (AOSD)*. Rennes and Saint-Malo, France: ACM, 2010, pp. 169–180.

[93] David L. Parnas. "On the criteria to be used in decomposing systems into modules". In: *Communications of the ACM* 15.12 (Dec. 1972), pp. 1053–1058.

[94] Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wąsowski, and Paulo Borba. "Coevolution of variability models and related artifacts: A case study from the Linux kernel". In: *Proceedings of the International Software Product Line Conference (SPLC)*. Tokyo, Japan: ACM Press, 2013, pp. 91–100.

[95] Leonardo Passos, Marko Novakovic, Yingfei Xiong, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. "A study of non-Boolean constraints in variability models of an embedded operating system". In: *Proceedings of the International Software Product Line Conference (SPLC)*. Munich, Germany: ACM Press, 2011, 2:1–2:8.

[96] Ariel Rabkin and Randy Katz. "Static extraction of program configuration options". In: *Proceedings of the International Conference Software Engineering (ICSE)*. Waikiki, Honolulu, HI, USA: ACM Press, 2011, pp. 131–140.

[97] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. "Using symbolic evaluation to understand behavior in configurable software systems". In: *Proceedings of the International Conference Software Engineering (ICSE)*. Cape Town, South Africa: ACM Press, 2010, pp. 445–454.

[98] Roland McGrath Richard M. Stallman and Paul D. Smith. "The GNU Make Manual". In: (2010). Available at `http://www.gnu.org/software/make/manual/`.

[99] Gregorio Robles, Jesus M. Gonzalez-Barahona, and Juan J. Merelo. “Beyond source code: The importance of other artifacts in software development (a case study)”. In: *Journal of systems and software* 79.9 (2006), pp. 1233–1248.

[100] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. “Managing cloned variants: a framework and experience”. In: *Proceedings of the International Software Product Line Conference (SPLC)*. Tokyo, Japan: ACM, 2013, pp. 101–110.

[101] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. “Generic semantics of feature diagrams”. In: *Computer networks* 51.2 (2007). Feature Interaction, pp. 456 –479.

[102] Steven She and Thorsten Berger. “Formal semantics of the Kconfig language”. Technical Note. Available at `eng.uwaterloo.ca/~shshe/kconfig_semantics.pdf`.

[103] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. “Reverse engineering feature models”. In: *Proceedings of the International Conference Software Engineering (ICSE)*. Waikiki, Honolulu, HI, USA: ACM Press, 2011, pp. 461–470.

[104] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. “The variability model of the Linux kernel”. In: *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. Essen: University of Duisburg-Essen, 2010, pp. 45–51.

[105] Daniel Simon and Thomas Eisenbarth. “Evolutionary introduction of software product lines”. In: *Proceedings of the International Software Product Line Conference (SPLC)*. Vol. 2379. Springer, 2002, pp. 272–282.

[106] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. “Is the Linux kernel a software product line?” In: *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL)*. 2007.

[107] Julio Sincero and Wolfgang Schröder-Preikschat. “The Linux kernel configurator as a feature modeling tool”. In: *Proceedings of the International Software Product Line Conference (SPLC)*. 2008, pp. 257–260.

[108] Julio Sincero, Reinhard Tartler, Christopher Egger, Wolfgang Schröder-Preikschat, and Daniel Lohmann. “Facing the Linux 8000 Feature Nightmare”. In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. 2010.

[109] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. “Efficient extraction and analysis of preprocessor-based variability”. In: *Proceedings of the International Conference Generative Programming and Component Engineering (GPCE)*. Eindhoven, The Netherland: ACM Press, 2010, pp. 33–42.

[110] Marco Sinnema and Sybren Deelstra. “Classifying variability modeling techniques”. In: *Information and software technology* 49.7 (2007), pp. 717 –739.

[111] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. "When do changes induce fixes?" In: *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. St. Louis, Missouri: ACM, 2005, pp. 1–5.

[112] Henry Spencer and Geoff Collyer. "#ifdef considered harmful or portability experience with C news". In: *Proceedings usenix conference*. Berkeley, CA: USENIX Association, 1992, pp. 185–198.

[113] Diomidis Spinellis. "A tale of four kernels". In: *Proceedings of the International Conference Software Engineering (ICSE)*. Leipzig, Germany: ACM, 2008, pp. 381–390.

[114] Christoph Stoermer and Liam O'Brien. "MAP – Mining architectures for product line evaluations". In: *Proceedings of the Working Conference Software Architecture (WICSA)*. Washington, DC: IEEE Computer Society, 2001, pp. 35–44.

[115] Tijs van der Storm. "Variability and component composition". In: *Proceedings of the International Conference Software Reuse (ICSR)*. Vol. 3107. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer, 2004, pp. 157–166.

[116] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. "A taxonomy of variability realization techniques". In: *Software–practice & experience* 35.8 (2005), pp. 705–754.

[117] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N. Nguyen. "Build code analysis with symbolic evaluation". In: *Proceedings of the International Conference Software Engineering (ICSE)*. Zurich, Switzerland: IEEE Computer Society, 2012, pp. 650–660.

[118] 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 European Conference on Computer Systems (EuroSys)*. Salzburg: ACM Press, 2011, pp. 47–60.

[119] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Dead or alive: Finding zombie features in the Linux kernel". In: *Proceedings of the GPCE Workshop on Feature-Oriented Software Development (FOSD)*. Denver, Colorado: ACM Press, 2009, pp. 81–86.

[120] Sahil Thaker. "Design and analysis of multidimensional program structures". MA thesis. The Department of Computer Science. The University of Texas at Austin, 2006.

[121] Sahil Thaker, Don Batory, David Kitchin, and William Cook. "Safe composition of product lines". In: *Proceedings of the International Conference Generative Programming and Component Engineering (GPCE)*. Salzburg, Austria: ACM Press, 2007, pp. 95–104.

[122] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. "A classification and survey of analysis strategies for software product lines". In: *ACM Computing Surveys* (2014). to appear.

[123] Thomas Thüm, Don Batory, and Christian Kästner. "Reasoning about edits to feature models". In: *Proceedings of the International Conference Software Engineering (ICSE)*. Vancouver, Canada: IEEE Computer Society, 2009, pp. 254–264.

[124] Salvador Trujillo, Don Batory, and Oscar Diaz. “Feature refactoring a multi-representation program into a product line”. In: *Proceedings of the International Conference Generative Programming and Component Engineering (GPCE)*. Portland, OR: ACM Press, 2006, pp. 191–200.

[125] Bart Veer and John Dallaway. *The eCos component writer’s guide*. `ecos.sourceware.org/ecos/docs-latest/cdl-guide/cdl-guide.html`.

[126] Jules White, Douglas Schmidt, David Benavides, Pablo Trinidad, and Antonio Cortés. “Automated diagnosis of product-line configuration errors in feature models”. In: *Proceedings of the International Software Product Line Conference (SPLC)*. IEEE Computer Society, 2008, pp. 225–234.

[127] Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. “Generating range fixes for software configuration”. In: *Proceedings of the International Conference Software Engineering (ICSE)*. IEEE Computer Society, 2012, pp. 58–68.

[128] Christoph Zengler and Wolfgang Küchlin. “Encoding the Linux kernel configuration in propositional logic”. In: *Proceedings of European Conference on Artificial Intelligence ECAI (Workshop on Configuration)*. 2010, pp. 51–56.

[129] Bo Zhang and Martin Becker. “Code-based variability model extraction for software product line improvement”. In: *Proceedings of the International Software Product Line Conference (SPLC)*. Salvador, Brazil: ACM Press, 2012, pp. 91–98.

[130] Sai Zhang and Michael D. Ernst. “Automated diagnosis of software configuration errors”. In: *Proceedings of the International Conference Software Engineering (ICSE)*. San Francisco, CA, USA: IEEE Press, 2013, pp. 312–321.

[131] Tewfik Ziadi, Luz Frias, Marcos A.A. da Silva, and Mikal Ziane. “Feature identification from the source code of product variants”. In: *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. 2012, pp. 417–422.

[132] Roman Zippel and contributors. “kconfig-language.txt”. available in the kernel tree at `www.kernel.org`.