

Analysis and Transformation of Configurable Systems

Jörg Liebig

2015-04-30

Eingereicht an der Fakultät für Informatik und Mathematik
der Universität Passau als Dissertation zur Erlangung des
Grades eines Doktors der Naturwissenschaften

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

Betreuer / Advisors:

Prof. Dr.-Ing. Sven Apel
University of Passau, Germany

Prof. Christian Lengauer, Ph.D.
University of Passau, Germany

Externer Gutachter / External Examiner:

Prof. Dr. Paul Grünbacher
University of Linz, Austria



Abstract

Static analysis tools and transformation engines for source code belong to the standard equipment of a software developer. Their use simplifies a developer's everyday work of maintaining and evolving software systems significantly and, hence, accounts for much of a developer's programming efficiency and programming productivity. This is also beneficial from a financial point of view, as programming errors are early detected and avoided in the the development process, thus the use of static analysis tools reduces the overall software-development costs considerably.

In practice, software systems are often developed as configurable systems to account for different requirements of application scenarios and use cases. To implement configurable systems, developers often use compile-time implementation techniques, such as preprocessors, by using `#ifdef` directives. Configuration options control the inclusion and exclusion of `#ifdef`-annotated source code and their selection/deselection serve as an input for generating tailor-made system variants on demand. Existing configurable systems, such as the LINUX kernel, often provide thousands of configuration options, forming a huge configuration space with billions of system variants.

Unfortunately, existing tool support cannot handle the myriads of system variants that can typically be derived from a configurable system. Analysis and transformation tools are not prepared for variability in source code, and, hence, they may process it incorrectly with the result of an incomplete and often broken tool support.

We challenge the way configurable systems are analyzed and transformed by introducing variability-aware static analysis tools and a variability-aware transformation engine for configurable systems' development. The main idea of such tool support is to exploit commonalities between system variants, reducing the effort of analyzing and transforming a configurable system. In particular, we develop novel analysis approaches for analyzing the myriads of system variants and compare them to state-of-the-art analysis approaches (namely sampling). The comparison shows that variability-aware analysis is complete (with respect to covering the whole configuration space), efficient (it outperforms some of the sampling heuristics), and scales even to large software systems. We demonstrate that variability-aware analysis is even practical when using it with non-trivial case studies, such as the LINUX kernel.

On top of variability-aware analysis, we develop a transformation engine for C, which respects variability induced by the preprocessor. The engine provides three common refactorings (rename identifier, extract function, and inline function) and overcomes shortcomings (completeness, use of heuristics, and scalability issues) of existing engines, while still being semantics-preserving with respect to all variants and being fast, providing an instantaneous user experience. To validate semantics preservation, we extend a standard testing approach for

refactoring engines with variability and show in real-world case studies the effectiveness and scalability of our engine.

In the end, our analysis and transformation techniques show that configurable systems can efficiently be analyzed and transformed (even for large-scale systems), providing the same guarantees for configurable systems as for standard systems in terms of detecting and avoiding programming errors.

Acknowledgements

Pursuing a Ph.D. in computer science is a long and difficult endeavor during which the Ph.D. student is faced with many challenging tasks. Luckily, I could rely on the support of many persons when solving them.

First, I would like to thank Sven Apel. After having gotten my diploma from the University of Magdeburg, Sven invited me to Passau and encouraged me to start a Ph.D. Working with Sven was a great pleasure for me because his dedication and enthusiasm to scientific work is contagious and provides a great source of inspiration. During many discussions he helped me to understand the big picture of my work, fostered my way of scientific writing, and directed me towards interesting research questions. To me, Sven is a scientist's best role model.

Second, I'm grateful for the support of Christian Lengauer. Despite the fact that Christian's research group focuses on a different aspect of programming, he has always given me the freedom to follow my own research direction and has supported me unconditionally in financial and organizational aspects. Coming to Passau and working at his group was a big step for me. But I soon realized that Passau was not only a place to work, but also a place to live and I have always felt welcome. In this context I would like to particularly thank Eva Reichhart.

Furthermore, I would like to thank many colleagues and students for their collaboration, for discussions, and for their support regarding different aspects of this thesis: Jens Dörre, Florian Garbe, Armin Größlinger, Claus Hunsen, Andreas Janker, Christian Kästner, Sergiy Kolesnikov, Olaf Leßenich, Christopher Resch, Wolfgang Scholz, Sandro Schulze, Janet Siegmund, Norbert Siegmund, Andreas Simbürger, Reinhard Tartler, and Alexander von Rhein. In particular, I'm grateful to have met Christian Kästner. Christian has been a constant source of support and I highly value his feedback. By challenging my research work down to its foundations, Christian helped me to improve it in a way I could have never imagined.

Last but not least, I have to thank Elke, my wife, and my family for their continuous support and love that helped through hard times of my thesis.

Contents

List of Figures	vii
List of Tables	ix
List of Acronyms	xi
1 Introduction	1
1.1 Contributions	3
1.2 Outline	5
2 Background	7
2.1 Software Product Lines and Configurable Software Systems	7
2.1.1 Configuration Knowledge	9
2.1.2 Implementing Configurable Systems	15
2.2 Static Analysis	18
2.3 Refactoring	21
3 Understanding Preprocessor Annotations	25
3.1 Methodology	26
3.1.1 Goals	26
3.1.2 Questions	30
3.1.3 Metrics	34
3.2 Analysis	39
3.3 Interpretation and Discussion	42
3.3.1 Answering the Research Questions	42
3.3.2 Threats to Validity	52
3.4 Related Work	54
3.5 Summary	58
4 Analyzing C Code with Preprocessor Annotations	61
4.1 Sampling-based Analysis and Variability-aware Analysis	62
4.1.1 Sampling-based Analysis	62
4.1.2 Variability-aware Analysis	66
4.2 Analysis Time	72
4.2.1 Empirical Study	73

Contents

4.3	Upfront Investment	81
4.3.1	Experience with Sampling	82
4.3.2	Patterns of Variability-aware Analyses	85
4.3.3	Variability-aware Intra-procedural Data-flow Framework	86
4.4	Error Detection	89
4.4.1	Evaluation	90
4.4.2	Experiment Setup	94
4.4.3	Results and Discussion	95
4.5	Perspectives	96
4.6	Related Work	97
4.7	Summary	102
5	Refactoring C Code with #ifdefs	105
5.1	State of the Art	106
5.2	Variability-aware Refactorings with Morpheus	110
5.2.1	Specification of Refactorings	110
5.3	Experiments	116
5.3.1	Subject Systems	116
5.3.2	Experiment Setup	117
5.3.3	Performance Results	118
5.3.4	Testing the Refactoring Engine	118
5.3.5	Perspectives of Variability-aware Refactoring	120
5.3.6	Threats to Validity	121
5.4	Related Work	122
5.5	Summary	124
6	Conclusion and Future Work	125
	Bibliography	129

List of Figures

2.1	Compile-time configuration in LINUX; adapted from Tartler [2013].	11
2.2	Definition of HOTPLUG_CPU with KCONFIG in LINUX [Tartler, 2013].	11
2.3	Examples of configuration-knowledge use in MAKEFILES in BUSYBOX.	12
2.4	Excerpt of the operating-system configuration with CPP in SQLITE.	13
2.5	Propositional formulae for the configuration knowledge presented in Figures 2.2, 2.3, and 2.4.	14
2.6	Example of configuration knowledge in GLIBC using non-boolean constraints.	15
2.7	A variable list implementation with CPP; taken from Liebig et al. [2010].	16
2.8	Implementation of the configuration option DLINKED using aspect-oriented and feature-oriented programming.	18
2.9	Code example with its corresponding AST (excerpt) and its CFG representation.	19
2.10	Examples of type errors in C.	20
3.1	Goal Question Metric (GQM) model of our study on preprocessor usage.	27
3.2	Trade-off between expressiveness, comprehension, and replication [Schulze et al., 2013a].	28
3.3	Excerpt of a CPP-extended ISO/IEC 9899 lexical C grammar; rules for preprocessor directives are highlighted (Line 7 and Line 10 to 14); <code>cppexp</code> is the condition; <code>nl</code> is a newline; <code>cppthenfunc</code> represents the <code>#endif</code> or alternative function definitions.	33
3.4	Examples of disciplined annotations in VIM.	34
3.5	Examples of undisciplined annotations in VIM.	35
3.6	Example of undisciplined annotations in XTERM.	36
3.7	Example of coarse-grained and fine-grained extensions.	38
3.8	Variability plots: lines of code vs configuration options (LOC/CO), lines of code vs lines of variable code (LOC/LVC), and configuration options vs lines of variable code (CO/LVC).	42
3.9	Complexity plots: configuration options vs scattering degree (CO/SD), configuration options vs tangling degree (CO/TD), and configuration options vs nesting depth (CO/ND).	46
3.10	Undisciplined parameter annotation (a) and two disciplined variants of it (b and c).	52

List of Figures

4.1	C code with preprocessor directives; the header file (left) contains one alternative and one optional definition; the C file (right) uses the definitions of the header file.	63
4.2	Running example for variability-aware analysis in C.	67
4.3	Excerpt of the variability-aware CFG of the running example in Figure 4.2.	70
4.4	CFG representation with compile-time and run-time variability.	71
4.5	Experimental setup.	76
4.6	Distribution of analysis times for BUSYBOX (times in milliseconds; logarithmic scale).	78
4.7	Distribution of analysis times for LINUX (times in milliseconds; logarithmic scale).	79
4.8	Distribution of analysis times for OPENSLL (times in milliseconds; logarithmic scale).	80
4.9	Number of variants vs analysis time for liveness analysis of BUSYBOX.	81
4.10	Patterns of variability-aware analysis illustrated using liveness analysis for an excerpt of our running example in Figure 4.2, including the variability-aware CFG.	85
4.11	Example of a dead store.	91
4.12	Example of a missing error handling.	91
4.13	Example of a double free.	92
4.14	Example of freeing a variable, which was allocated statically.	92
4.15	Example of an uninitialized variable being used.	93
4.16	Example of a case block without a terminating break statement.	93
4.17	Example of a non- void function with a missing return statement.	93
4.18	Example of dangling switch code.	94
5.1	Before (cf. Figure 5.1a) and after (cf. Figure 5.1b) applying RENAME IDENTIFIER in XCODE ; type error after renaming (cf. Figure 5.1c); before (cf. Figure 5.1d) and after (cf. Figure 5.1e) applying EXTRACT FUNCTION in ECLIPSE with the corresponding program outputs.	107
5.2	AST representation enriched with reference information of the RENAME-IDENTIFIER example in Figure 5.1a; Choice A represents a variable AST node providing a selection of two different definitions of variable global	112
5.3	Auxiliary functions of MORPHEUS that provide the interface to underlying variability-aware analyses and transformations.	113
5.4	Specification of RENAME IDENTIFIER	114
5.5	Specification of EXTRACT FUNCTION	115
5.6	Specification of INLINE FUNCTION	116

List of Tables

- 2.1 Terminology configurable systems vs software product lines. 8
- 2.2 Common refactorings in C and CPP code. 22

- 3.1 Analyzed software systems. 41
- 3.2 Results configuration knowledge (G1.1) and variable code base (G1.2) 44
- 3.3 Results of variability representation (G1.3). 48

- 4.1 Conditional symbol table (CST) at Line 6 of our running example in Figure 4.2. 69
- 4.2 Liveness-computation result of our running example in Figure 4.2. 72
- 4.3 Total times for analyzing the subject systems with each approach (time in seconds, with three significant digits) and speedups, showing the relative performance improvement of variability-aware analysis compared to sampling-based analyses; a speedup lower than 1.0 reflects a slowdown. 77
- 4.4 Times for sample-set computation (time in seconds, with three significant digits). 84
- 4.5 Error coverage of selected sampling approaches. 95

- 5.1 Classification of refactoring support in integrated development environments. 111
- 5.2 Measurement results for `RENAME IDENTIFIER`, `EXTRACT FUNCTION`, and `INLINE FUNCTION` for all three subject systems (`BUSYBOX`, `OPENSSL`, and `SQLITE`); for each refactoring we give the mean \pm standard deviation, the maximum of refactoring times in milliseconds, and the number of affected configurations; box-plots show the corresponding distributions (excluding outliers). 119

List of Acronyms

AST	Abstract Syntax Tree
BDD	Binary Decision Diagram
CFG	Control-flow Graph
CSP	Constraint Satisfaction Problem
GQM	Goal Question Metric
IDE	Integrated Development Environment
SAT	Boolean Satisfiability Problem
SPL	Software Product Line
XML	Extensible Markup Language

1 Introduction

Today's software systems have grown to a level of complexity that make them difficult to understand and maintain for developers. It is not only their size, but also their variability that contributes to their complexity. Most software systems are not developed only to be applied in a single scenario or use case, but in a multitude of them. Therefore, software systems are typically developed as configurable systems, of which specific variants can be generated to match the requirements of different application scenarios and use cases.

Practitioners and researchers have proposed many different techniques for developing configurable systems. Among these are preprocessors [Kernighan and Ritchie, 1988; Boucher et al., 2010], plugins [Johnson and Foote, 1988], components [Szyperski, 2002], aspects [Kiczales et al., 1997], and feature modules [Prehofer, 1997; Batory et al., 2004]. Although all these approaches differ in technical details, they share two common characteristics: a configurable code base and a representation of configuration knowledge. Based on configuration options, the configurable code base implements variable code artifacts that can be used in different variants. Configuration knowledge describes valid combinations of configuration options and states the configuration space of a system. Given a valid configuration, a generator generates a single variant by combining the reusable code artifacts in a predefined fashion. The generation step typically involves source-code transformations performed during the compilation of a program.

We use the C preprocessor (CPP) as a representative implementation technique for configurable systems and review several aspects of analyzing and transforming configurable systems. The preprocessor is a frequently applied tool for the implementation of configurable systems, and it has been used in many industrial projects such as HP's printer-driver implementation OWEN [Pearse and Oman, 1997; Refstrup, 2009], Wilkon's remote control systems [Pech et al., 2009], Danfoss' software system for frequency converters named VLT[®] [Jepsen and Beuche, 2009; Zhang et al., 2013], and NASA's core flight software (CFS) [Ganesan et al., 2009], as well as in open-source projects such as the LINUX kernel,¹ the BUSYBOX tool suite,² and the Apache web server.³ By means of preprocessor directives, developers define explicit source-to-source transformations to make the code base configurable. Using directives such as `#ifdef X` and `#endif`, developers annotate code fragments with configuration options (here X) that allow for the generation of different configurations, given the end-users' selection of configuration options. Based on a configuration, different variants of a configurable system can be generated by applying the generator (CPP), which evaluates preprocessor directives (which are basically

¹<http://kernel.org>

²<http://busybox.net>

³<http://httpd.apache.org>

boolean or arithmetic expressions) and generates a specific variant by including and excluding code fragments.

Although configurable systems and corresponding development techniques have been adopted widely in practice, they often lack a tool infrastructure to support their development. This means that dedicated tools for program analysis and program transformation are missing; existing tools usually aim at the development of software systems that are not configurable. At the same time, the use of powerful tool support is often indispensable for the production of complex software systems. Nowadays, it is not possible to imagine the development of current software systems without them. Despite C++ has been used and adopted widely in practice, it has been often criticized by the research community, and anecdotal evidence from industrial practice illustrates the developers' struggle with it. Criticism concerns maintenance (lack of existing refactoring and transformation tools) [Garrido and Johnson, 2003; Lohmann et al., 2009; McCloskey and Brewer, 2005], diagnostics (lack of analysis tools for error and bug detection) [Favre, 1997, 1996], and readability (code obfuscation) [Spencer and Collyer, 1992; Favre, 1995]. Although the criticism partially dates back more than 20 years, there is no evidence of major improvements.

The main advantage of powerful tool support is the adjoint ability to find or avoid programming errors early in the development process. A variety of studies (e.g., [Ko and Myers, 2003]) of software-development costs illustrate that most programming errors are introduced during coding of a system and remain there until found in subsequent phases of the development process (e.g., testing or quality assurance) or even only after a system's release. Unfortunately, the costs of error repair increase significantly from coding phase to release [Madachy, 1994].⁴ Even higher costs occur, because configurable systems are often shipped to end users in the form of source code instead of a ready-to-use product, and so existing errors are often found by end users first,⁵ who are typically overwhelmed by cryptic error messages or program crashes. The main reason is that configuration, variant generation, and deployment usually happen on the user side, and this may lead to various ways of usage and integration of a system in a way unforeseen by its developers. This poses a huge risk, because configurable systems such as OPENSSL, a cryptographic library that implements different algorithms for secure communication protocols, can reveal security flaws after deployment.

At the same time, evolving and maintaining configurable systems is a burden not to be underestimated. Software changes are inevitable and, thus, after primal deployment, changes such as error corrections, enhancements, and improvements, are usually applied to a system for which automated transformation tools are usually employed. Since software maintenance takes place over a long time period, the resources required for maintenance usually exceed

⁴It is hard to predict or give representative numbers, because programming errors depend on a multitude of aspects such as project type, project size, used programming languages, number of developers, and so forth. However, it is commonly acknowledged that repairing an error after release is many times more expensive than its repair during coding.

⁵Different forms of programming errors are frequently reported by end users as bug reports in dedicated forums or bug-tracking systems. For OpenSSL the corresponding bug tracker is available at <http://rt.openssl.org/NoAuth/Buglist.html>.

the ones of the initial development of a system and sum up to 75–80 % of the total costs of a system [Lientz et al., 1978]. However, trustworthy tools for automated transformations of configurable systems are still missing, forcing developers to apply error-prone transformations manually. In the case of the C preprocessor, transformation engines of current development tools such as ECLIPSE or VISUAL STUDIO, still struggle with preprocessor directives as they do not incorporate configuration knowledge in the transformation process or apply some sort of heuristics to reason about them. Thus, even simple changes are likely to introduce errors (e.g., type errors).

The main advantage of configurable systems—the possibility for adaption to different application scenarios and use cases—is also their main drawback. Even small configurable systems can have billions of possible variants, since the number of valid configurations of a system can exponentially grow with the number of configuration options available. Analyzing or transforming all these variants brute-force (i.e., by handling each variant separately) usually does not scale, since the computational effort of applying a brute-force analysis or transformation typically exceeds available hardware resources in terms of time and space by several orders of magnitude. Additionally, the configuration space of a configurable system tends to grow fast. For example, the number of configuration options of the LINUX kernel more than doubled between 2005 and 2010 (from 5338 in version 2.6.12 to 11 223 in version 2.6.35) [Tartler et al., 2011].

1.1 Contributions

We converge to all problems just stated using the following two research questions: first, how can we efficiently analyze all variants of a configurable system? Second, how do we ensure that a transformation preserves the behavior of all variants? By answering both questions we make significant contributions to the development practice of configurable systems. During our discussions we focus on systems developed with compile-time variability. This way we cover much of a system’s configuration space and provide hands-on improvements for developers of configurable systems.

We analyze the variability of configurable systems in a large-scale empirical study, because a system’s variability affects its analysis and transformation significantly. We define a set of metrics based on different implementation aspects of variability, such as scattering (distribution of variability implementation), tangling (mixture of configuration options), and nesting (hierarchy of preprocessor directives). We apply the metrics to 42 different, well-known software systems to infer consequences on the analysis and transformation challenge of configurable systems.

To avoid redundant computations in the analysis of a configurable system, differences and similarities of the system have to be represented in a common, compact form. Analysis and transformation techniques that can handle such a representation are able to reason about it efficiently. For example, a type checker that incorporates variability is able to check all variants for type errors by traversing the input representation only once. Unfortunately, such a compact representation is not always easy to create. In the case of the C preprocessor, it is actually

quite difficult to create a representation, which includes both common and variable code (using `#ifdefs`). The main reason is that C is a token-based text processor that supports the manipulation of arbitrary text fragments, which can lead, if misused, to violations of the code structure and of naming conventions. For example, it is possible to annotate single tokens of the source code, such as an opening or closing brace. On the one hand, developers benefit from using such undisciplined use of the preprocessor to develop configurable source code at a fine grain, by avoiding code duplication. On the other hand, undisciplined annotations impair the development of efficient analysis and transformation tools, since these tools require well-structured inputs and cannot reason about arbitrary changes to source code.

There are two competing approaches for analyzing configurable systems efficiently: sampling-based analysis and variability-aware analysis. Each of them stands for a different perspective regarding the problem of scaling an analysis to possibly billions of variants. Sampling-based analysis employs strategies of reducing the number of variants for analysis to a reasonable subset of all variants of the configurable system in question. Variability-aware analysis rests on compact representations of configurable systems and, thus, exploits similarities between variants by incorporating configuration knowledge during the analysis process. We analyze both approaches with respect to three criteria (analysis time, effort, and configuration coverage of analysis results) revealing individual strengths and weaknesses. The outcome serves as a guideline for preferring one approach to the other when analyzing large-scale, configurable systems.

Myriads of system variants pose an important challenge when transforming configurable systems as important properties of program transformations, such as semantic equivalence, have to be ensured. Based on compact representations and scalable analysis techniques for configurable systems, we develop a scalable refactoring engine for C incorporating `#ifdefs`. To this end, we enrich existing specifications for C refactorings with configuration knowledge and test the feasibility of our engine in real-world scenarios.

Specifically, we make the following four contributions:

1. Driven by practical program analysis and transformation scenarios, we introduce a set of metrics that capture code-quality properties, such as scattering, tangling, and nesting, of configurable software systems developed with the C preprocessor. The results of an empirical case study based on these metrics guide our discussions throughout this thesis and allow us to infer general recommendations on preprocessor use in software development. Three major insights are: (1) on average 21 % of a system's code base can be configured with `#ifdefs`, (2) developers use `#ifdefs` to annotate code at a coarse grain (i.e., annotating entire function definitions) as well as at a fine grain (i.e., annotating subexpressions or function parameters), and (3) configurable code is often scattered across the entire code base.
2. We define the concept of disciplined preprocessor annotations, which can serve as a basis for the generation of compact representations for configurable systems. We show that enforcing preprocessor discipline does not overly restrict preprocessor use in practice, and we provide a solution for handling undisciplined use of the preprocessor.

3. We implement and compare variability-aware analysis with three state-of-the-art sampling strategies, revealing principles and obstacles when applying both analysis approaches in practice. We identify three general patterns of variability-aware analysis and propose a specific implementation of a variability-aware analysis framework using these patterns. The patterns as well as the framework simplify the development of further analysis techniques for configurable systems at the scale of the LINUX kernel.
4. Refactoring in the presence of preprocessor directives is challenging, since the behavior of all variants that can be derived from a system has to be preserved. We demonstrate the feasibility of variability-aware refactoring by implementing three common refactorings `RENAME IDENTIFIER`, `EXTRACT FUNCTION`, and `INLINE FUNCTION`, as part of a C refactoring engine. By means of experiments using real-world, configurable systems, we show the practicability of variability-aware transformations for such systems for the first time. Finally, we provide a means for verification and validation of our refactoring engine to demonstrate behavior preservation.

In summary, this dissertation closes a gap in the development of configurable systems by providing scalable techniques and specific tool implementations for analyzing and transforming large-scale, configurable software systems. Our techniques may serve as starting point for further research on analysis and transformation techniques for configurable systems. Furthermore, our tool implementations pave the way for a variety of research directions that can be applied for the first time in practical application scenarios. Besides research, our results may stimulate the development of reliable analysis and transformation tools for productive software development environments, such as ECLIPSE or VISUAL STUDIO, gaining confidence for both developers and users that all variants of a configurable systems are analyzed for errors. Overall, we align the current development practice of configurable systems with the development practice of single software systems, in general, and for systems developed with CPP in particular.

1.2 Outline

In Chapter 2 (*Background*), we lay the foundation for this thesis by introducing configurable software systems and their development with the C preprocessor CPP. For readers familiar with product-line engineering, we highlight differences and similarities with the development of configurable systems. Furthermore, we introduce central ideas of static analysis and refactoring of source code that are both central in discussions in the remaining chapters of this thesis.

In Chapter 3 (*Understanding Preprocessor Annotations*), we investigate how developers use the preprocessor for implementing configurable systems. To this end, we introduce a set of metrics to capture software-engineering questions with respect to program analysis and program transformation. We compute the metrics for a set of 42 open-source software systems in a first case study to get a comprehensive overview of `#ifdef` usage in practice. We reuse the 42 software systems in a second case study and discuss the effect of disciplined preprocessor annotations on the development of tool support. To this end, we analyze all systems regarding

preprocessor discipline and give recommendations for handling undisciplined preprocessor annotations.

In Chapter 4 (*Analyzing C Code with Preprocessor Annotations*), we review the common practice of analyzing configurable systems. We compare variability-aware and sampling-based type checking and liveness analysis for three medium- to large-scale software systems (BUSYBOX, LINUX, and OPENSLL). We demonstrate the extension of an existing data-flow framework for intra-procedural analysis to make it variability-aware. Based on this framework, we implement a set of static analyses, such as double free and uninitialized variables, and report our findings regarding error detection.

In Chapter 5 (*Refactoring C Code with Preprocessor Annotations*), we pick up on the analysis strategies developed in Chapter 4 for the implementation of an variability-aware refactoring engine that supports three common refactorings: `RENAME IDENTIFIER`, `EXTRACT FUNCTION`, and `INLINE FUNCTION`. We verify our refactoring engine rigorously using system tests.

In Chapter 6 (*Conclusion and Future Work*), we summarize our contributions and highlight further research directions.

2 Background

In this thesis, we propose improvements of the analysis and transformation of configurable systems. In this chapter, we introduce the motivation and main concepts for configurable systems as well as their development. In particular, we outline the development of configurable systems with the C preprocessor (CPP). In the remaining chapters, we frequently refer to existing work from the software-product-line community and draw conclusions for implementation techniques used in there. To ease understanding, we briefly outline similarities and differences of configurable systems and product lines in the following. Furthermore, we give a brief introduction into static analyses and refactoring, which we use as an example application of the analysis (cf. Chapter 4) and the transformation (cf. Chapter 5) of configurable systems.

2.1 Software Product Lines and Configurable Software Systems

The term software product line was coined in the mid-1990s to summarize a beneficial business strategy for companies developing software systems [Bass et al., 1997]. The Software Engineering Institute at Carnegie Mellon University defines a software product line as follows:¹

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

The main idea of SPLs is to center the development of related software systems around features, which represent optional or incremental units of functionality [Batory et al., 2004; Czarnecki and Eisenecker, 2000; Kang et al., 1990]. Based on features, single variants can be created that are tailored to the requirements of different application scenarios and use cases with which potential end users work. Using an SPL approach as a business strategy has three main advantages [Biggerstaff, 1998; Bass et al., 2003; Pohl et al., 2005]. First, reusing existing, well-tested feature implementations results in a higher quality of the system variants. Second, modifying an SPL to satisfy new requirements of a market is easier to realize and results in a decreased time to market. Third, since reimplementations from scratch are avoided, the overall costs of single variants are lower than those of individual implementations.

¹<http://www.sei.cmu.edu/productlines/>

Driven solely by economic aspects of a company, the original definition of a software product line does not include a clear process of its technical realization. In software engineering, two different approaches have been proposed and discussed in the context of SPL development [Apel, 2007]: stepwise refinement [Wirth, 1971] and program families [Parnas, 1976]. Both approaches suggest to use an incremental development strategy as their central design methodology for the development of software systems. Based on these approaches, various implementation techniques have been invented, such as component software [Szyperki, 2002], frameworks [Johnson and Foote, 1988], feature-oriented programming [Prehofer, 1997; Batory et al., 2004], aspect-oriented programming [Kiczales et al., 1997], and others. Rather than focusing on a single implementation technique, we take a step back and put the variability aspect of a software system into the center of our discussions. That is, we use the term configurable system as an abstraction of any software system that exhibits variability and discuss how variability affects the design and implementation of analysis and transformation techniques for these systems.

Nowadays, research on software product lines often adopts a technical perspective, e.g., by defining formalism, proposing implementation approaches, or comparing techniques and tools. This research often comes with its own terminology. To ease comparison, we contrast the terminology of configurable systems and software product lines in Table 2.1 and use the product-line terms as synonyms for configurable systems.

A configurable software system is a software system that provides a set of configuration options to adapt the system to end users' requirements. Given a set of configuration options, end users create, with the help of configuration knowledge that describes relationships of options to encode valid combinations of configuration options, a configuration. The functionality of configuration options is implemented in the form of a variable code base, which consists of a set of reusable implementation artifacts. Based on a valid configuration, a dedicated generator creates a system variant by applying the configuration to the variable code base. System configuration and the generator application involve several technical steps, including different formalism for configuration knowledge and different implementation approaches for variable code, which we introduce next.

Configurable system	Software product line
configuration option	feature
configuration	feature selection
configuration model	feature model or variability model
dependency	constraint
variant	product

Table 2.1: Terminology configurable systems vs software product lines.

2.1.1 Configuration Knowledge

A central ingredient of a configurable system is configuration knowledge. Configuration knowledge specifies which configuration options are available in a system and what their relationships to each other are. Together both span a system's configuration space, i.e., the set of all valid configurations that can possibly be derived. The analysis and transformation of a configurable system is governed by reasoning about configuration knowledge. For example, a programming error in a configurable system is determined by its position (file and line of code) in the variable code base and the specific system configuration that triggers the error. In the following, we describe different kinds and representations of configuration knowledge. Our description focuses on knowledge that is relevant for our subsequent case studies.

Configuration Options

Configurable systems usually provide a rich set of configuration options. Based on the selection/deselection of configuration options, the associated functionality of the option is activated or not, thus changing the behavior of a system variant. At the technical level, there are three points in time when configuration can take place:

Compile Time Configuration at compilation time influences which portions of source code (and functionality) of a configurable system are translated into an executable system variant. To this end, a compiler translates source code into object files containing machine code and subsequently assembles different object files into a program executable. This compilation process can be preceded by a preprocessing step, in which source code is transformed based on the values of configuration options. The developers of the LINUX kernel employ such a transformation using CPP. Configuration options of LINUX influence, for example, processor type, power management, and networking support.

Load time Load-time configuration is the preparation of a program at program start-up. This preparation can influence the availability of program functionalities by selecting or deselecting configuration options at program start. Command-line arguments are typical examples of load-time configuration options. For example, the GNU compiler collection (GCC) provides a set of configuration options that enable different optimization techniques during a program's compilation to improve the performance and the binary size of the resulting program.

Run time Configuration at run time is the most flexible way to influence a program's functionality. By selecting or deselecting configuration options during the program's execution, a configurable system can be adapted to a dynamically changing environment or to new user preferences. One example is NASA's Deep Space 1 software, which allows dynamic reconfiguration of a system by activating or deactivating system modules based on the spacecraft or mission status [Dvorak et al., 2000].

We concentrate on compile-time variability for two reasons. First, many programming errors (syntax or type errors, as well as semantic errors) are introduced during the coding phase of a system. Many of them can already be found using static analysis tools before the system is being compiled and executed. Detecting and repairing programming errors during coding is beneficial, since it catches errors of all variants, which can possibly be derived from a variable code base. Catching errors at load or run time of a program is limited to the analysis of functionality of configuration options that have been selected at compile time and, thus, have been included in the executable. The result of this analysis provides only a limited view of programming errors that may hide in a system's implementation, as only a single variant is analyzed for errors.

Second, configurable systems are often deployed to end users in the form of source code instead of an executable program. A typical example is the embedded database system `SQLITE`: `SQLITE` is a library-based implementation of a relational database management system, often deployed as an integral part of many programs, such as web browsers and operating systems. Given such a configurable system, variant configuration and variant generation typically happen on the user side, and it is not unlikely that an end user reveals a programming error in a single variant not previously tested by the developers of the configurable system. It is known that fixing an error reported by end users can be several orders of magnitude more costly than fixing it during the development of the system [Ko and Myers, 2003].

Configuration Options and Dependencies

Not all combinations of options of a configurable system necessarily make sense. Configuration options often depend on each other, e.g., one option implies another one, two options exclude each other, or three options have to be used together. By means of configuration knowledge, developers encode such dependencies and state which combinations of configuration options are valid.

The implementation of configuration options is often scattered across the variable code base of the system, involving different implementation artifacts such as models, source code, and build scripts. As a result, configuration knowledge is usually scattered, too. Similar to the previous discussion on compile-time, load-time, and run-time configuration, scattered configuration knowledge affects the configuration and derivation of system variants. A comprehensive overview of scattered configuration knowledge was recently given by Tartler [2013]. After studying several configurable systems, including the `LINUX` kernel, the author identified and classified six different levels (ranging from compile-time to run-time configuration) in the implementation of configurable systems that exhibit configuration knowledge. Figure 2.1 shows the three compile-time configuration levels, with excerpts of configuration knowledge for CPU hot-plugging in the `LINUX` kernel. The configuration of an option can span across different levels, where the configuration at one level affects the configuration of another. That is, selection/deselection of configuration options at the upper level l_0 dominates configuration options at lower levels (e.g., l_1 and so forth). Next, we describe the three levels (l_0 to l_2) in more detail by introducing languages and tools for configuration representation.

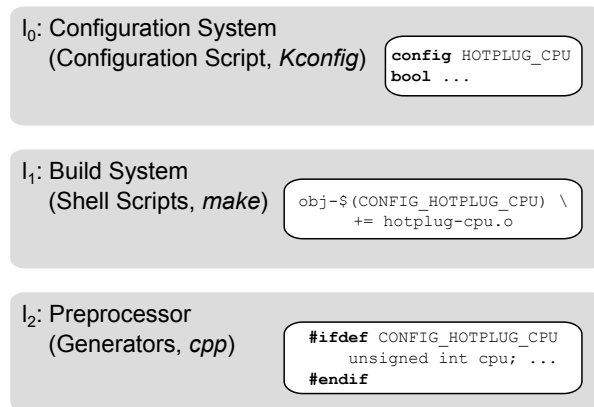


Figure 2.1: Compile-time configuration in LINUX; adapted from Tartler [2013].

Configuration System At the configuration-system level l_0 , developers use configuration options and their dependencies to specify a formal variability model of the configurable system. The resulting model is often employed during the configuration of a system variant, and its application during the configuration process ensures that only valid configurations are created. For the purpose of system configuration, the LINUX kernel developers invented the domain-specific language KCONFIG,² which enables developers to model variability. The following listing illustrates a typical definition of a configuration option in LINUX using KCONFIG:

```

1 config HOTPLUG_CPU
2 bool "Support for hot-pluggable CPUs"
3 depends on SMP && HOTPLUG && SYS_SUPPORTS_HOTPLUG_CPU
4 ---help---
5 Say Y here to allow turning CPUs off and on. CPUs can
6 be controlled through /sys/devices/system/cpu.
7 ( Note: power management support will enable this
8 option automatically on SMP systems. )

```

Figure 2.2: Definition of HOTPLUG_CPU with KCONFIG in LINUX [Tartler, 2013].

The definition includes a name (HOTPLUG_CPU), one of several predefined types (here boolean),³ the option's dependency (HOTPLUG_CPU is only selectable if SMP, HOTPLUG, and SYS_SUPPORTS_HOTPLUG_CPU are selected), and a help message. A configurator tool, such as the identically named configurator KCONFIG, reuses the

²<https://www.kernel.org/doc/Documentation/kbuild/>

³The KCONFIG language provides several types for configuration options. Beside boolean, tristate, string, hexadecimal, and integer are possible.

variability model to guide end users during the configuration of a kernel variant. To this end, the configurator checks the satisfiability of all dependencies and automatically infers selection/deselection of configuration options when an end user selects/deselects an option.

Build System At the build-system level (l_1), developers write a specification to automatically build executable programs and libraries from source files. The specification, most often written in `MAKEFILES` using the language of the `MAKE` utility, contains a set of rules describing build targets, their dependencies (including source files), and a sequence of commands (typically including a call to a compiler). Via configuration knowledge, developers explicitly state deviations within the regular build process by adding or removing build targets, or by changing their dependencies. To this end, developers integrate configuration options of the configuration system (Level l_0) within `MAKEFILES`. One example is illustrated in Figure 2.3, showing an excerpt of the `BUSYBOX` build system `KBUILD`. `KBUILD` is a framework centered around the `MAKE` utility and was invented by the `LINUX` developers.

```
1 lib-y:=
2 lib-$(CONFIG_fdisk) += fdisk.o
```

Figure 2.3: Examples of configuration-knowledge use in `MAKEFILES` in `BUSYBOX`.

`BUSYBOX` is a collection of standard `UNIX` tools combined in a single executable program, most often used in embedded systems. The main build target of `BUSYBOX` is specified with `lib-y`. The determination of configuration options during the configuration process (Level l_0) of `BUSYBOX` (y for inclusion and n for exclusion) advises `KBUILD` to include or exclude the functionality of a single tool in `BUSYBOX`'s main build target. According to the configuration of `CONFIG_fdisk`, the object file `fdisk.o` is assembled into the `BUSYBOX` program executable.

Preprocessor While at the previous level l_1 developers specify configuration knowledge at a coarse grain (i.e., they specify which files to include in a build process), they employ `CPP`'s configuration language to express knowledge within source files. By means of `#ifdef` directives and `#define` macros provided by `CPP`, developers can state complex configuration settings to encode a system's configuration knowledge. `#ifdef` directives, a static form of conditional statements that are evaluated before the compilation process, control the inclusion or exclusion of subsequent lines of source code. We take a closer look at `CPP` in Section 2.1.2. In Figure 2.4, we show an excerpt of `SQLITE`'s configuration knowledge for supported operating systems.

```
1 #if defined(SQLITE_OS_OTHER)
2 # if SQLITE_OS_OTHER==1
3 #   undef SQLITE_OS_UNIX
4 #   define SQLITE_OS_UNIX 0
5 #   undef SQLITE_OS_WIN
6 #   define SQLITE_OS_WIN 0
7 # else
8 #   undef SQLITE_OS_OTHER
9 # endif
10#endif
```

Figure 2.4: Excerpt of the operating-system configuration with CPP in SQLITE.

The preprocessor macros ensure that the configuration options `SQLITE_OS_OTHER`, `SQLITE_OS_UNIX`, and `SQLITE_OS_WIN` form an alternative group; that is, exactly one of the three options can be selected at a time. The definition (`#define`) and recall (`#undef`) set values for configuration options and so influence SQLITE’s operating-system support.

Scattered configuration knowledge and its representation using different formalism pose a huge problem [Berger, 2013; Tartler, 2013; Pearse and Oman, 1997; Dietrich et al., 2012b; Elsnor et al., 2011; Nadi, 2013]. The major challenge is that configuration knowledge represented in specialized languages, such as KCONFIG, MAKE, and CPP, is often only accessible by dedicated tools. As a result, represented knowledge cannot be used in tasks different from their original design and specification such as consistency checks or variability-aware analysis and transformation. Accessing of and reasoning about configuration knowledge in a uniform and systematic way requires a transformation of the different formalism into a canonical representation. For the three formalism mentioned a canonical representation is propositional logic [Berger, 2013; Tartler, 2013; Batory, 2005]. Using propositional logic, configuration knowledge is represented with propositional formulae: variables represent configuration options and logical operators encode dependencies between different options. A propositional formula (also known as *presence condition*) represents a set of valid configurations. Although the configuration languages often lack a clear semantics, several researchers proposed transformation tools to extract presence conditions from KCONFIG [Berger, 2013; Tartler, 2013; Zengler and Küchlin, 2010; Nadi, 2013], MAKE and KBUILD [Dietrich et al., 2012a], and CPP [Sincero et al., 2010]. Figure 2.5 shows the corresponding propositional formulae for the just mentioned examples of KCONFIG, KBUILD, and CPP.

We can reason about propositional formulae by means of automated reasoning tools [Mendonça et al., 2009; Benavides et al., 2005, 2010] such as Boolean Satisfiability Problem (SAT) solvers, Constraint Satisfaction Problem (CSP) solvers, or Binary Decision Diagrams (BDDs). This way, different questions related to the analysis and transformation of configurable systems can be answered (adapted from [Benavides, 2007; Mendonça, 2009]):

- Is a given configuration option dead, i.e., is it never selectable?

2 Background

$$\begin{array}{ll} \text{HOTPLUG_CPU} \implies & \text{CONFIG_fdisk} \\ \text{SMP} \wedge \text{HOTPLUG} \wedge \text{SYS_SUPPORTS_HOTPLUG_CPU} & \end{array}$$

(a) KCONFIG; cf. Figure 2.2

(b) MAKE; cf. Figure 2.3

$$\begin{array}{l} (\text{SQLITE_OS_OTHER} \wedge \neg \text{SQLITE_OS_UNIX} \wedge \neg \text{SQLITE_OS_WIN}) \\ \vee (\neg \text{SQLITE_OS_OTHER} \wedge \text{SQLITE_OS_UNIX} \wedge \neg \text{SQLITE_OS_WIN}) \\ \vee (\neg \text{SQLITE_OS_OTHER} \wedge \neg \text{SQLITE_OS_UNIX} \wedge \text{SQLITE_OS_WIN}) \end{array}$$

(c) CPP; cf. Figure 2.4

Figure 2.5: Propositional formulae for the configuration knowledge presented in Figures 2.2, 2.3, and 2.4.

- Is a given presence condition, representing a set of configurations, satisfiable? Is it a tautology? Is it a contradiction?
- How many valid solutions does a given presence condition have?
- Are two given presence conditions equivalent?

The interpretation of these questions, encoded as propositional formulae, requires a boolean-satisfiability check, which is known to be in NP-complete [Mendonça, 2009]. However, existing solvers can reason about questions that arise in the context of configurable systems efficiently, even for systems with thousands of configuration options [Thüm et al., 2009; Mendonça, 2009; Mendonça et al., 2008; Apel et al., 2010b].

In practical systems, propositional logic is often not sufficient to encode a system’s configuration knowledge [Czarnecki et al., 2012; Berger et al., 2010b; She et al., 2010]. For example, the developers of LINUX make regular use of configuration options with non-boolean types, such as integer, hexadecimal, and string; around 3.7% of LINUX’ configuration options use these types [Berger et al., 2010b]. Together with arithmetic, relational, and string operators, these options form configuration knowledge in first-order logic. Figure 2.6 shows one example of a non-boolean constraint taken from the C standard library GLIBC. Beside propositional logic, the constraint contains two configuration options (`DBL_MANT_DIG` and `LDBL_MANT_DIG`) that take values from the integer domain and that are combined using arithmetic (+) and relational (`>=` and `!=`) operators. Satisfiability of first-order logic is in general undecidable (i.e., there is no algorithm to determine satisfiability). However, for practical examples of configuration knowledge with non-boolean constraints, algorithms exist that translate such representations into propositional logic [Thüm, 2008; Tartler, 2013]. For our purposes of analyzing and transforming configurable systems we use propositional logic that has been extracted and translated from the original representations [Berger et al., 2010b; Tartler et al., 2011; Berger et al., 2010a].

The different sources of configuration knowledge that we reviewed in this section are only a subset of available representations. Alternative representations such as grammars [Batory, 2005] or graphical representations such as feature diagrams [Czarnecki and Wąsowski, 2007] could be used for our purposes, too, as long as they can be transformed into an equivalent

```
1 #if !defined(NO_LONG_DOUBLE)
2   && (LDBL_MANT_DIG >= DBL_MANT_DIG + 4)
3   && (LDBL_MANT_DIG != 106)
```

Figure 2.6: Example of configuration knowledge in GLIBC using non-boolean constraints.

representation using propositional logic.

2.1.2 Implementing Configurable Systems

There is a variety of tools for the implementation of configurable software systems. Among all approaches, two different techniques have emerged that represent two contrary paradigms: annotative approaches and compositional approaches. While in annotative approaches common and variable code is combined in a single code base, compositional approaches separate common and variable code into code units. We give only a brief overview of the two approaches here. For more information, especially regarding the individual strengths and weaknesses, we refer the interested reader to elsewhere (e.g., [Kästner and Apel, 2009; Apel and Kästner, 2009; Apel, 2010, 2007]). To illustrate the use of both approaches for the implementation we use the example of a simplified list data structure in C. The list consists of a mandatory implementation of a singly-linked list; it can be configured by means of several configuration options to support doubly-linked functionality (`DLINKED`) and different sorting algorithms (`BUBBLESORT` and `INSERTIONSORT`).

Annotative Approaches

The key idea of annotative approaches is to annotate source-code fragments with presence conditions in order to make code configurable. A presence condition is a propositional formula about configuration options, representing a slice of configuration knowledge, in the implementation artifacts of a system. Annotated code fragments reside next to non-annotated code in a single code base. Individual variants of the configurable system can be generated by applying a generator (also known as preprocessor) to the code base that evaluates presence conditions of annotated code fragments and, based on the evaluation result, includes or excludes the corresponding fragments. A widely used preprocessor is CPP.

The preprocessor CPP is a stand-alone tool for text processing, which enhances C by lightweight meta-programming capabilities [Kernighan and Ritchie, 1988]. Originally considered solely as an optional adjunct for C [Ritchie, 1993], the preprocessor language lacks a complete integration into the syntax of C; a fact that has persisted until now. Over the years, developers have adopted CPP, and it has been commonly used for quite a while [Spencer and Collyer, 1992; Favre, 1996, 1997]. The lack of integration into C is the reason why the preprocessor is not limited to a specific language. Rather, it can be used for arbitrary text and source-code transformations also in other languages [Ritchie, 1993].

CPP works on the basis of preprocessor directives (indicated by `#` at the start of line) that are used by developers to define explicit source-to-source transformations. Using these directives, developers can overcome limitations of the programming language C in terms of abstraction [Favre, 1995], modularity [Badros and Notkin, 2000], and portability [Spencer and Collyer, 1992]. For example, arbitrary sequences of tokens can be named and referenced in the source code (abstraction), code can be outsourced into different files (modularity), and different hardware requirements can be met by use of conditional inclusion (portability).

In Figure 2.7, we show the implementation of our variable list in C using the C preprocessor. Source code that is common to all variants of the list implementation is expressed using the capabilities of C in terms of data abstraction (data types; e.g., `struct node` in Lines 1 to 7) and procedural abstraction (functions; e.g., `init` in Lines 9 to 17). The implementation of configurable source code rests on conditional inclusion provided by CPP. An optional code fragment (*variable code*) is enclosed in `#ifdef X` and `#endif` directives, and can be configured according to the selection of `X`.

Configuration options can be defined using CPP’s `#define` macro (for textual substitutions) in the source code directly, or externally in MAKEFILES, configuration files, configuration tools, or in the form of compiler parameters. Developers can express complex configuration settings by combining configuration options using different operators of the CPP language. The language provides operators for logical reasoning (e.g., `&&`), bit manipulations (e.g., `&`), common mathematical operators such as plus for addition, or operators for comparisons such as less than. Single configuration options as well as combined expressions represent presence conditions that control the inclusion or exclusion of subsequent lines of source code up to the next conditional inclusion directive, i.e., one of `#if`, `#ifdef`, `#ifndef`, `#else`, or `#endif`. For example, Line 5 (the pointer to the previous list node) is included if configuration option `DLINKED` is selected. We call the use of `#ifdef` also *source-code annotation*.

Technically, every source-code fragment that is enclosed by an `#ifdef` is configurable. Annotated code fragments can belong to an alternative, which represents complementary implementations in the configurable system. To this end, developers either use a conditional-

```

1 struct node {
2     int item;
3     struct node *next;
4 #if DLINKED
5     struct node *prev;
6 #endif
7 };
8
9 struct node *init(int newitem) {
10     struct node *n = (struct node *)
11         malloc(sizeof(struct node));
12     n->item = newitem;
13     n->next = NULL;
14 #if DLINKED
15     n->prev = NULL;
16 #endif
17 }
18
19 struct node *first = NULL;
20 #if DLINKED
21 struct node *last = NULL;
22 #endif
23
24 void insert(struct node *elem) {
25 #if BUBBLESORT
26     /* sorting algorithm 1 */
27 #elif INSERTIONSORT
28     /* sorting algorithm 2 */
29 #else
30     /* unsorted insertion */
31 #endif
32 }

```

Figure 2.7: A variable list implementation with CPP; taken from Liebig et al. [2010].

inclusion cascade `#if-#elif-#else` (Lines 25 to 31) or they separate the implementation into multiple `#ifdef` directives with mutually exclusive presence conditions. Alternative implementations are, with respect to code analysis, particularly challenging, as complementary implementations cannot be part of a single system variant and, hence, cannot be analyzed together. Thus, they increase the effort of analyzing configurable systems. We deal with this problem in Chapter 4 (*Analyzing C Code with Preprocessor Annotations*) in detail.

The implementation of a single presence condition is often scattered across the variable code base of a configurable system. For example, the implementation of `DLINKED` in Figure 2.7 has three code fragments in Lines 5, 15, and 21. The scattered implementation tangles with the mandatory base implementation of the list and with the implementation of other configuration options. The implementations of `BUBBLESORT` and `INSERTIONSORT` tangle in the list implementation. Another form of tangling occurs when `#ifdef` directives are nested. The directive inside tangles with the enclosing one and only gets evaluated if the presence condition of the enclosed directive is satisfiable. This special form is worth noting as it is a major source of complexity when implementing variability with `#ifdef` directives.

Compositional Approaches

In contrast to annotative approaches, compositional approaches separate variable code into code units. Instead of scattered implementations of configuration options, as with `#ifdefs`, implementations of configuration options are „modularized“ in single code artifacts such as aspects [Kiczales et al., 1997], feature modules [Batory et al., 2004], and plugins [Johnson and Foote, 1988]. Each approach provides a set of abstractions for extending an existing code base. For example, frameworks provide a common platform in which configurable code is modularized in the form of plugins. Plugin implementations make use of predefined extension points of the platform for the realization of their functionality. When executing the framework, plugins, which can statically be loaded at compilation time, are executed when the control flow hits an extension point. While frameworks are implemented in a language itself, often using specific design patterns [Gamma et al., 1995], other approaches often define a new language or extend an existing one to implement configurable systems. Two approaches, which are frequently discussed in the context of configurable systems, are aspects [Kiczales et al., 1997] and feature modules [Batory et al., 2004; Apel et al., 2009].

Figure 2.8 shows an excerpt of the implementation of our variable list using aspects and feature modules. The common base implementation of a singly-linked list is extensible by a separated implementation of the configuration option `DLINKED`. In Figure 2.8a, we use ASPECT-ORIENTED C (ACC) [Gong and Jacobsen, 2008], an implementation of aspect-oriented programming for C. ACC provides a central abstraction, called *aspect*, for the extension of a program at well-defined points of the program execution context (*join point*) with additional program functionality (*advice*) and the static introduction of new program elements (*intertype declaration*). In our example, the new member-variable `prev`, holding a reference to the previous list node element, is introduced via an intertype declaration (Lines 9 to 11). The initialization code for the new member is implemented by the advice code in Lines 16 to 18.

2 Background

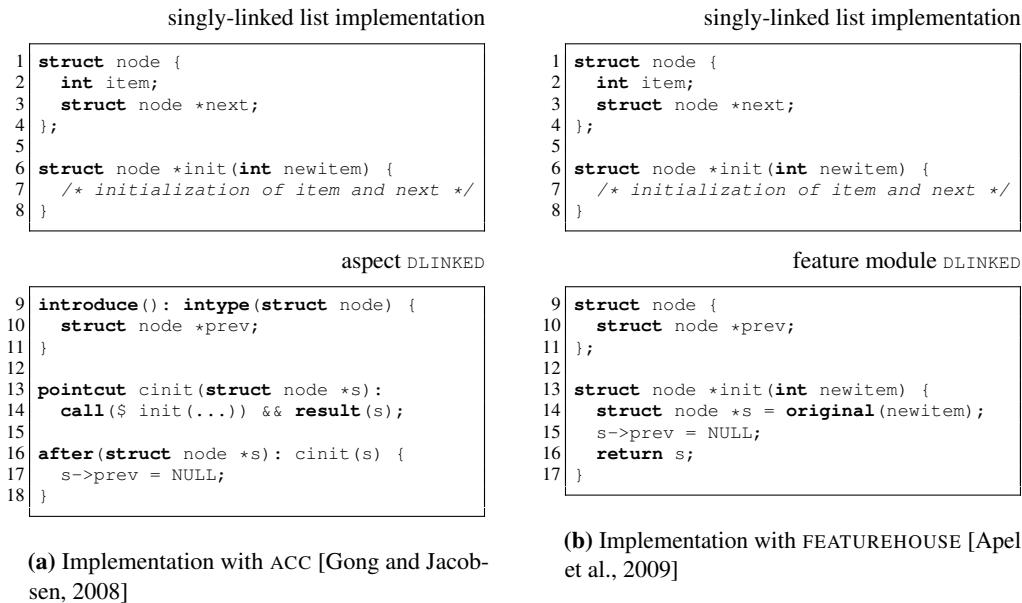


Figure 2.8: Implementation of the configuration option `DLINKED` using aspect-oriented and feature-oriented programming.

The code is placed into the base implementation using the pointcut `cinit` that captures each call to the list-node initialization function `init` (Lines 13 to 14).

Similarly, Figure 2.8b shows the basic list implementation and the configurable code for `DLINKED` with feature modules using `FEATUREHOUSE` [Apel et al., 2009]. A feature module encapsulates type and function introductions as well as type and function refinements for an existing base implementation. To derive a specific system variant, a selected set of feature modules is composed using *superimposition* [Apel and Lengauer, 2008]. Superimposition merges different software artifacts by combining corresponding substructures of the input feature modules. In our example, superimposition adds the new member `prev` to the existing structure definition of the list node in each variant including the `DLINKED` option. Furthermore, superimposition controls the refinement of functions using function wrappers. The function definition (Lines 13 to 17) wraps the original function definition from Lines 6 to 8 by storing its result and executing additional initialization code.

2.2 Static Analysis

Static analysis is the computation of properties of source code for the purpose of automatic code analysis [Nielsen et al., 1999; Khedker et al., 2009]. The analysis is performed without

program execution, usually at code-development time with dedicated tools, often integrated within modern development environments, such as ECLIPSE or VISUAL STUDIO. Alternatively, some static analyses are performed by the compiler as part of the compilation process. A main goal of static analysis is to find programming errors early in the development process of a software system. Even though an executable program is often not available at the early stages of the development process, static analysis can still help to find errors in preliminary versions of a program. Examples of such analyses are syntax and type checks or more sophisticated checks. The latter deal with questions whether dynamically allocated memory is freed only once or whether each variable is initialized with a proper value before its first use. In contrast to dynamic program analysis, program parameters are not set and static analysis can only reveal static aspects of a program behavior, i.e., properties that all program executions share. As a result, analysis findings can be regarded as errors, although they are not. Nevertheless, static analysis is frequently applied in practice [Zheng et al., 2006].

Static analysis usually does not operate on the source code of a program directly, but uses its abstract representations that capture essential information of a program to compute a desired program property. A typical representation of source code is the Abstract Syntax Tree (AST). This representation is a result of a syntax analysis of the input program and abstracts from specific information such as code layout, punctuation, and comments. Figure 2.9b shows the AST representation of the code example in Figure 2.9a. ASTs serve as an input for the creation of additional representations such as Control-flow Graphs (CFGs). A CFG is a graph representation of all potential execution paths during program execution (successor relation). As run-time information is not available, the graph is only a (conservative) approximation of the actual program behavior, i.e., some paths may never be executed. Figure 2.9c shows the CFG for our example. The nodes refer to line numbers in the input program and the edges encode the successor relation of program statements that are possibly executed.

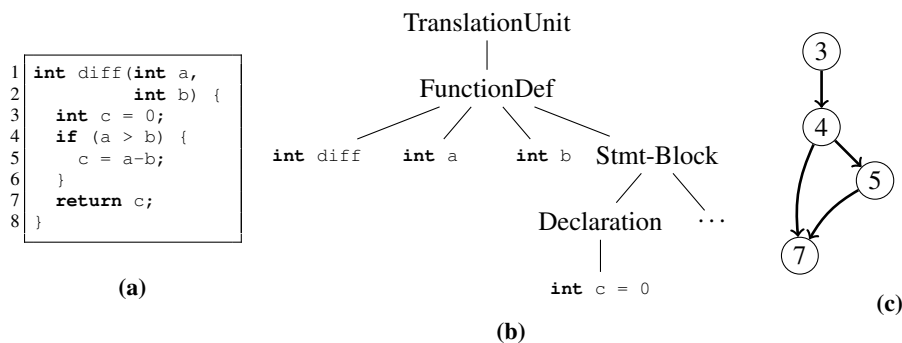


Figure 2.9: Code example with its corresponding AST (excerpt) and its CFG representation.

Abstract representations form the basis of an automatic analysis employed by a static analysis tool. State-of-the-art static analysis tools compute a large set of program properties

that play an important role in different parts of software development. The properties computed are used, for example, in debugging [Ball and Rajamani, 2002], maintenance [Cole et al., 2006], verification [Lev-Ami et al., 2000], testing [Nagappan and Ball, 2005], and program transformation [Morgenthaler, 1997]. Next, we illustrate two cases of static analysis: type-checking and data-flow analysis, which we use to discuss scalable analysis techniques for configurable systems in Chapter 4.

Type-Checking

Type-checking is the process of checking well-typedness of a program's source code [Pierce, 2002]. A type is a classification of possible values that can safely be handled by a program element such as variables, expressions, or functions. Using a program's AST representation and based on a set of typing rules, a type checker assigns types to program elements (AST nodes) and automatically checks that types of different program elements are compatible. In this way, a type checker

proves that an input program is either type correct (the program is well-typed) or that the program contains type errors (the program is ill-typed). Similar to all static analyses type-checking is only a conservative approximation of the actual program behavior. Thus, a type checker may reveal a type error at a program element that can never occur during the execution of the program. Existing type checkers for C are able to detect a variety of type errors including incompatible type casts (e.g., a variable holding a sequence of characters is converted to type integer; cf. Figure 2.10, Line 2), dangling references (e.g., a function call cannot be resolved to its corresponding function definition; cf. Figure 2.10, Line 3), and duplicate variable or function declaration (e.g., the function definition in Line 5 is in conflict with the definition in Line 1 as function overloading is not allowed in C).

```
1 int foo(char* s) {  
2     int i = (int)s;  
3     return i*bar();  
4 }  
5 int foo(int x) { ... }
```

Figure 2.10: Examples of type errors in C.

Data-flow Analysis

Data-flow analysis is a lightweight technique for gathering program properties of the dynamic behavior of a program by analyzing its source code. Without actually executing a program's source code, the properties statically gathered can be used to detect programming errors, beyond type errors. A common example is the detection of uninitialized variables, i.e., variables that are declared but not assigned a proper value before they are used, which is a frequent programming error in practice. Data-flow analysis works on a program's CFG representation, which represents all potential execution paths of a program. For each single element of the CFG, the analysis computes the desired property using a set of data-flow equations. The equations include functions to simulate actual program behavior (transfer functions) and a function for joining intermediate results from different execution paths. Due to dependencies between data-flow properties of CFG elements, i.e., the result of one computation is part of the input of another, the computation of data-flow analysis typically employs an iterative algorithm. The

algorithm repeatedly computes data-flow properties for all elements of the input CFG and passes computed properties to depending computations. After completing several steps of the iterative computation, the computed properties of all CFG elements eventually stabilize by reaching a fix-point and, thus, terminating the computation.

Another example of data-flow analysis, which we discuss in more detail later (cf. Section 4.1.2), is the computation of live variables, which is called liveness analysis. Liveness analysis is a classical data-flow analysis that computes a set of variables whose values may be needed during the subsequent program execution, i.e., variables that are read before their next update. The resulting set can be used to detect dead code conservatively. A variable assignment that is not used, i.e., whose value is not read in the future, is probably dead. The iterative computation of live variables of condition in Line 4 (cf. Figure 2.9a) results in the following set: $\{a, b, c\}$.

2.3 Refactoring

Refactoring is the process of changing the source code of a program while preserving the semantics and, therefore, the overall behavior of the program [Fowler, 1999]. There are various reasons for refactoring source code. For example, refactoring can be used to fix poorly designed or gradually impaired code, to clean up code hackings, or to prepare the introduction of new features [Fowler, 1999; Mens and Tourwé, 2004]. Furthermore, refactorings may be used to restructure code to support a different programming language or language paradigm [Liu et al., 2006]. Overall, refactoring accompanies software evolution to optimize non-functional properties of a software system such as extensibility, modularity, reusability, complexity, maintainability, and efficiency [Mens and Tourwé, 2004].

To apply a refactoring, developers indicate code transformations often in the form of informal descriptions. Typical examples of such informal descriptions are: change the identifier of a function, type, or variable from `old` to `new` (RENAMING IDENTIFIER refactoring), extract the selection of statements into a new function named `f○○` (EXTRACT FUNCTION refactoring), or inline the function definition of `f○○` for all function calls in a program (INLINE FUNCTION refactoring). Based on such descriptions, refactoring engines implement refactorings as (semi-)automatic code transformations. To do so, refactoring algorithms typically use results of static analysis. By unburdening developers from manual code transformations that are tedious and error-prone, the application of refactoring engines result in a significant increase in developer productivity. Nowadays, refactoring engines are available for most mainstream languages, and they are usually integrated in modern development environments such as ECLIPSE,⁴ VISUAL STUDIO,⁵ and XCODE.⁶ Existing refactoring engines usually provide a small but well-documented set of refactorings [Fowler, 1999]. Table 2.2 lists a set of common refactorings for C and CPP.

⁴<http://eclipse.org>

⁵<http://microsoft.com/visualstudio/>

⁶<http://developer.apple.com/xcode/>

	Refactoring	Description
C	DELETE UNREFERENCED VARIABLE	Determine and remove unused or unnecessary variables
	RENAMING IDENTIFIER	Rename identifiers of variables, function names, and user-defined type names consistently
	EXTRACT FUNCTION	Extract a selected code fragment of a function, put it into a new function definition, and insert a function call at the place of extraction
	INLINE FUNCTION	Substitute function calls with the code of their corresponding function definition
CPP	RENAMING MACRO	Rename #define macro definition and all its usages
	REMOVE CONDITION	Remove #ifdef annotated code, based on a given configuration

Table 2.2: Common refactorings in C and CPP code.

To ensure that typically informally specified refactorings do not change the behavior of a program, one has to verify that they are semantics-preserving. A manual verification is usually infeasible as existing mainstream languages and software systems are often too complex. Consequently, (semi-)automatic approaches for checking the correctness of a refactoring are needed. Using formal verification, the informal descriptions of refactorings are refined to formal specifications, enabling (semi-)automatic verification using rewriting logic (e.g., MAUDE SYSTEM)⁷ or proof assistants such as COQ⁸ or ISABELLE.⁹ However, formal verification is a difficult task, because it requires a formal description of a programming language's semantics [Schäfer and de Moor, 2010]. Creating such a description is usually a very difficult task on its own, and, therefore, formal descriptions are typically not available for mainstream programming languages [Schäfer, 2010]. As a result, formal verification is only used for simplified languages [Sultana and Thompson, 2008].

Developers of refactoring engines usually apply precondition satisfaction to ensure behavior preservation [Obdyke, 1992]. A precondition encodes portions of a language's semantics and represents a global condition that must be satisfied before an engine applies a refactoring. For example, the renaming refactoring in C has two preconditions for new identifiers: first, a new identifier must start with an underscore or a letter followed by the same ones or a digit. Second, the new identifier must be unique, i.e., it must not clash with existing identifiers in

⁷<http://maude.cs.uiuc.edu/>

⁸<http://coq.inria.fr/>

⁹<http://www.cl.cam.ac.uk/research/hvg/Isabelle/>

the same scope. If a refactoring's preconditions are satisfied, a number of predefined code transformations are applied. However, the use of preconditions is often problematic as a refactoring's preconditions are often incomplete, resulting in erroneous transformations in corner cases of the refactoring. Two approaches that have been applied successfully to deal with the precondition problem are constrained-based refactoring and testing [Mens and Tourwé, 2004].

Constraint-based refactoring uses a weaker notion of behavior preservation that does not require complete formal descriptions [Mens and Tourwé, 2004]. A formalism based on constraints helps to capture solely required properties of a programming language for checking the satisfiability of a refactoring's preconditions [Tip, 2007]. The precondition check is transformed into a constraint-solution problem that can be tracked with existing solvers. Proposed engines usually target only specific refactorings such as refactoring of types and class hierarchies with the help of type constraints [Tip, 2007], or refactorings for moving classes and pulling-up methods with the help of access-modifier constraints [Steimann and Thies, 2009]. Although a full formal specification of a refactoring is not necessary for this approach, the definition of constraint rules is not trivial and usually requires a deep understanding of a programming language's semantics.

The most common approach to prove the correctness of a refactoring engine—if at all—is testing [Mens and Tourwé, 2004]. In contrast to formal verification, testing directly validates a refactoring-engine's implementation. Testing is a simple but effective approach, which can fully be automated. The standard procedure of testing refactoring engines is to check the refactoring input against a set of trustworthy tests, before and after a refactoring is applied. Differences in the outcomes indicate, for example, a missing precondition check or an erroneous refactoring of the input. Existing approaches for testing refactoring engines use either existing software systems [Gligoric et al., 2013] or randomly generated programs [Soares et al., 2013] as input. While existing software systems usually come with a dedicated test suite that can be employed during testing, randomly generated programs often compute values that could be checked in a test. A refactoring-engine test does not cover the entire language definition or the full behavior of a program. So it typically captures only some aspects of program behavior, e.g., name binding, accessibility, types, control flow, and data flow [Soares et al., 2013]. Although testing allows only to check the existence but not the absence of errors, researchers found a large number of errors in existing, even commercially used refactoring engines of modern development environments [Gligoric et al., 2013].

3 Understanding Preprocessor Annotations

This chapter shares material with the ICSE'10 paper „An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines“ [Liebig et al., 2010] and the AOSD'11 paper „Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code“ [Liebig et al., 2011].

Before we get to the analysis and transformation of configurable systems, we need to understand how developers use variability mechanisms to implement configurable systems. This is because the implementation of a system's variable code base directly influences the development of proper tool support. Although CPP is a simple tool, invented more than 40 years ago, its use in large-scale configurable systems has been largely unexplored, and so even basic information, such as the number of available configuration options in a system, is often not available.

Based on prior work on the preprocessor by other researchers, we analyze the use of preprocessor annotations in software systems with respect to three criteria. First, we investigate the implementation of *configuration knowledge* with `#ifdef` directives in the source code. As a result, we can estimate the effort of an analysis or transformation and are able to give recommendations for reasoning about configuration knowledge in configurable systems. Second, we investigate the use of `#ifdef` directives for the implementation of a configurable system's *variable code base*. The developers' use of `#ifdef` directives determines the design and implementation of tool support, as variability has to be represented and processed in analysis and transformation approaches. Understanding the current practice of variability implementation helps to derive requirements for analysis and transformation approaches. Third, tool support requires a uniform representation of `#ifdef` directives and source code for analysis or transformation (*variability representation*). Since preprocessors can be used for arbitrary annotations of source code, we investigate to what extent `#ifdef` directives already conform to structured variability representations.

To obtain a comprehensive overview of `#ifdef`-directive usage in practice, we analyze a set of 42 open-source software projects from different domains and of different sizes, which are all written in C. The analyzed systems cover a wide range of application domains, including web servers, operating systems, application software, and systems with several thousands up to millions of lines of source code.

3.1 Methodology

For our study, we use the Goal-Question-Metric (GQM) approach [Basili et al., 1994], which helps to systematically identify and measure relevant factors for a qualitative analysis of a predefined objective. To this end, GQM defines three levels: conceptual, operational, and quantitative. At the conceptual level, the purpose of measurement is defined in terms of goals. A *goal* represents a specific measurement object, often related to a product, process, or resource. At the operational level, each goal is refined using a set of *questions* that characterize models and qualitative assessments. For the quantitative analysis (quantitative level), a set of *metrics* is defined and associated with the previously stated questions. The GQM approach does not enforce a strict one-to-one mapping between questions and metrics, so a question can be associated with multiple metrics and vice versa.

Next, we describe the GQM model for our study, which is summarized in Figure 3.1. To this end, we state a set of goals, questions, and corresponding metrics, which are inspired by previous case studies on CPP, and we discuss why they are important.

3.1.1 Goals

In our context, the measurement object is the source code of the configurable systems in question, and our main goal is the determination of the influence of `#ifdef` usage on program analysis and program transformation (G1). We divide this goal into three sub-goals, which reflect three different aspects of CPP usage.

Configuration knowledge (G1.1)

The implementation of configuration knowledge with `#ifdef` directives is likely to affect analysis and transformation approaches for configurable systems. For example, the number of configuration options affects the number of valid configurations that can possibly be derived from a system. The higher the number, the lower the chances to successfully apply an analysis or transformation approach without built-in support for variability. Analyzing and testing all variants for errors is known to be time-consuming, and experience shows that even though some code may work in one variant, it may fail to compile and run in another [Pearse and Oman, 1997; Spencer and Collyer, 1992].

According to practitioners and researchers, developers already benefit from simple quantitative and qualitative access to configuration knowledge [Pearse and Oman, 1997; Favre, 1996, 1997; Spencer and Collyer, 1992]. Simple information, such as the number and distribution of configuration options, helps developers to understand conditional compilation [Pearse and Oman, 1997]. In the past, different researchers developed basic tools for preprocessor analysis to answer specific questions regarding two lines of research: maintenance effort and implementation complexity [Sutton and Maletic, 2007; Ernst et al., 2002; Pearse and Oman, 1997; Krone and Snelting, 1994]. With G1.1, we pick up on both lines and examine the use of `#ifdef` directives for the implementation of configuration knowledge in a system. The goal represents

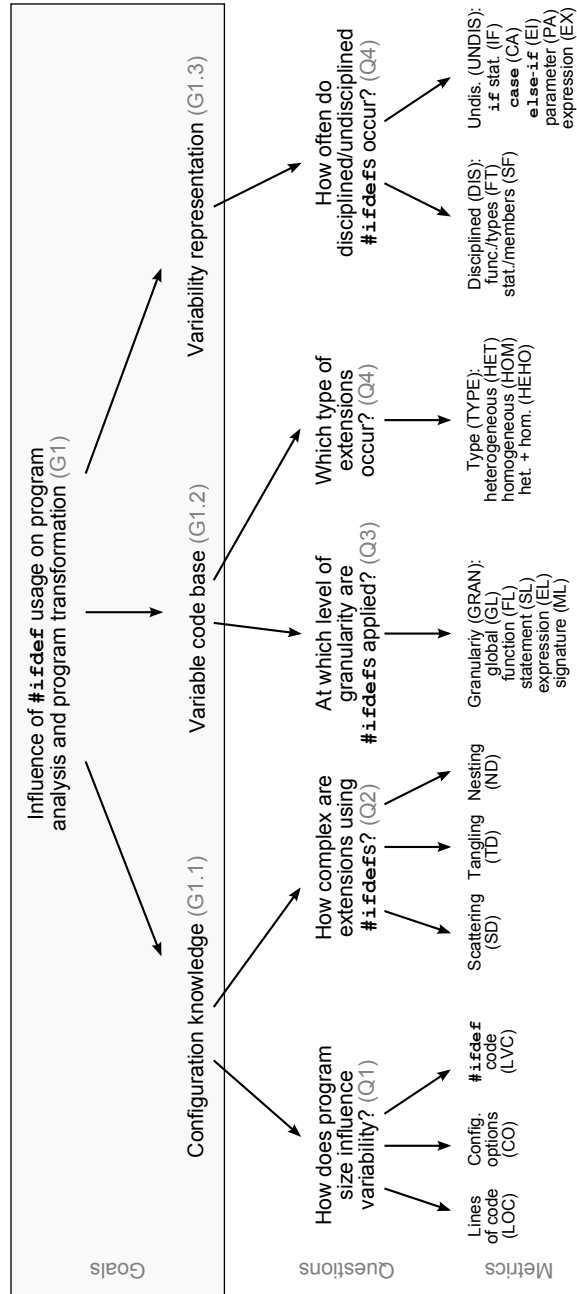


Figure 3.1: GQM model of our study on preprocessor usage.

analysis effort and implementation complexity that `#ifdef` directives cause in the source code.

Variable code base (G1.2)

The use of `#ifdef` directives dictates to what extent variability-aware tools should be able to handle variability. In particular, developers can use directives at different levels of granularity (ranging from single tokens to annotating the entire content of a file). These different levels have to be respected by tool writers. For example, the variability-aware refactoring engine CREFACTORY supports annotations on statements, declarations, structure members, enumeration values, and array-initializer values [Garrido and Johnson, 2005]. Annotations on other elements have to be transformed into these five supported annotations (in a preprocessing step) before refactorings can be applied.

CPP usage is at the center of a larger discussion on variability implementation regarding expressiveness, comprehension, and replication (cf. Figure 3.2) [Schulze et al., 2011, 2013a; Liebig et al., 2010]. Expressiveness denotes the ability and flexibility to make changes to the source code. CPP's capability to express variability at a fine grain enables developers to annotate arbitrary sequences of tokens. As a result, the preprocessor has a high expressiveness with respect to variability implementation and, thus, can help to avoid code replication. By factoring out common source code, developers may annotate only differences in the implementation of different variants by annotating specific code fragments. Consequently, developers can reduce the amount of replicated code in a system significantly. However, an expressive preprocessor comes at a cost: fine-granular extensions often disrupt the source-code layout, making the code difficult to understand for developers [Favre, 1996, 1997; Pearse and Oman, 1997; Baxter and Mehlich, 2001]. Limiting the expressiveness of the preprocessor to a subset of all annotations (disciplined annotations) requires a certain amount of code replication. Although some researchers consider code replication harmful (e.g., [Jürgens et al., 2009; Roy and Cordy, 2007]), we and others [Kapsner and Godfrey, 2008] argue that limited amounts of replicated code are manageable. This is because the alternatives containing the replicated code are side-by-side and, therefore, easy to track for the programmer.

With respect to analysis and maintenance tasks, researchers frequently report from the struggle developers have with scattered preprocessor implementations [Spencer and Collyer, 1992; Pearse and Oman, 1997]. Even simple tasks, such as following the control flow in a specific code fragment, become tedious in the presence of preprocessor directives, as developers have to go over the source code multiple times to understand it [Pearse and Oman, 1997]. Some researchers have summarized their frustration and struggle with verdicts such as “`#ifdef`

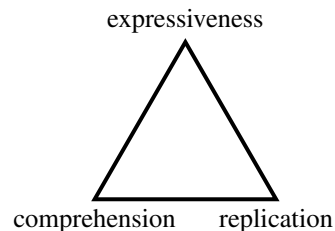


Figure 3.2: Trade-off between expressiveness, comprehension, and replication [Schulze et al., 2013a].

considered harmful” [Spencer and Collyer, 1992] or “**#ifdef** hell” [Lohmann et al., 2006].

With G1.2, we investigate important facets of variability implementation that influence the design and development of variability-aware tool support.

Variability representation (G1.3)

The development of analysis and transformation tools relies on an integrated analysis of the source-code structure and the effect of preprocessor directives on variable source code. Since the preprocessor is oblivious to the underlying source-code structure, developers can use it to annotate arbitrary tokens of the source code, such as a single closing brace, thus ignoring existing naming conventions and destroying the code structure. As a result, parsing and analyzing the unprocessed representation of the source code (pre-CPP) is difficult. Although this is not a problem for tools that analyze a single preprocessed variant of the source code (post-CPP), such as a compiler or many static analysis tools, pre-CPP tools have difficulties handling arbitrary text-based preprocessor annotations. The major problem is that arbitrary annotations do not represent structured input, on which these tools usually operate. A typical approach that has been used to analyze pre-CPP code (sometimes explicitly, but mostly implicitly) is to handle not all, but only a subset of annotations, which we call *disciplined annotations*. Disciplined annotations are annotations on certain syntactic code structures, such as entire functions and statements (we give a definition later in this section). Analogously, we call annotations of individual tokens or braces that do not align with underlying code structure *undisciplined annotations*. Restricting developers to disciplined annotations makes it much easier to build proper tools and to ensure correctness and completeness of the mechanisms involved, as we will explain.

Many tools work on a parse tree or an AST that also contains variability information from the annotations, often enriched with information on types for analysis and transformation. It must be possible to identify variable code as compared to the base code. Hence, it is desirable to map **#ifdef** annotations to complete subtrees in the AST, so that reasoning about the source-code structure and its annotations is facilitated on the basis of a uniform code representation. Many approaches that advocate the uniform representation of **#ifdef** variability for the implementation of proper tool support. There are two common scenarios in the area of software product lines: **#ifdef** transformation [Adams et al., 2009; Lohmann et al., 2006] and **#ifdef** management [Atkins et al., 2002; Heidenreich et al., 2008; Janzen and De Volder, 2004; Kästner and Apel, 2008, 2009].

A clean mapping from annotations to AST elements has the following advantages:

- The AST representation contains all information of a program including its variability. This simplifies tools for **#ifdef** transformation and **#ifdef** management, and programmers can safely analyze and transform the AST. We do not need to preprocess the code upfront, transform it, and then revert the preprocessing step. Instead, we can reason about code and presence conditions as part of the uniform representation directly. For example, when we transform code by extracting an annotated sequence of statements

into an aspect, we can make sure that we move the statements correctly without loss of variability information.

- The mapping of `#ifdef` annotations to AST nodes ensures the absence of syntax errors in conditionally compiled code. When we allow annotations only on structural elements but not on arbitrary tokens, such as single braces, the removal of `#ifdefs` cannot create invalid parse trees or ASTs [Kästner et al., 2009b].
- Based on an AST, we can reason about types and control flow, or perform other static analyses and model checking [Classen et al., 2010], always including information on variability. Analyses become more complex due to preprocessor annotations. For example, a variable can have different types depending on the selection of preprocessor constants. Nevertheless, such analyses can still be performed on the entire variable code base in a single step (instead of generating all variants upfront). For example, Kästner et al. [2012a] have built a type system that compares annotations between function declarations and function calls, which depend on variability annotations in a single AST.

To summarize G1.3, we investigate whether developers have already been using `#ifdef` directives in a structured way.

3.1.2 Questions

How does program size influence variability? (Q1)

A large software system usually provides more configuration options than a small software system. The reason is that in a large code base, the potential for variability is likely to be higher than in a small one. This is due to a possibly higher number of implemented functionalities. We are interested in how many configuration options are present in the source code, because they mark all possible parameters for system configuration and form the basis of the configuration space of a configurable system. Furthermore, we are interested in the amount of variable code, because it represents variability at the source-code level at which the programmer operates. Both, the number of configuration options as well as the amount of variable code have an influence on the analysis and transformation of configurable systems, as more variants and, consequently, more variable code has to be considered.

How complex are extensions using `#ifdefs`? (Q2)

This question addresses the presence of scattered and tangled implementations of configuration options. CPP's language definition allows programmers to nest preprocessor directives (e.g., an `#ifdef` can be nested inside another `#ifdef`, which gets evaluated only if the presence condition of the outer `#ifdef` is satisfiable). The use of `#ifdef` nesting as well as the use of logical operators, such as `&&` or `||`, results in complex presence conditions for variable code fragments. Analysis and transformation tools should be able to handle complex presence

conditions to avoid a potentially exponential number of redundant computations that are necessary to obtain analysis and transformation results. For example, redundant computations are avoidable if a tool handles the presence condition `#if defined(A) || defined(B)` directly instead of handling all combinations of A and B in isolation.

We are interested in whether a higher number of configuration options increases the occurrences of scattering and tangling. Furthermore, we are interested in the nesting depth of `#ifdefs`, a special form of code tangling.

At which level of granularity are extensions applied? (Q3)

Question Q3 is directly motivated by prior discussions on the granularity of extensions in software product lines [Kästner et al., 2008, 2009a]. The discussions revolve around the necessity of fine-grained modularization techniques, such as statement and expression extensions or function-signature changes. In particular, the granularity issue raises the question at which level variability-aware tools should support variability. So it directly affects design decisions of tool writers. If we limit `#ifdef` usage to a coarse grain, i.e., by annotating only function or type definitions, we can simplify the development of tool support. For example, keeping this condition, it is sufficient to implement a renaming refactoring for local function variables that is not variability-aware. The reason is that, since `#ifdefs` cannot occur inside the function body, the refactoring engine implementing the renaming does not need to incorporate presence conditions.

Although CPP allows a programmer to annotate code even at the finest level of granularity [Kästner et al., 2008], little is known about the necessity of making such fine-grained extensions. Further discussions on the modularization of configuration options also motivate this question [Kästner et al., 2007; Murphy et al., 2001; Sullivan et al., 2005], because most modular implementation techniques for configurable systems either lack the ability to make fine-grained extensions [Kästner et al., 2008; Murphy et al., 2001] or require workarounds [Kästner et al., 2007; Rosenmüller et al., 2007; Apel et al., 2008]. To this end, we are interested in identifying the levels at which `#ifdefs` are used to make fine-grained extensions. We also want to know how often developers use `#ifdefs` at these levels.

Which types of extension occur? (Q4)

While the previous three questions target the analysis and transformation process of configurable systems, question Q4 addresses the possibility of transforming `#ifdef`-annotated code fragments into alternative code representations using different implementation techniques. To this end, we discuss aspects and feature modules as transformation targets, which have both been used in literature [Kästner et al., 2009a; Adams et al., 2009; Lohmann et al., 2009, 2006; Reynolds et al., 2008]. The underlying concepts of aspect-oriented programming and feature-oriented programming have different strengths and weaknesses for the implementation of configurable software, which have thoroughly been discussed [Apel et al., 2008, 2013a]. For example, homogeneous extensions can be implemented easily with aspect-oriented language

extensions of C, because they provide a quantification and weaving mechanism (pointcut and advice) for extending multiple places in the source code at the same time. Heterogeneous extensions can be specified by simpler mechanisms, such as mixins or feature modules [Apel et al., 2008; Batory et al., 2004]. Furthermore, question Q4 is motivated by a prior case study by Apel [2010] on the development of configurable systems with aspect-oriented programming. In this study of eleven programs written in AspectJ, Apel observed that most extensions in AspectJ code are heterogeneous. We want to explore whether this observation also applies to configurable systems implemented with CPP.

How often do disciplined and undisciplined annotations occur? (Q5)

Based on the idea of mapping annotations to elements of the underlying source-code structure, we propose a definition of disciplined annotations. Actually, it is quite difficult to find a common definition, because different tools may have different requirements (e.g., some can handle annotated function parameters, others cannot). Here, we put forward a conservative definition:

Definition. Disciplined Annotations: *In C, annotations on one or a sequence of entire functions and type definitions (e.g., `struct`) are disciplined. Furthermore, annotations on one or a sequence of entire statements and annotations on members of type definitions are disciplined. All other annotations are undisciplined.*

We believe that writing disciplined annotations (as defined here) is sufficient for the representation of variability, and we expect that the majority of `#ifdef` directives in C programs is already disciplined for three reasons:

1. Developers prefer disciplined annotations, because they consider undisciplined annotations hard to read. For example, Baxter and Mehlich [2001] report of a project that contained some undisciplined annotations: *“The reaction of most staff to this kind of trick is first, horror, and then second, to insist on removing the trick from the source.”*
2. Some software projects have *coding guidelines* that state how to use the preprocessor. They typically favor disciplined over undisciplined annotations. For example, in LINUX kernel development, guidelines state that programmers shall annotate entire functions instead of arbitrary source-code fragments: *“Code cluttered with `ifdefs` is difficult to read and maintain. Don’t do it. Instead, put your `ifdefs` in a header, and conditionally define static inline functions, or macros, which are used in the code.”*¹
3. The developers of previous tool support for configurable systems used a similar definition for disciplined annotations [Baxter and Mehlich, 2001; Garrido and Johnson, 2003, 2005]. It seems that disciplined annotations are sufficient for most problems in everyday software development. Arbitrary undisciplined annotations are simply unnecessary

¹/Documentation/SubmittingPatches in the LINUX source

in most cases. Even though variability involves changes at the subfunction level, it is questionable whether the usefulness of annotations at the level of expressions or parameters outweighs their problems (e.g., control-flow analysis and readability).

Using our definition, we can map functions, type definitions, and statements in a straightforward way to subtrees in the AST. This can be exploited when enriching an existing C grammar with annotations (cf. Figure 3.3). Within the grammar, preprocessor directives simply wrap production rules of the host language to represent annotated subtrees in an AST. The extended grammar in Figure 3.3 supports optional or alternative function definitions with CPP directives (additional productions for annotating functions are highlighted). The resulting parser is capable of parsing the entire unprocessed source code (pre-CPP) in a single step. Note that, due to CPP's unlimited annotation capabilities, it is difficult to write a preprocessor-aware grammar that covers all possibly undisciplined annotations. Some researchers even consider it impossible [Padioleau, 2009]. But when we enforce disciplined annotations, this approach becomes practical. When preprocessor directives are already part of the grammar and, hence, recognized by the parser, we can assign parsed annotations directly to code fragments of the AST.

```

1 translation_unit
2   : external_declaration
3   | translation_unit external_declaration
4   ;
5 external_declaration
6   : function_definition
7   | '\#' 'if' cppexp nl function_definition nl cppthenfunc
8   | declaration
9   ;
10 cppthenfunc
11  : '\#' 'endif' nl
12  | '\#' 'else' nl function_definition nl '\#' 'endif' nl
13  | '\#' 'elseif' cppexp nl function_definition nl cppthenfunc
14  ;
15 function_definition ...

```

Figure 3.3: Excerpt of a CPP-extended ISO/IEC 9899 lexical C grammar; rules for preprocessor directives are highlighted (Line 7 and Line 10 to 14); `cppexp` is the condition; `nl` is a newline; `cppthenfunc` represents the `#endif` or alternative function definitions.

Figure 3.4 contains some examples of disciplined annotations taken from the text editor VIM: an annotation on an entire function (cf. Figure 3.4a), an annotation on an entire statement including a nested substatement (cf. Figure 3.4b), and an annotation on a field inside a struct (cf. Figure 3.4c).

One may argue that our definition disciplined annotations is too strict, because we could also allow disciplined annotations on expressions (cf. Figure 3.5d), on parameters (cf. Figure 3.5g), on `case` blocks in a `switch` statement (cf. Figure 3.5h), on a jump target of a `goto` statement (cf. Figure 3.5f), or on the `else` branch of an `if` statement (cf. Figure 3.5b). In all these

<pre> 1 #if defined(__MORPHOS__) \ 2 && defined(__libnix__) 3 extern unsigned long * 4 __stdfilesdes; 5 6 static unsigned long 7 fdtofh(int filedescriptor) { 8 return __stdfilesdes[9 filedescriptor]; 10 } 11 #endif </pre> <p style="text-align: center;">(a) compilation unit</p>	<pre> 1 void tcl_end() { 2 #ifdef DYNAMIC_TCL 3 if (hTclLib) { 4 FreeLibrary(hTclLib); 5 hTclLib = NULL; 6 } 7 #endif 8 } </pre> <p style="text-align: center;">(b) subfunction level</p>	<pre> 1 typedef struct { 2 typebuf_T save_typebuf; 3 int typebuf_valid; 4 struct buffheader 5 save_stuffbuff; 6 #if USE_INPUT_BUF 7 char_u *save_inputbuf; 8 #endif 9 } tasave_T; </pre> <p style="text-align: center;">(c) subtype level</p>
--	---	---

Figure 3.4: Examples of disciplined annotations in VIM.

cases we can map the annotation to an entire subtree of the AST. Actually, we can even map partial annotations on **if**, **for**, or **while** statements that do not include the nested body as in Figure 3.5a. We would map the annotation to an individual AST element and not to an entire subtree. This strategy, known as wrapping, is discussed by Kästner et al. [2009b]. A typical wrapper is an annotation of some control structure, e.g., an **if** statement (cf. Figure 3.5a) or a **for** loop (cf. Figure 3.5c). All these fine-grained annotations (and several more) could be viewed as disciplined, but then the tools that work on the resulting AST would be more complex. Some tools benefit from disallowing annotations on expressions or parameters, because this way they need to consider fewer annotated code fragments and fewer transformation patterns [Kästner et al., 2009a]. One goal of our analysis is to find out whether our conservative definition of disciplined annotations is sufficient in practice or some or all fine-grained annotations should also be considered disciplined, because software engineers use them frequently.

We can classify some annotations as undisciplined without any doubt: ill-formed, in which already the number of **#ifdef** and **#endif** statements does not match. Furthermore, annotations that can produce syntax errors when removed, such as an annotation of an opening brace without an annotation on the corresponding closing brace. Figure 3.6 shows an example of ill-formed annotations in XTERM. At first sight, this code exhibits a syntax error if `__GLIBC__` and `USE_ISPTS_FLAG` are selected together but, according to the XTERM developers, this selection is not possible [Medeiros et al., 2013].

To summarize, with question Q5, we are interested in the annotation discipline that developers of current configurable systems have been using.

3.1.3 Metrics

To measure the different aspects of **#ifdef** usage with respect to the previously stated questions, we introduce a set of corresponding metrics. Next, we explain each metric, its measurement, and its utility for our study and for subsequent discussions.

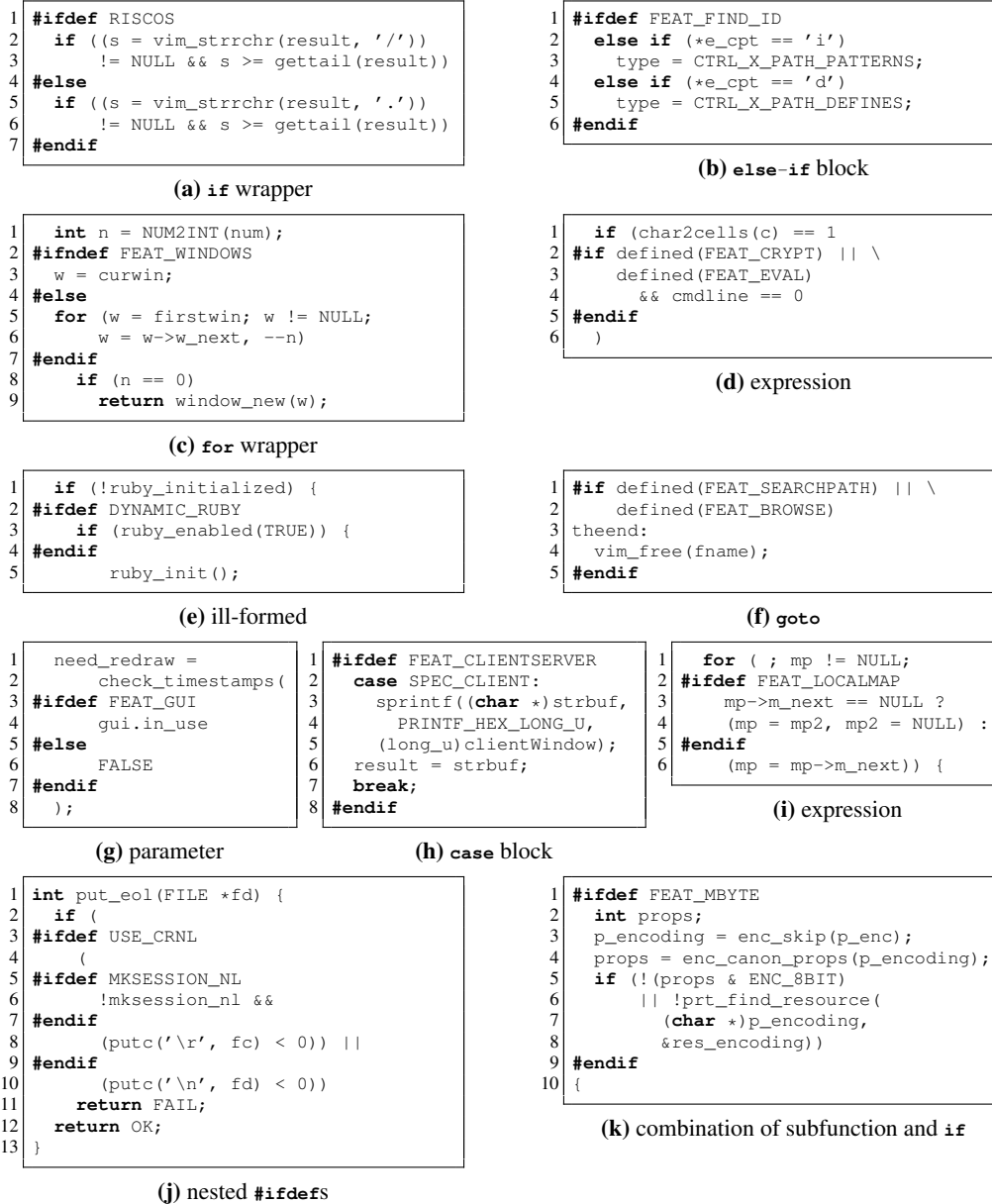


Figure 3.5: Examples of undisciplined annotations in VIM.

```
1 #if defined(__GLIBC__)
2 // additional lines of code
3 #elif defined(__MVS__)
4 result = pty_search(pty);
5 #else
6 #ifdef USE_ISPTS_FLAG
7     if (result) {
8 #endif
9         result = ((*pty = open("/dev/ptmx", O_RDWR)) < 0);
10 #endif
11 #if defined(SVR4) || defined(__SCO__) || defined(USE_ISPTS_FLAG)
12     if (!result)
13         strcpy(ttydev, ptsname(*pty));
14 #ifdef USE_ISPTS_FLAG
15     IsPts = !result;
16 }
17 #endif
18 #endif
```

Figure 3.6: Example of undisciplined annotations in XTERM.

Lines of Code (LOC)

The LOC metric quantifies the size of a software system. We measure it by counting the number of newlines of each normalized source code file and taking their sum. We use it later to discuss the influence of program size on other metrics.

Number of Configuration Options (CO)

The CO metric characterizes the configuration space of a configurable system. We measure this metric by extracting names of configuration options from presence conditions in the source code and summing them per project. For example, the CO value of `#if defined(BUBBLESORT) || defined(INSERTIONSORT)` is two. Note that we do not extract configuration options from `#define` directives, because these directives do not directly contribute to the configuration space of a system.

Lines of Variable Code (LVC)

The LVC metric quantifies the number of source-code lines enclosed in `#ifdef` directives. The metric tells us whether a small or a large fraction of the code base is variable. We extract this metric by counting the number of lines between two `#ifdefs` in source-code files and adding them up per project. Favre [1997] and Muthig and Patzke [2002] claim that, despite the problems `#ifdefs` cause, they are still useful, and developers can use them in small software systems with moderate effort. In our discussions, we use the CO and the LVC metrics as rough indicators of the analysis and transformation effort.

Scattering Degree (SD)

The SD metric quantifies the number of configuration options in different presence conditions. We measure this metric by extracting configuration options from `#ifdefs` and by calculating its average and standard deviation per project of all occurring configuration options. This metric provides insights into the complexity of configuration-option implementations. For example, a widely scattered option that extends a software system in several files and at multiple places requires a complex analysis and transformation than an option that applies only a few extensions within a single file.

Tangling Degree (TD)

The TD metric quantifies the number of different configuration options that occur in a presence condition. The term tangling has a non-standard meaning here. Usually, tangling refers to the mixture of code fragments with each other (side by side) or with the non-variable base code (cf. Section 2.1.2). Here, tangling refers to the mixture of configuration options in a presence condition, neglecting C code. The TD degree is an important metric that indicates to what degree variability-aware analysis and transformation tools can benefit from taking configuration knowledge into account. A static analysis that directly analyzes code annotated with `#if defined(A) || defined(B)` is faster than an analysis that analyzes both variants (one selecting A and one B) separately.

Nesting Depth of `#ifdefs` (ND)

The ND metric quantifies the average nesting depth of `#ifdefs`. We calculate the average and the standard deviation of all `#ifdefs` in a file and compute, based on these values, the average and standard deviation per project. Nesting is one source of exponential explosion in configurable systems implemented with CPP, because each additional nesting level doubles the preceding sub-configuration space, eventually leading to up to 2^n configurations for n levels. The higher the ND metric, the higher the analysis effort.

Granularity (GRAN)

Since CPP is independent of the host language, developers can use it to apply arbitrary extensions to the base code. These extensions are commonly divided into *coarse-grained extensions* and *fine-grained extensions* [Kästner et al., 2008]. Coarse-grained extensions add new functions or data structures, whereas fine-grained extensions add code fragments, such as statement and expression extensions or function-signature changes [Kästner et al., 2008]. To this end, we introduce several GRAN metrics (i.e., a family of granularity metrics) that represent the number of `#ifdefs` that occur at particular levels of granularity in the source code.

Based on prior work of Kästner et al. [2008] and on the capabilities of alternative product-line implementation techniques, we measure the GRAN metric at six granularity levels of interest: global level (GL; e.g., adding a structure or function; cf. Figure 3.7, Lines 2 to 4), function or type level (FL; e.g., adding an `if`-block or statement inside a function or a field to a structure; Lines 7 to 9), block level (BL; e.g., adding a code block; Line 15ff.), statement level (SL; e.g., varying the type of a local variable), expression level (EL; e.g., changing an expression; Line 19 to 21), or parameter level (PL; e.g., adding a parameter to a function declaration). We use the GRAN metric to discuss the development of tool support with respect to variability awareness. We measure it by counting the number of occurrences of annotations at each GRAN level and summing them up for each project. In our example, there are four GRAN metrics: one for global and block level, and two for function/type and expression level.

```

1 struct node *first = NULL;
2 #if DLINKED
3 struct node *last = NULL;
4 #endif
5
6 void insert(struct node *elem) {
7 #if BUBBLESORT
8     struct node *a,*b,*c,*e,*tmp = NULL;
9 #endif
10 #if INSERTIONSORT
11     struct node *a,*b = NULL;
12 #endif
13     if (NULL == first) first = elem;
14     else {
15 #if INSERTIONSORT
16         a = first;
17         b = first->next;
18         if (first->item
19 #if SORTASCENDING
20             >
21 #else
22             <
23 #endif
24         ...
25     }

```

Figure 3.7: Example of coarse-grained and fine-grained extensions.

Type (TYPE)

Developers extend a configurable system either with distinct extensions (heterogeneous) or with the same extensions, thus leading to code duplicates (homogeneous). The TYPE metric quantifies the number of occurrences of particular extensions in the source code. We measure this metric by comparing subsequent lines of source code that belong to the same presence condition, using string comparison (we will discuss this threat to validity later). Using this measurement, we distinguish three types: *homogeneous extension* (HOM), *heterogeneous extension* (HET), and their combination (HEHO). We use this metric to discuss possible transformations of `#ifdefs` into aspects or feature modules.

Disciplined (DIS) and Undisciplined (UNDIS) Annotations

To measure annotation discipline, we distinguish between disciplined (DIS) and undisciplined (UNDIS) annotations. According to our definition (cf. Section 3.1.2), disciplined annotations annotate entire functions and structures as well as members and statements. To measure the annotation discipline, we match each `#ifdef` annotation to certain patterns of disciplined (DIS) and undisciplined (UNDIS) annotations.

For patterns of disciplined annotations, we count the number of completely annotated

function and type definitions (FT; cf. Figure 3.4a). Furthermore, we count annotations on one or multiple statements inside a function or on members inside a type definition (SF; cf. Figures 3.4b and 3.4c) as disciplined. The identification of patterns that may be considered disciplined was an iterative process. We started with patterns with which we were familiar with and iteratively added additional patterns that we encountered during the manual inspection of undisciplined annotations in the analysis results. As a result, we distinguish five patterns of undisciplined annotations: (1) partial annotations of an **if** statement (e.g., an annotation of the **if** condition or the **if**-then branch without the corresponding **else** branch (IF; cf. Figure 3.5a)), (2) annotations on a single branch inside a **case** block (CA; cf. Figure 3.5h), (3) annotations on an **else-if** branch inside an **if-else** cascade (EI; cf. Figure 3.5b), (4) annotations on a parameter of a function declaration or a function call (PA; cf. Figure 3.5g), and (5) annotations on well-formed parts of expressions (EX; cf. Figure 3.5d, 3.5i, and 3.5j).

There are certain similarities between measuring the granularity and measuring the discipline of preprocessor annotations. For example, the GRAN metric GL (global level) includes annotations of functions and type definitions and is similar to the discipline metric FT (function or type level). While both metrics refer to the same elements in the AST representation, all discipline metrics additionally determine whether an annotation can be represented in a variable AST. The GRAN metrics target the development of novel language mechanisms for the representation of variability. In contrast, the discipline metrics (DIS and UNDIS) aim at the representation of variability in a uniform representation, such as variable ASTs.

Figure 3.1 summarizes our GQM model of measuring the influence of **#ifdef** usage on program analysis and program transformation. Based on this model, we describe our analysis to obtain answers to the research questions stated in our model.

3.2 Analysis

To answer our questions, we selected 42 different configurable systems (cf. Table 3.1). We selected these systems for two reasons. First, they cover different domains, such as operating systems, web servers, and database-management systems. They vary in size (from a few thousand lines of code to several million lines of source code). Using this criterion, we account for different CPP usages in practice. Second, all systems are primarily written in C. We limit our analysis to C, because it is widely used in software development with an enormous number of open-source software systems available. Although CPP is being used with other programming languages as well (e.g., C++), the limitation to a single language enables us to compare preprocessor use of different systems without the influence of the programming language itself. We discuss this threat to validity further in Section 3.3.2.

We consider the selected systems configurable, because they provide possibilities for configuration including support for different hardware platforms and application-specific configuration options. Additionally, some subject systems of our selection have already been discussed in the context of configurable systems. For example, many researchers used the LINUX kernel in their case studies and published numerous results [Tartler, 2013; Sincero, 2013; Berger, 2013] that

enter into our discussions.

To rule out deviations due to different coding practices and to make the results of different systems comparable, we normalize the source code of each software system before we obtain the metrics. The normalization includes:

- the formatting of source code, unifying different coding conventions such as the position of braces,
- rewriting `#ifdef` directives to normalize different usages of conditional-inclusion macros (e.g., `#ifdef A` vs `#if defined(A)`),
- deleting include guards, a common preprocessor pattern that is solely used to avoid multiple inclusions of files and that does not add variability to a system,
- removing comments and empty lines to avoid misinterpretations of obtained results that include the counting of source-code lines.

We used the tool SRC2SRCML² to generate an Extensible Markup Language (XML) representation of the source code [Maletic et al., 2004; Collard et al., 2013] to measure the granularity of extensions made with CPP. The XML representation contains all information of the basic language C in AST form, including CPP directives. The two levels of programming (the meta level of CPP and the source-code level of C) have separate namespaces in the XML representation, which gave us the opportunity to conduct a combined and a separate analysis of the source code. For our analysis, we split alternative annotations, such as `#ifdef-#else-#endif`, into two annotations, one from `#ifdef` to `#else` and one from `#else` to `#endif`, because they enclose different code fragments and have to be analyzed separately.

We measured the metrics introduced in the Section 3.1 on the XML representation using our tool CPPSTATS. CPPSTATS gathers the desired information by traversing the XML representation and processing the raw data. The tool and the comprehensive data of each system are available on our project’s website.³ Table 3.2 on page 44 depicts the condensed data of all 42 software systems.

We omitted 46 of 105 466 files during the analysis (0.44 % of all files analyzed), because either SRC2SRCML could not correctly parse these files or they contained incomplete annotations, such as an `#ifdef` without the corresponding `#endif`. Without a proper AST representation, we were not able to determine the metrics. Furthermore, we excluded all files from GCC’s test suite, because it contains many regression tests for the compiler implementation with potentially erroneous C and CPP code.

²<http://www.srcml.org/>

³<http://fosd.net/cppstats/>

Software system	Version or date of download	Domain
APACHE ¹	2.2.11	Web server
BERKELEY DB ¹	4.7.25	Database system
BUSYBOX ¹	1.18.5	System utility
CHEROKEE ¹	0.99.11	Web server
CLAMAV ¹	0.94.2	Anti-virus software
DIA ¹	0.96.1	Diagramming software
EMACS ¹	22.3	Text editor
FREEBSD ¹	7.1	Operating system
GCC ¹	4.3.3	Compiler framework
GHOSTSCRIPT ¹	8.62.0	Postscript interpreter
GIMP ¹	2.6.4	Graphics editor
GLIBC ¹	2.9	Programming library
GNUMERIC ¹	1.9.5	Spreadsheet application
GNUPLOT ¹	4.2.5	Plotting tool
IRSSI ¹	0.8.13	IRC client
LIBXML 2 ¹	2.7.3	XML library
LIGHTTPD ¹	1.4.22	Web server
LINUX ¹	2.6.28.7	Operating system
LYNX ¹	2.8.6	Web browser
MINIX ¹	3.1.1	Operating system
MPLAYER ¹	1.0rc2	Media player
MPSOLVE ²	2.2	Mathematical software
OPENLDAP ¹	2.4.16	LDAP directory service
OPENSOLARIS ³	(2009-05-08)	Operating system
OPENSSL ¹	1.0.1c	Cryptographic library
OPENVPN ¹	2.0.9	Security application
PARROT ¹	0.9.1	Virtual machine
PHP ¹	5.2.8	Program interpreter
PIDGIN ¹	2.4.0	Instant messenger
POSTGRESQL ¹	(2009-05-08)	Database system
PRIVOXY ¹	3.0.12	Proxy server
PYTHON ¹	2.6.1	Program interpreter
SENDMAIL ¹	8.14.2	Mail transfer agent
SQLITE ¹	3.6.10	Database system
SUBVERSION ¹	1.5.1	Revision control system
SYLPHEED ¹	2.6.0	E-mail client
TCL ¹	8.5.7	Program interpreter
VIM ¹	7.2	Text editor
XFIG ¹	3.2.5	Vector graphics editor
XINE-LIB ¹	1.1.16.2	Media library
XORG-SERVER ⁴	1.5.1	X server
XTERM ¹	2.4.3	Terminal emulator

¹<http://freecode.com>; ²<http://www.dm.unipi.it/cluster-pages/mpsolve/>;
³project discontinued ⁴<http://x.org>;

Table 3.1: Analyzed software systems.

3.3 Interpretation and Discussion

We use the collected data to provide answers to our five research questions (cf. Figure 3.1). The percentages given in this section are the mean and the standard deviation per project ($\mu \pm \sigma$). All plots show the relationship between LOC/CO and some other that metric we introduced in Section 3.1.3. Additionally, we calculate the correlation coefficient, including its significance level (i.e., p value), between related metrics using the method of Kendall [Kendall and Babington Smith, 1939], because the input data regarding the metrics are not normally distributed.

3.3.1 Answering the Research Questions

Our answers to the questions stated in Section 3.1.2 are solely based on the metrics (cf. Section 3.1.3). We do not use additional sources, such as developer interviews or code inspections, because they would have been elusive to obtain for the large number of software systems we analyzed.

How does program size influence variability? (Q1)

The data reveal that the variability of a software system increases with its size (cf. Figure 3.8a). The metrics LOC and CO as well as LOC and LVC correlate strongly. An explanation for this correlation is the observation that larger software systems usually exhibit more configuration options and, consequently, are more variable. The amount of variable source code (LVC metric) in each project correlates with its size and is $21 \pm 15\%$, on average (cf. Figure 3.8b).

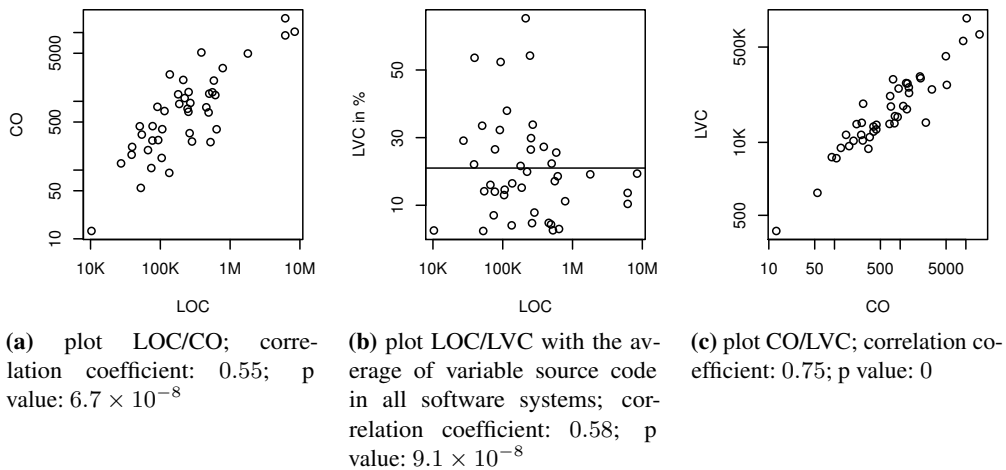


Figure 3.8: Variability plots: lines of code vs configuration options (LOC/CO), lines of code vs lines of variable code (LOC/LVC), and configuration options vs lines of variable code (CO/LVC).

The LVC metric reveals two interesting issues. First, we found that in some mid-sized software systems such as LIBXML2, OPENVPN, SQLITE, and VIM the amount of variable code exceeds 50 % of the code base. Second, the four largest software systems (FREEBSD, GCC, LINUX, and OPENSOLARIS) contain smaller percentages of variable source code than the average. One reason is that the variable code base of a system is not solely implemented with `#ifdef` directives. Dietrich et al. [2012b] observed that only one third (approximately 33.5 %) of all configuration options in LINUX occur in CPP code. The remaining two thirds of configuration options are referenced in LINUX build system (cf. Section 2.1.1). The build system controls the selection/deselection of source files containing the driver implementations of configurable hardware resources. Another explanation may be that, due to the size of these systems, the specification of configuration options is more complex than that of smaller systems. The higher complexity aligns with possibly more scattered and tangled configuration options, a correlation which we address next.

How complex are extensions using `#ifdefs`? (Q2)

The complexity of CPP-based implementations increases with the size of a system. That is, larger systems exhibit higher values of complexity metrics (SD, TD, and ND). Nevertheless, we cannot observe a relationship between the number of configuration options in a system and the complexity. For the SD metric, the p value of the computed correlation coefficient is larger than 0.05, which means that the result is not significantly different from 0. Initially, we expected the complexity of presence conditions in configurable systems with a high number of configuration options to be higher than in smaller software systems, but the complexity remains the same. The reason for this expectation was the assumption that the higher the number of configurations options in a system the higher the number of potential configuration-option dependencies (in particular the number of options involved in dependencies). As a consequence, a higher number of configuration options results in higher values of complexity metrics.

Notably, the average and the standard deviation of the scattering degree is quite high in some systems (e.g., 21.11 ± 46.87 for SUBVERSION, 19.30 ± 82.13 for VIM, and 10.73 ± 96.67 for OPENSOLARIS). That is, a significant number of configuration options cause a high scattering degree and the respective implementation scatters possibly across the entire system. However, we cannot infer from the scattering degree the places where configuration options occur in the source code. The scattered option may only appear in a subsystem (e.g., a group of files).

The average and the standard deviation of the tangling degree are quite small in most systems (one to three, on average) and, consequently, the complexity of presence conditions is low. A lower complexity is preferable, because presence conditions of low complexity decrease the effort during analysis and transformation of a system.

Our third complexity metric (ND) shows that developers use `#ifdef` nesting moderately, i.e., in all software systems, the average ND is approximately 1. That is, the number of nested `#ifdefs` does not grow with the number of configuration options—CO metric. A lower ND is preferable, because it reduces the complexity of analyzing and transforming configurable software. Nesting increases the configuration space significantly. Each additional, nested

3 Understanding Preprocessor Annotations

Software system	LOC	CO	LVC	ND	SD	TD
APACHE	223 061	1 113	44 425	1.17± 0.17	5.57±15.22	1.74± 1.37
BERKELEY DB	186 893	915	28 400	1.09± 0.12	4.66±13.80	1.39± 0.82
BUSYBOX	180 206	1 264	38 989	1.11± 0.14	3.65± 5.79	1.53± 1.41
CHEROKEE	54 259	328	7 676	1.07± 0.08	3.70± 7.28	1.59± 1.12
CLAMAV	77 031	269	10 793	1.11± 0.16	6.22±12.44	1.47± 0.81
DIA	134 274	91	5 450	1.03± 0.04	5.40±16.69	1.03± 0.55
EMACS	254 236	1 364	75 897	1.27± 0.30	7.33±23.60	2.09± 1.47
FREEBSD	6 163 964	16 134	841 282	1.14± 0.17	6.84±39.43	1.64± 1.41
GCC	1 794 892	4 957	343 602	1.19± 0.22	7.16±20.89	2.36± 3.64
GHOSTSCRIPT	455 879	818	21 883	1.06± 0.08	3.96± 9.39	1.25± 1.12
GIMP	640 041	392	19 112	1.07± 0.08	6.53±15.48	1.65± 1.64
GLIBC	784 157	3 040	87 833	1.15± 0.16	7.28±31.44	1.70± 1.21
GNUMERIC	262 835	344	12 399	1.04± 0.06	3.71± 5.62	1.46± 1.46
GNUPLOT	77 497	435	20 555	1.16± 0.21	7.69±20.09	2.09± 2.11
IRSSI	52 500	55	1 259	1.05± 0.07	2.51± 2.43	1.20± 0.64
LIBXML2	212 996	2 046	139 055	1.58± 0.39	8.36±90.11	5.02± 3.02
LIGHTTPD	38 793	167	8 571	1.15± 0.19	4.68± 6.12	1.36± 0.97
LINUX	6 172 434	9 093	642 284	1.09± 0.11	6.07±47.99	1.44± 1.37
LYNX	114 787	722	43 567	1.24± 0.25	7.63±24.59	1.92± 1.19
MINIX	66 569	195	10 671	1.10± 0.10	5.03± 8.05	1.39± 0.83
MPLAYER	613 608	1 232	113 991	1.11± 0.14	5.83±14.44	1.50± 1.35
MPSOLVE	10 313	13	263	1.15± 0.04	2.54± 2.30	1.27± 0.45
OPENLDAP	252 497	704	66 820	1.20± 0.19	4.64± 7.81	1.62± 1.13
OPENSOLARIS	8 417 082	10 289	1 630 430	1.12± 0.13	10.73±96.67	1.81± 1.46
OPENSSL	268 984	946	90 991	1.25± 0.21	9.99±22.27	2.06± 1.93
OPENVPN	39 447	217	21 145	1.25± 0.26	6.00±16.23	1.69± 1.24
PARROT	106 584	393	15 567	1.19± 0.24	8.10±17.95	1.95± 1.81
PHP	584 470	2 002	149 522	1.19± 0.19	5.84±21.13	1.75± 1.33
PIDGIN	284 299	260	22 318	1.06± 0.10	4.83±16.45	0.99± 0.86
POSTGRESQL	491 511	688	21 296	1.10± 0.12	4.90±18.45	1.56± 1.23
PRIVOXY	27 371	125	7 965	1.16± 0.18	7.44±14.86	1.73± 1.01
PYTHON	387 313	5 130	105 648	1.18± 0.21	2.62±34.95	1.72± 1.05
SENDMAIL	90 854	831	29 303	1.24± 0.24	5.23±10.35	1.75± 1.11
SQLITE	93 278	273	48 842	1.29± 0.25	7.59±12.92	1.67± 1.10
SUBVERSION	523 641	255	13 612	1.08± 0.12	21.11±46.87	1.79± 1.23
SYLPHEED	104 603	150	13 607	1.06± 0.07	6.31±13.51	1.38± 1.43
TCL	136 662	2 459	22 518	1.15± 0.16	2.89±26.42	2.04± 1.29
VIM	245 587	779	133 203	1.57± 0.51	19.30±82.13	2.40± 1.47
XFIG	74 232	107	5 216	1.08± 0.12	4.67± 6.66	1.87± 1.99
XINE-LIB	502 270	1 291	112 202	1.12± 0.14	5.38±16.19	1.52± 1.27
XORG-SERVER	557 084	1 353	95 374	1.09± 0.11	9.10±28.38	1.85± 2.02
XTERM	50 731	433	17 003	1.21± 0.32	7.26±15.24	2.03± 1.62

LOC: lines of code; **CO:** number of configuration options; **LVC:** lines of variable code; **ND:** average nesting depth of `#ifdefs`; **SD:** average scattering degree; **TD:** average tangling degree

Table 3.2: Results configuration knowledge (G1.1) and variable code base (G1.2); part 1.

Software system	TYPE			GRAN					
	HOM	HET	HOHE	GL	FL	BL	SL	EL	PL
APACHE	120	1 972	94	1 858	2 163	827	44	35	13
BERKELEY DB	189	1 507	108	1 248	1 142	1 581	2	19	12
BUSYBOX	78	2 086	61	1 966	1 744	1 032	131	22	19
CHEROKEE	42	460	10	533	273	218	0	7	0
CLAMAV	41	463	41	848	385	402	1	0	2
DIA	3	144	16	439	222	76	7	0	5
EMACS	113	3 205	119	3 428	2 309	1 501	17	114	30
FREEBSD	2 771	29 819	2 293	48 520	32 485	19 032	615	688	589
GCC	480	9 261	320	11 554	5 828	2 731	258	369	54
GHOSTSCRIPT	145	1 144	67	2 162	1 214	687	15	12	23
GIMP	163	462	32	1 224	622	341	2	0	1
GLIBC	425	5 886	392	10 045	4 250	2 226	119	98	55
GNUMERIC	11	616	35	491	833	644	8	15	17
GNUPLOT	35	948	48	1 102	609	694	50	19	4
IRSSI	4	73	3	76	96	24	1	0	0
LIBXML2	161	2 689	76	6 913	1 967	935	0	29	0
LIGHTTPD	22	222	35	296	252	289	51	1	16
LINUX	435	17 330	877	30 651	19 974	7 048	499	77	120
LYNX	22	1 800	62	1 373	1 733	1 234	9	69	8
MINIX	30	367	18	625	309	253	4	3	0
MPLAYER	82	2 415	154	2 957	3 191	1 669	145	19	27
MPSOLVE	0	18	0	20	10	4	0	0	0
OPENLDAP	33	1 232	67	1 199	1 334	756	18	26	15
OPENSOLARIS	2 593	21 281	1 307	37 824	43 973	16 351	297	308	386
OPENSSL	106	2 353	143	2 590	2 794	1 530	25	73	10
OPENVPN	5	391	11	543	417	186	2	5	13
PARROT	129	599	52	1 290	640	253	1	2	0
PHP	336	3 614	175	4 690	3 052	1 990	145	53	30
PIDGIN	38	396	66	750	1 064	360	0	9	39
POSTGRESQL	64	1 025	72	1 585	1 394	757	0	23	12
PRIVOXY	7	281	15	299	313	187	6	9	0
PYTHON	233	6 227	107	2 218	6 771	883	26	31	75
SENDMAIL	49	1 683	59	1 246	1 053	1 021	70	65	27
SQLITE	7	782	12	806	667	323	13	6	3
SUBVERSION	186	194	41	2 668	1 532	415	4	4	4
SYLPHEED	4	276	19	609	398	156	13	5	11
TCL	79	2 978	38	2 947	682	629	3	18	4
VIM	108	4 085	205	2 852	4 356	3 854	107	713	174
XFIG	1	152	10	197	156	101	2	6	1
XINE-LIB	77	2 544	107	2 963	2 877	1 208	65	11	18
XORG-SERVER	264	2 786	224	5 109	2 962	2 329	45	84	18
XTERM	29	963	43	975	859	500	1	21	4

TYPE: # extensions measuring the type (**HOM:** homogeneous; **HET:** heterogeneous; **HEHO:** heterogeneous and homogeneous); **GRAN:** # extensions measuring the granularity (**GL:** global level; **FL:** function or type; **BL:** block; **SL:** statement; **EL:** expression; **PL:** function parameter)

Table 3.2: Results of configuration knowledge (G1.1) and variable code base (G1.2); part 2.

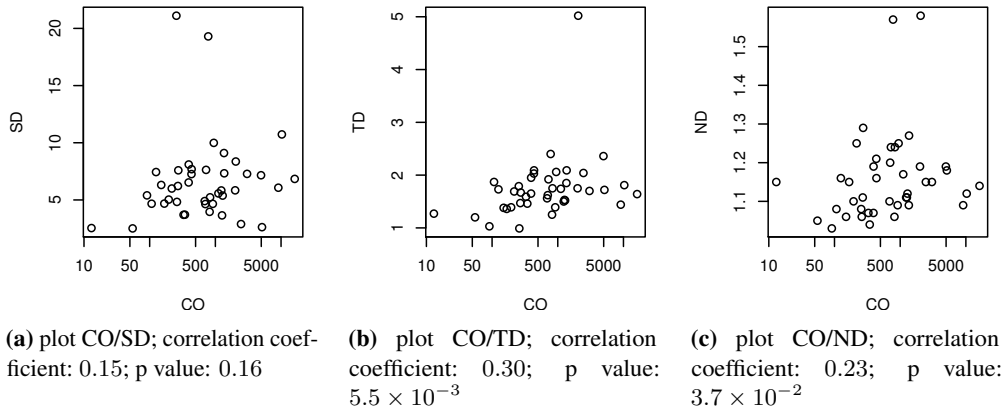


Figure 3.9: Complexity plots: configuration options vs scattering degree (CO/SD), configuration options vs tangling degree (CO/TD), and configuration options vs nesting depth (CO/ND).

#ifdef doubles the number of configurations and, hence, increases the effort of conducting an analysis or transformation approach. Potentially, 2^n different configurations with n nested **#ifdef** directives have to be explored. We also determined the maximum number of nested **#ifdefs** in a file. Two projects (FREEBSD and GCC) reached a maximum number of 24. The rest of the systems remained at a level of 2 to 9. We argue that high numbers of nested **#ifdefs** are hard to manage manually (with respect to program analysis and program transformation) and automatic solutions are necessary. Furthermore, a high nesting depth may reduce the potential for transformations. Since a nested **#ifdef** depends on the enclosing one, the dependence between both **#ifdef** directives has to be taken into consideration to decide whether a transformation should be applied.

At which level of granularity are extensions applied? (Q3)

Our data reveal that programmers use fine-grained extensions (e.g., statement or expression extensions) infrequently. Most extensions occur at the global level (GL metric; $45 \pm 11\%$): annotating functions, type declarations/definitions, or (re-)definitions of macros using **#define** directives. Below the global level, the second largest set are extensions that occur at the function and block level (FL and BL metric; $34 \pm 9\%$ and $19 \pm 7\%$): annotations (e.g., an **if**-block or a statement) inside a function. The overall occurrence of fine-grained extensions is $1.8 \pm 2.4\%$, on average. Two projects use them slightly more frequently: LIGHTTPD with 6% statement extensions and VIM with 6% expression extensions. The results draw a rather diverse picture of **#ifdef** usage in practice, but it seems that automatic tool support should be able to handle fine-grained extensions also.

From the perspective of transforming configurable systems, the results are promising. Since most extensions occur at a high level of granularity, chances are good that they can be trans-

formed into alternative code representation, such as aspects or feature modules. In particular, aspects are favorable, because they enable extensions below the function or type level by addressing particular extension points in the source code. By contrast, the application of specific modularization techniques for implementing fine-grained extensions seem to be limited [Rosenmüller et al., 2007; Fowler, 1999; Murphy et al., 2001].

However, the data do not reveal whether an implementation technique is applicable and whether workarounds for refactorings using either feature modules or aspects may be necessary [Kästner et al., 2007; Rosenmüller et al., 2009].

Which types of extension occur? (Q4)

Our data show that $89 \pm 9\%$ of the extensions are heterogeneous (HET metric). Homogeneous extensions (HOM metric) add up to $6 \pm 8\%$, and the combination of both extension types (HEHO metric) sums up to $5 \pm 3\%$. Aspects are well known for their ability to implement homogeneous extensions [Apel et al., 2008]. We observe that 6% of the extensions would benefit from aspects; 89% would suffice with simpler mechanisms, such as mixins or feature modules; for the rest of the extensions, a combination of aspects and feature modules would be profitable [Apel et al., 2008]. The data coincide with an analysis conducted by Apel [2010], which revealed that most extensions are heterogeneous. Apel analyzed the use of AspectJ rather than CPP, though.

How often do disciplined and undisciplined annotations occur? (Q5)

In Table 3.3, we list the results of our analysis: we present the number of occurrences of disciplined and undisciplined preprocessor uses for all 42 projects. The key result is that $85.1 \pm 6.3\%$ of all annotations are disciplined. Disciplined preprocessor use in the software projects ranges from 69.1% in SUBVERSION to 100.0% in MPSOLVE. All systems except MPSOLVE contain some undisciplined annotations.

Looking more closely at the disciplined annotations, we found that $27.0 \pm 11.7\%$ of all annotations wrap entire functions and type definitions and $58.1 \pm 10.7\%$ wrap statements or type members.

The most frequent undisciplined annotations are annotations on **case** blocks in **switch** statements (CA; $4.1 \pm 3.2\%$). Except for IRSSI and MPSOLVE, they occur in every project. Partial annotations on **if** statements (IF; $2.5 \pm 2.0\%$) are less frequent, but still occur several times in every project except for LIGHTTPD and MPSOLVE. Annotations on **else-if** (EI; $0.4 \pm 0.4\%$), parameters (PA; $0.3 \pm 0.4\%$), and expressions (EX; $0.8 \pm 1.0\%$) occur infrequently and only in some projects. There are only few projects with an exceptionally high number of occurrences of such patterns (up to 5.6%), for example, GCC, SENDMAIL, or VIM.

We were not able to classify the remaining annotations automatically ($6.9 \pm 5.0\%$; range from 0.0 to 28.7%). Among these are ill-formed annotations and infrequent patterns (or combinations of identified patterns), such as in Figure 3.5c, 3.5e, 3.5f, and 3.5k. The reason

Software system	#AA	Disciplined		Undisciplined					
		% FT	% SF	% IF	% CA	% EI	% PA	% EX	% NC
APACHE	3494	17.1	64.1	2.2	5.1	0.7	0.2	0.7	9.9
BERKELEY DB	3129	13.2	75.7	0.8	4.1	0.0	0.3	0.6	5.3
BUSYBOX	3616	24.0	63.9	2.3	4.3	1.1	0.4	0.5	3.6
CHEROKEE	640	25.8	51.7	6.6	11.4	0.3	0.0	0.2	4.1
CLAMAV	998	20.8	61.4	3.3	2.5	0.1	0.0	0.0	11.8
DIA	385	17.9	68.6	0.5	1.3	0.0	1.3	0.0	10.4
EMACS	5185	28.1	61.1	2.5	2.2	0.5	0.3	1.8	3.4
FREEBSD	72306	29.6	55.4	2.2	6.4	0.3	0.4	0.8	4.9
GCC	14357	34.5	48.5	1.7	2.2	0.3	0.2	2.4	10.1
GHOSTSCRIPT	2793	37.6	52.1	1.0	2.9	0.2	0.3	0.4	5.5
GIMP	1406	34.1	58.0	0.7	2.7	0.2	0.1	0.0	4.2
GLIBC	11651	39.8	43.1	2.7	5.0	0.2	0.2	0.6	8.2
GNUMERIC	1394	13.4	71.8	2.4	0.9	0.4	0.9	0.6	9.7
GNUPLLOT	1826	31.4	48.1	4.2	7.3	0.4	0.1	0.9	7.6
IRSSI	149	14.8	71.1	6.7	0.0	0.0	0.0	0.0	7.4
LIBXML2	9161	69.0	24.6	0.9	2.2	0.7	0.0	0.3	2.2
LIGHTTPD	590	16.6	74.1	0.0	5.3	0.0	0.0	0.2	3.9
LINUX	41403	36.9	55.8	1.1	3.5	0.2	0.2	0.1	2.3
LYNX	3590	17.9	62.0	5.4	3.8	0.3	0.2	1.8	8.7
MINIX	778	36.6	58.2	0.8	1.4	0.1	0.0	0.1	2.7
MPLAYER	5696	21.5	62.0	2.8	5.1	0.1	0.4	0.3	7.8
MPSOLVE	30	53.3	46.7	0.0	0.0	0.0	0.0	0.0	0.0
OPENLDAP	2581	22.9	65.3	2.0	5.0	0.0	0.2	0.9	3.6
OPENSOLARIS	76438	19.1	57.9	1.6	3.4	0.2	0.3	0.3	17.1
OPENSSE	5376	24.4	58.7	3.1	3.4	1.8	0.0	1.3	7.3
OPENVPN	877	30.2	63.1	1.7	0.6	0.7	1.5	0.6	1.7
PARROT	1473	40.1	54.4	0.5	3.0	0.1	0.0	0.1	1.8
PHP	7167	28.1	55.6	1.9	7.6	0.1	0.3	0.5	5.8
PIDGIN	1559	17.0	68.1	0.8	0.7	0.0	1.8	0.6	11.0
POSTGRESQL	2862	22.5	58.5	1.6	5.5	0.3	0.3	0.6	10.6
PRIVOXY	652	19.6	54.8	4.6	8.7	0.0	0.0	1.1	11.2
PYTHON	8584	11.8	78.0	0.8	2.8	0.6	0.8	0.3	4.9
SENDMAIL	2662	17.5	58.6	3.1	12.7	1.4	0.5	2.2	4.1
SQLITE	1431	31.7	56.5	2.9	3.2	0.1	0.1	0.4	5.2
SUBVERSION	2642	25.6	43.5	1.8	0.1	0.0	0.2	0.2	28.7
SYLPHEED	813	25.6	58.4	1.4	1.0	0.5	1.0	0.5	11.7
TCL	2683	52.3	28.8	1.0	14.5	0.0	0.1	0.6	2.7
VIM	11450	15.9	58.5	7.9	2.8	1.4	0.8	5.6	7.1
XFIG	360	26.4	55.0	7.5	2.8	1.1	0.3	1.4	5.6
XINE-LIB	4382	22.6	63.0	1.2	3.1	0.0	0.3	0.2	9.6
XORG-SERVER	6999	25.3	59.8	3.0	5.0	0.4	0.2	1.1	5.2
XTERM	1761	22.5	64.5	4.4	4.9	0.2	0.2	1.1	2.3

AA: all annotations; FT: function and type definitions; SF: statement or field; IF: `if` wrapper;

CA: variable `case`; EI: variable `else if`; PA: parameter; EX: expression; NC: not classified;

Table 3.3: Results of variability representation (G1.3).

for SRC2SRCML’s inability to identify these few patterns is that it is based on heuristics. We discuss this limitation as a threat to validity in Section 3.3.2.

The results raise a number of questions. If 85 % of all annotations are disciplined, how do we handle the remaining 15 %? Should we define further annotation patterns as disciplined? Should we transform undisciplined into disciplined annotations?

Toward a Common Definition of Disciplined Annotations

Transforming undisciplined annotations into disciplined annotations requires a certain effort. Disciplined annotations are most useful when the community can agree on a common definition to establish a solid foundation for the development of inter-operable tools.

In Section 3.1.2 we proposed a conservative definition of disciplined annotations, which tools can handle easily. We already noted that several other annotation patterns could be regarded as disciplined at the expense of more complex tool implementations (for all tools), but at the benefit of less effort being necessary to transform legacy code into disciplined annotations. Here, we come back to this issue and initiate a discussion about the suitability of our definition.

First, our results show that, following our conservative definition, 85 % of all annotations are disciplined. Making the definition stricter is not feasible, because this way we cover fewer annotations without any further benefit. For example, when considering only annotations on function and type definitions disciplined, only 27 % of all annotations would count as disciplined; annotations on statements and members would be considered undisciplined, even though they are similarly easy to handle by tools.

Second, the most common pattern of undisciplined annotations that we recognized are annotations on **case** blocks inside **switch** statements as in Figure 3.5h ($4.1 \pm 3.2\%$). Such annotations occur in every sample project except MPSOLVE and IRSSI. They can easily be mapped to AST subtrees, but (similar to or even worse than **if-else** chains) they are surprisingly difficult to handle by tools due to the complicated control flow (in particular, in the presence of **break** statements).

Third, the next common pattern of undisciplined annotations ($2.5 \pm 2.0\%$) is a partial annotation of **if** statements (e.g., only the condition or the **if** branch without the alternative **else**) as in Figure 3.5a. Although this pattern occurs quite frequently in some projects (and, at least, once in every project, except for MPSOLVE and LIGHTTPD), handling such annotations is difficult, since we cannot map them to an entire AST subtree, but only to an individual AST element without its children.⁴ Aiming at an inter-operable infrastructure for many tools, we suggest not to regard such annotations as disciplined, but to transform the source code as we will discuss later in this section.

Fourth, the remaining identified patterns of annotated **if-else** chains ($0.4 \pm 0.4\%$), parameters ($0.3 \pm 0.4\%$), and expressions ($0.8 \pm 1.0\%$) occur infrequently (and not in all projects).

⁴Partial annotations complicate tool support, because, depending on the evaluation of the **#ifdef** directive, the child element belongs to two different AST elements. For example, the **if** statement (Line 8) in Figure 3.5c belongs either to the **for** loop (Line 5) or the function (not printed there) directly.

Due to the rarity of these patterns, we consider transforming the source code as the better and more inter-operable solution.

Finally, there are several annotations that do not fit into any of our patterns ($6.9 \pm 5.0\%$). The individual patterns (or combination of patterns) behind these annotations occur so infrequently that we can safely discard them as undisciplined. Overall, we interpret the results of our analysis as a confirmation of our initial conservative definition of disciplined annotations.

Handling the Remaining 15 % of Undisciplined Annotations

Tools aiming at disciplined annotations are able to handle most `#ifdefs` (85 %). This may be sufficient for simple analyses, such as the measurement of source-code complexity metrics or a rough estimation of the potential for refactorings as done by Adams et al. [2009]. Their tool simply ignored all annotations it did not understand. However, for some tools, a single undisciplined annotation may render the tool unsafe or the results useless. For example, concern refactoring tools may fail or even produce incorrect results, or concern management tools may show inconsistent views when they are unable to parse certain files or code fragments. Taking into account that, except for one project, all projects contain at least a few undisciplined annotations, this is a serious issue that deserves attention.

For handling the remaining 15 %, we see two possibilities. First, we can introduce more sophisticated tools that use heuristics to accept more annotations that we currently classify as undisciplined. Still, since arbitrary undisciplined annotations may occur in a software system, the effort to write such tools is extraordinary. Two examples are Garrido's refactoring tool CREFACTORY [Garrido, 2005] or Padioleau's preprocessor-aware parser YACFE [Padioleau, 2009]. But as both tools use heuristics, their results may not be 100 % correct.

Second, we can enforce disciplined annotations and require the developer to transform all remaining undisciplined annotations upfront into disciplined annotations. To make this approach practical, tool support is necessary for the automation of the transformation task. Enforcing disciplined annotations simplifies the development of tools significantly and, hence, can foster a community of tool developers for pre-CPP code.

To transform undisciplined annotations into disciplined annotations, `#ifdefs` must be expanded until they wrap entire functions, type definitions, statements, and members. Except for ill-formed annotations, undisciplined annotations can always be expanded to disciplined annotations. A brute-force algorithm that works in every case is to replicate the source code for every possible combination of `#ifdefs` and annotate the entire replicated code fragment.⁵ For example, Figure 3.10a shows an example of an undisciplined parameter annotation with two possible variants. We can replicate the code fragment (one version in which `FEAT_GUI` is defined and one version in which it is not defined) and annotate the entire statement in a disciplined form, as shown in Figure 3.10c. In the worst-case scenario, we would have to replicate the entire file multiple times and annotate the file's content accordingly. However, in

⁵Note that the use of `#include`, `#define`, and `#undef` macros does not hinder the expansion, because we can replicate these macros as well.

most cases, more sophisticated expansions at the level of statements or functions are appropriate.

An automatic expansion is not always that easy to make because of nested `#ifdefs` and scattered annotations. First, we found some annotations that were ill-formed and that did not map to nodes or subtrees in the AST or even a parse tree (cf. Figure 3.6). We argue that a programmer may expand these annotations manually after a tool has automatically identified them as undisciplined. Second, consider the nested `#ifdef` example in Figure 3.5j. `#ifdef Mksession_nl` is embedded in `#ifdef Use_crnl`, and the preprocessor evaluates it only in case the enclosing option `Use_crnl` evaluates to true. The expansion of this nesting leads to three alternatives: without `Use_crnl` and `Mksession_nl`, with `Use_crnl` and without `Mksession_nl`, and with both. This example may be simple, but it suggests that nesting occurs frequently, and a nesting depth beyond two is quite common (cf. Section 3.3.1). Expanding nested `#ifdefs` may lead to an exponential number of code clones as a result of the combinatorial explosion of all annotations involved. It is questionable whether it is feasible to expand deeply nested `#ifdefs` in favor of source-code refactoring. Since we limit the expansion to statements and functions, we believe that the expansion approach does not lead to a severe combinatorial explosion and that the expanded output is manageable by concern management tools.

There are two projects with parsing approaches (TYPECHEF [Kästner et al., 2011] and SUPERC [Gazillo and Grimm, 2012]) that are able to parse any C code with presence conditions into an AST representation including variability of the input. For example, TYPECHEF uses an `#ifdef`-enriched grammar specification. Rules for parsing `#ifdef` directives wrap rules for parsing C code. When parsing variable code, TYPECHEF's parser creates variable AST nodes using the presence condition of annotated source code. The resulting variability-aware AST is a one-to-one mapping of the annotated source code to variable AST nodes and can be processed further by variability-aware tools. TYPECHEF's grammar specification supports a set of disciplined annotations and automatically expands annotations that do not align with the specification. The expansion follows an approach similar to the one we just discussed.

Although a brute-force expansion is possible, developers can often write more elegant annotations manually. For example, for annotations on parameters or fragments of expressions, such as in Figure 3.5d, 3.5i, 3.5j, and 3.10a, we would rather introduce a variable and one or more annotated assignment statements, as illustrated in Figure 3.10b. Hence, we recommend a semi-automatic process instead of a fully automatic transformation. A tool locates undisciplined annotations and proposes expansions, but a developer can provide better implementations. More examples of automated and manual expansions of undisciplined `#ifdef` annotations (cf. Figure 3.5) are available on the project's website (<http://fosd.de/cppstats/>). Recently, Medeiros et al. [2014] proposed a set of five refactorings to replace the most common undisciplined annotations with disciplined annotations. The authors showed in a case study with 12 software systems that most undisciplined annotations could be replaced using small refactorings with a local effect on source code only. Most proposed refactoring patterns involve a combination of local code expansions, minor code transformations, such as extract local variable, and code manipulations using `#define` directives (cf. Section 3.1.3).

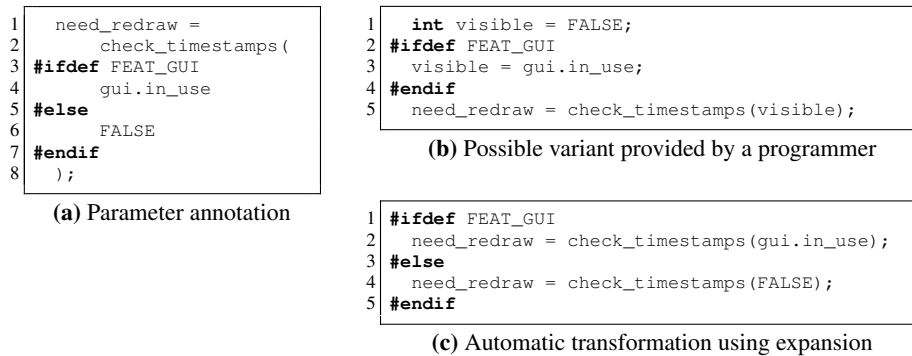


Figure 3.10: Undisciplined parameter annotation (a) and two disciplined variants of it (b and c).

3.3.2 Threats to Validity

Next, we describe two classes of threats to validity that are relevant for our empirical analysis: threats to internal validity (relation of the observed output to the input) and threats to external validity (generalizability of findings).

Threats to Internal Validity

- An automated analysis of the source code cannot distinguish well between high- and low-level configuration options. High-level options represent requirements of end users, whereas low-level options reflect design decisions made by developers (e.g., tracing or portability issues such as different types of signed integers for different compilers or platforms). These low-level options are typically of no interest to most end users [Tartler, 2013]. Making this distinction is not possible without additional expertise regarding the software systems and the domains. There are minor exceptions such as the LINUX kernel. The kernel developers distinguish between low-level and high-level options by prefixing configuration options relevant for end users with `CONFIG_`. We do not distinguish between high- and low-level options, because we are only looking at the usage of CPP’s variability mechanisms at the implementation level.
- Not all annotation patterns have a functional aspect. In this thesis, we omit the include-guard pattern. An include guard prevents the multiple inclusion of a header file during the compilation process and is per se disciplined, as it annotates the entire content of a header file including function definitions and so forth. Considering include guards in our statistics would have biased our results towards disciplined annotations, so we excluded them.
- The tool SRC2SRCML, which we use for our analysis, is partially based on heuristics that are used to infer an AST from unpreprocessed source code. In our experience,

SRC2SRCML's heuristic approach fails from time to time, especially when it comes to ill-formed annotations, with the result of producing an erroneous mapping of `#ifdef` directives to AST elements. Especially, the GRAN, TYPE, and DIS/UNDIS metrics heavily rely on the correctness of this mapping. The authors of SRC2SRCML use an extensive test suite to verify the relation of `#ifdefs` and source code in the XML representation. We believe that the enormous size of the data set amortizes this threat, and that the impact of a few false negatives (e.g., identifying a disciplined annotation not as such) can be neglected.

- We use string comparison to check the equality of different presence conditions to determine which code fragments belong to the same conditions. Our analysis cannot handle the semantic equivalence of presence conditions like `A && B` and `B && A`. Using CPP, developers sometimes state configuration knowledge in the form of predicates, e.g., `#if X > 10`. In contrast to presence conditions, which solely encode propositional formulae, reasoning about predicates requires a CSP solver [Benavides et al., 2005]. However, we found that these rarely occur in the analyzed configurable systems. We used the tool MAPLE⁶ to check the equivalence of predicates in some of our smaller systems. In a random inspection of 12 smaller systems, we found the error of not considering the equivalence to be below 2.5%. Due to time constraints caused by NP-completeness of CSP for larger software systems, such as the LINUX kernel, we omitted the equivalence check in our study.
- The distinction of extensions into heterogeneous and homogeneous is common in the software-engineering literature [Apel et al., 2008; Colyer and Clement, 2004]. Our tool distinguishes between heterogeneous and homogeneous source-code fragments by string comparison. Thus, character-based, syntactic changes in the source code are not classified correctly. The additional information we gather from the AST does not help here, because semantically equivalent code fragments can differ in syntax and, thus, are not recognized by a comparison of subtrees of the AST. The problem is even more serious, because developers use undisciplined annotations that destroy the source-code structure so that a correct AST representation is not available. Generally, the problem is related to the detection of code clones in configurable systems [Schulze, 2013]. Code clones are replicated code fragments across a system's implementation. In this context, homogeneous extensions represent code clones of equally annotated code fragments. Schulze [2013] analyzed `#ifdef` clones, i.e., code clones that are enclosed by `#ifdef` directives, in a subset of 15 configurable systems that were included in our study. The results of his study show that `#ifdef` clones are rare in practice. Nevertheless, classifying `#ifdefs` as heterogeneous and homogeneous extensions with code-clone analysis may lead to more precise results. Our conservative approximation using string comparison marks a lower bound of homogeneous extensions.

⁶<http://maplesoft.com>

Threats to External Validity

- The selection of sample software projects is crucial for an empirical analysis, because a biased selection leads to biased results. To control this confounding variable, we analyzed a large number of configurable systems of different domains and different sizes.
- Different coding conventions used in software systems may lead to wrong conclusions. For this reason, we preprocessed the analyzed source code by eliminating comments, empty lines, and include guards, and applied source code pretty printing.
- Using a large sample size, we have good reasons to believe that our results are representative for CPP usage in configurable systems written in C. However, we cannot generalize our results to other programming languages that also make use of the preprocessor, such as C++. This is because C++, for instance, provides additional capabilities for expressing variable source code (e.g., template metaprogramming), and, consequently, programmers may use CPP differently [Czarnecki and Eisenecker, 2000; Smaragdakis and Batory, 1998].
- Our analysis is limited to the source code and to **#ifdef** directives. A system's variable code base does not solely depend on configuration knowledge implemented with **#ifdef** directives. Additional configuration levels, such as build systems, are also used for configuration purposes in practice (cf. Section 2.1.1). A comprehensive analysis of such configuration levels is out of scope, because build-system use highly depends on the project and the involved technologies. In general, we expect the amount of configurable source code to increase when we include additional configuration levels—something other researchers have already observed for single systems, such as the LINUX kernel [Dietrich et al., 2012b]. Our focus on the source code coincides with developers' point of view on the implementation of configuration options.

3.4 Related Work

The preprocessor has been the subject of many studies. Next, we discuss these studies in light of our three sub-goals: configuration space, variable code base, and variability representation (cf. Section 3.1).

Configuration Knowledge

It has been commonly acknowledged that **#ifdef** directives in source code are an additional source of complexity, and that automated tools are necessary to analyze and extract configuration knowledge for further processing. Pearse and Oman [1997] suggested to measure conditional-compilation complexity similar to measuring traditional source-code complexity. In their experience, measuring the number of configuration options and **#ifdef** nesting is crucial for understanding conditional logic during analysis and maintenance tasks. Similarly,

measuring the number as well as the span and distribution (i.e., similar to our scattering and tangling degree) of configuration options helps developers to understand the build process, because developers often define and use configuration options in multiple files.

Krone and Snelting [1994] proposed the creation of a configuration model based on configuration knowledge extracted from `#ifdef` directives. The extracted model, which is a graph representation of configuration options and their dependencies, helps to detect configuration-related bugs, such as ensuring that some configuration options do not depend on each other or limiting the entropy of configuration options when a system evolves. According to the authors, such bugs can be detected with quality guidelines. Our empirical study sheds light onto the configuration complexity that developers face when implementing software. Our proposed metrics are easily accessible and have proven useful in practice. For example, Zhang et al. [2013] experimented with different visualizations of our complexity metrics and presented them to the developers of a configurable system in industry (Danfoss SPL). According to the developers, the metrics and visualizations helped to point out complex parts in the system's implementation. Although the developers were aware of some of these implementation problems, the evaluation of the system using our metrics triggered corresponding code refactorings.

In addition to `#ifdef` directives, alternative sources of configuration knowledge, such as configuration systems and build systems, are important for the development of configurable systems (cf. Section 2.1.1). While variability with `#ifdef` directives is limited to a single file—with the exception of importing variability from header files—configuration systems and build systems span across the entire setup of configurable systems. Unfortunately, configuration knowledge at the level of configuration and build systems is usually not very easy to access. Developers of existing systems often rely on home-grown solutions, which makes it difficult to study them in depth. Nevertheless, there is some work on analyzing and measuring configuration knowledge in such systems [Berger, 2013; Tartler, 2013]. However, the main focus is usually to make knowledge of these systems accessible for other purposes, e.g., the detection of configuration errors across different representations of configuration knowledge [Tartler, 2013].

Variable Code Base

There are several case studies about the preprocessor regarding variability implementation. The most comprehensive case study of CPP was conducted by Ernst et al. [2002]. The authors analyzed the preprocessor usage mainly by covering macro definitions using `#define` directives. The analysis comes with a classification of `#define` usage in a set of 26 open-source software systems and a detailed analysis of the pitfalls that arise from `#define` usage. While `#define` directives are also important for tool support, conclusions based on their usage render an incomplete image of CPP, as `#ifdef` usage also affects tool support. Our findings complement the understanding of preprocessor usage for tool support.

In several studies, Favre [1997, 1996, 1995] outlines the importance and general usage of CPP in software development. The author classifies CPP's directives as important abstractions for expressing data structures, for defining software architecture, and for variant management in developing large-scale software systems. Spencer and Collyer [1992] report on experience

when using `#ifdefs` in one software system to implement portable code: despite CPP's simple development model (use of annotations), developers overuse `#ifdef` directives, which may result in code that is hard to understand and to maintain. In contrast to our work, the classification comes without an empirical evaluation that could help to design and implement tool support for configurable systems.

Inspired by language mechanisms and tool approaches for expressing variability in product lines, Kästner et al. [2008, 2009a] discussed variability implementations at a theoretical level. The authors' distinction between coarse-grained and fine-grained extensions provided the inspiration for our granularity metrics. Our empirical analysis contributes to discussions on variability implementations.

Variability Representation

There is a large body of work regarding variability representation, in particular stemming from the development of variability-aware analysis and variability-aware transformation. First of all, the idea of integrating configuration knowledge into a grammar specification to derive variability-aware code representations has been proposed before. Among the first proposers were Platoff et al. [1991], who used variability-aware abstract syntax graphs to represent `#ifdef` variability in C source code. The authors made one restriction about the use of `#define` and `#include` directives: they are only allowed in places at which regular C declarations can occur.

Vittek [2003] and Garrido and Johnson [2003, 2005] developed CPP-aware refactoring tools for C (XREFACTORY and CREFACTORY respectively), which do not have the restriction about `#define` and `#include` usage. To handle both directives, these tools automatically expand macros using textual substitutions. Furthermore, to create variability-aware ASTs, both tools automatically expand undisciplined annotations to disciplined annotations. XREFACTORY uses a brute-force expansion at the level of files (which leads to massive code replication), whereas CREFACTORY uses a very sophisticated expansion mechanism at finer granularity. However, CREFACTORY is tailored to refactoring and based on heuristics.

Baxter and Mehlich [2001] proposed DMS, a source-code transformation system for C and C++. The authors use CPP-aware grammars that are able to capture a subset of possible `#ifdef` annotations for both languages. To resolve interactions of macro substitution and file inclusion with conditional compilation, the authors use specialized heuristics in DMS. Baxter and Mehlich provide anecdotal evidence that their tool can parse 85 % of industrial pre-CPP code and that rewriting undisciplined annotations in 50 000 lines of code can be done in "an afternoon". Similarly, Platoff et al. [1991] reported that undisciplined annotations do not pose a problem when handling C code. Our findings do not support these statements. Using our definition of disciplined annotations, which aligns with the authors' specification of CPP-aware grammars, some systems have thousands of undisciplined annotations. Thus, it is unlikely that, in practice, developers adopt an approach that includes disciplining annotations manually.

The problems with arbitrary preprocessor transformation inspired other researchers to create their own disciplined macro language [Erwig and Walkingshaw, 2011; Leavenworth, 1966;

McCloskey and Brewer, 2005; Weise and Crew, 1993]. Erwig and Walkingshaw [2011] proposed *choice calculus*, a formal language for software variation management. The theoretical foundation of the choice calculus is similar to our definition of disciplined annotations; both approaches allow only the variation of sub-elements in AST-like structures. Considering macro expansion, there has been significant effort to introduce Lisp-style syntax macros that operate on ASTs instead of relying on token substitution. Syntax macros are a disciplined form of macros and are complementary to disciplined annotations for conditional inclusion [Weise and Crew, 1993]. In this context, especially ASTEC [McCloskey and Brewer, 2005] is interesting, because it also covers both syntax macros and conditional inclusion in a disciplined form (it allows annotations only on declarations, statements, and expressions). The use of ASTEC requires a one-time transformation of all CPP directives into the ASTEC macro language. The authors evaluated their approach using four different programs with the result that most **#ifdef** annotations could be transformed automatically. Our work differs from ASTEC and other syntax-macro systems in that we stick with the lexical preprocessor CPP (we only restrict its use) to avoid changing or adapting existing tool support. Additionally, our analysis gives a more comprehensive overview of **#ifdef** annotations, because we analyzed a substantial code base. Unclassified, undisciplined annotations account for 5.8 %, at the maximum, during the evaluation of ASTEC. We found that some software projects contain up to 28.2 % unclassified, undisciplined annotations (for example, the LINUX kernel used for the ASTEC evaluation seems to be a favorable case compared to the other systems in our analysis, presumably due to its coding guidelines mentioned in Section 3.1.2).⁷ Overall, our work confirms the findings in ASTEC, and we strongly agree that disciplined annotations are practical for implementing variable source code.

Similarly, Boucher et al. [2010] proposed a pragmatic tag-and-prune approach for developing configurable systems using a syntactic preprocessor. Developers annotate syntactically complete code blocks with configuration knowledge (feature tags) to explicitly state variability in the source code. The restriction to annotating solely complete code blocks enforces disciplined annotations, which are not prone to syntax errors. The tag-and-prune approach was successfully applied in the development of a library of a file-transmission protocol [Boucher et al., 2010]. However, we believe that abandoning CPP in favor of a different preprocessor will most likely not work in practice, as the huge amount of legacy code has to be transformed prior to the application of the new preprocessor. Additionally, existing tool support (e.g., editor support) has to be adapted in order to support a new approach. Instead of introducing a new preprocessor, we stick with CPP and improve the tool support for it.

Using disciplined annotations, there are two projects that are able to create variability-aware ASTs of any C code with **#ifdef** directives (TYPECHEF [Kästner et al., 2011] and SUPERC [Gazillo and Grimm, 2012]). Both projects use **#ifdef**-enriched grammars that impose constraints on the placement of **#ifdef** directives in the source code, of which a

⁷We assume that the authors classification of imperfect **#ifdef** annotation is comparable to our classification of unclassified annotations, because the authors use an expansion approach in order to align them with variability supported in ASTEC.

variability-aware parser creates variable AST nodes. If annotations in the source code do not adhere to supported annotations, they are automatically expanded until they can be represented as AST nodes, too. We use variability-aware ASTs in Chapter 4 and 5 as a basis for the analysis and transformation of configurable systems.

There are some approaches of refactoring `#ifdef`-based code into alternative representations, such as aspects. However, existing approaches are mostly conceptual or use manual refactorings due to the complexity of parsing pre-CPP code. For example, Adams et al. [2009] analyzed the feasibility of refactoring, but did not actually execute refactorings. Lohmann et al. [2006] refactored `#ifdef` annotations in an operating system’s kernel, but did not automate the refactoring. Kästner et al. [2009a] automated and formalized refactorings from annotated Java code to mixin-style feature modules (which can be adapted to AspectJ as a target language as well), but strictly relied on disciplined annotations.

In general, representing variability with aspects or feature modules does not entail the same problems (in particular, annotation discipline) as preprocessors. Aspects and feature modules represent syntactically complete code artifacts. A dedicated generator/composer composes code artifacts in a predefined manner in order to generate a desired variant [Apel et al., 2013b]. A generated variant is per se again syntactically correct, as no transformation step (during the generation process) introduces any errors. Instead of analyzing the entire code base as with preprocessors, it is sufficient to analyze all code artifacts in isolation to find all syntax errors.

At the same time, since aspects and feature modules are complete code artifacts—their language does not come with concepts for manipulating tokens based on directives—they can be represented with ASTs incorporating variability information as well. To do so, one has to establish a link from the variable implementation, using aspects or feature modules, to the non-variable base implementation. In the case of aspect-oriented programming, the non-variable base implementation provides implicit extension points (joinpoints) at which aspects hook in their implementation. Variability arises from activating aspects or by leaving the base implementation as it is. In a similar way, feature modules introduce new functionality or refine existing ones. The use of a single feature module during the variant-generation process is optional, unless it is not enforced by the system’s configuration knowledge. Hence, code that introduces new functionalities or refines existing ones can be represented in a variable AST representation, too.

We discuss the use of variable code representations for program analysis and program transformation in the following chapters.

3.5 Summary

Developers of configurable systems use CPP to a large extent to implement variability in a system. At the same time, CPP-based implementations have a huge impact on analyzing and transforming configurable systems. To infer valuable information for the development of proper tool support, we stated a set of research questions regarding the realization of configuration knowledge, the implementation of the variable code base, and the variability representation

for analyzing and transforming configurable systems. Based on the research questions, we introduced a set of metrics for measuring the required properties. We used the metrics for an empirical analysis of CPP-preprocessor use in 42 software systems, comprising more than 30 million lines of C code.

First, we found that CPP's mechanisms are frequently used for implementing a system's code base. On average, 21 % of a system's code base is variable, and existing systems provide up to thousands of options for configuration. Nevertheless, the complexity of configuration knowledge (scattering, tangling, and nesting) does not necessarily increase with the size of a configurable systems. For example, in most systems, the average number of nested `#ifdef` directives is close to one (i.e., developers hardly use `#ifdef` nesting). This is promising, because `#ifdef` nesting is one major cause of exponential growth when analyzing or transforming configurable systems. At the same time, scattering and tangling of configuration knowledge suggest that there is a huge potential for sharing (in particular sharing of analysis results).

Second, when looking at `#ifdef` directives in more detail, we observed that most extensions occur at a high level of granularity: developers use `#ifdef` directives mostly to enclose functions or entire blocks, such as `if` statements or `for` loops. Coarse-grained extensions are easier to transform into alternative representations, such as aspects or feature modules. Both representations provide capabilities to introduce or extend function definitions. Nevertheless, applying such transformations depends on a multitude of additional information such as type, control-flow, and data-flow dependencies. Therefore, our granularity measurements can serve only as a rough indicator. Furthermore, we discussed the benefits of transforming `#ifdef` directives into alternative representations. As a basis for discussion, we determined the number of distinct (heterogeneous) and similar (homogeneous) extensions in the code. Homogeneous extensions can benefit from quantification mechanisms that aspect-oriented languages provide. According to our measurements, most extensions are heterogeneous and only 5 % of the extensions would benefit from aspects.

Third, we discussed the creation of a variability representation for CPP-based implementations. For our purposes, we used an extended AST, in which AST elements incorporate configuration knowledge. To create such ASTs, arbitrary, undisciplined annotations need to be transformed into disciplined annotations. When measuring the annotation discipline for our case studies, we observed that 85 % of all `#ifdef` annotations were already disciplined. They respect the underlying source-code structure and enable a one-to-one mapping between source code and the AST. It seems that software engineers are aware of the problems of undisciplined annotations and deliberately limit their use. With our study, we demonstrated that disciplined annotations bear the potential to improve the situation for tool developers who aim at unprocessed source code significantly. This way, tools for analyzing and transforming configurable systems become feasible for a vast amount of legacy C code. Since undisciplined annotations occur in nearly all analyzed programs, we propose to make them disciplined by means of a semi-automatic transformation. We argue that, to take advantage of disciplined annotations, it is feasible to accept certain kinds and a certain amount of code replication in favor of better tool support and better readability of code.

4 Analyzing C Code with Preprocessor Annotations

This chapter shares material with the ESEC/FSE'13 paper „Scalable Analysis of Variable Software“ [Liebig et al., 2013].

In Chapter 3, we found out that many software systems provide a rich set of configuration options that allow users to tailor a software system to a specific application scenario. For example, the LINUX kernel can be configured by means of more than 7000 compile-time configuration options [Sincero et al., 2007], giving rise to possibly billions of variants that can be derived and compiled on demand. While advances in configuration management and generator technology facilitate the development of configurable software systems with myriads of variants, this high degree of configurability is not without cost. How can we analyze these many variants for errors? Unfortunately, traditional analyses look only at individual variants and do not scale in the presence of an exponential number of variants that can typically be derived from a configurable system. For systems such as the LINUX kernel, it is not even possible to derive all variants to analyze them separately, because there are so many—more than the estimated number of atoms in the universe [Tartler et al., 2011].¹ There are two competing approaches for the analysis of configurable systems with a huge number of valid variants: *sampling-based analysis* and *variability-aware analysis* [Thüm et al., 2014]. Both address the fact that an exhaustive analysis of all system variants (brute-force) is usually impossible.

The idea of sampling is to select a representative set of variants (also known as sample set) to be analyzed. The sample set contains valid configurations of the configurable system, and it is determined by means of a sampling heuristics that exploits configuration knowledge to compute a possibly small set for analysis. After a generation step, individual variants that do not contain variability any longer can be analyzed using a traditional, variability-unaware analysis. Sampling-based analysis is still a standard de facto [Thüm et al., 2014]. Many different sampling heuristics were proposed (e.g., [Johansen et al., 2011; Tartler et al., 2012]) and were successfully applied in various scenarios [Oster et al., 2010; Siegmund et al., 2012]. Although analysis time can be reduced significantly, the information obtained is necessarily incomplete, since only a subset of all variants is checked.

¹There were some attempts of determining all valid configurations (also known as model counting [Biere et al., 2009]) of systems of the size of the LINUX kernel using either SAT solvers [Thüm et al., 2009] or BDDs [Mendonça, 2009; Mendonça et al., 2008]. None of them has succeeded so far, since the computational effort exceeded available hardware resources. While SAT solvers perform well when solving individual satisfiability problems, they become intractable for an exponential number of problems (e.g., counting all valid configurations). The critical operation with BDDs is their initialization, for which attempts scale up to 2000 configuration options [Acher et al., 2012].

Recently, researchers have begun to develop a new type of analysis that is variability-aware [Thüm et al., 2014]. The key idea is to not generate and analyze individual variants separately, but to analyze the code base directly before variant generation, incorporating variability and configuration knowledge. In the case of the LINUX kernel, source code is analyzed directly including `#ifdefs`, which is contrary to applying the generator (CPP) to generate the plain C code of individual kernel variants and analyze them in isolation. Variability-aware analysis requires more effort (upfront investment) compared to traditional analysis of a single system, because all local variations need to be considered. However, and this is the key success factor, variability-aware analysis takes advantage of the similarities between variants and avoids analyzing common code over and over again.

Although both analysis approaches have been around for some time, there is no in-depth comparison that reveals a realistic picture of their application in practice. To close this gap, we survey both analysis approaches with respect to three important criteria for analysis tools: (1) analysis time, (2) upfront investment, and (3) error coverage. First, with analysis time we determine the overall performance of an analysis approach for the computation of analysis results. This is a crucial criterion, because an analysis approach is often accepted by developers only when it does not disrupt his/her workflow [Johnson et al., 2013]. Second, with upfront investment we discuss facets of setting up an analysis approach in practice. In particular, we take a look at technical requirements for sampling-based analysis and variability-aware analysis. With this criterion we investigate pros and cons of both analysis approaches and their feasibility in the long run. Third, since sampling-based analysis is per se incomplete, it is interesting to determine the number of errors that sampling heuristics still finds when analyzing configurable systems. If a sampling-based analysis is fast and the sampling heuristics covers most (if not all) errors, it may still be practical to use it in a given application scenario. With these criteria in mind we study the pros and cons of both analysis approaches and are able to make recommendations on choosing one approach over the other in practice.

Before we compare sampling-based analysis and variability-aware analysis with respect to the three criteria in Section 4.2, 4.3, and 4.4, we give a brief introduction to both analysis approaches next.

4.1 Sampling-based Analysis and Variability-aware Analysis

4.1.1 Sampling-based Analysis

Sampling-based analysis of configurable systems stems from the early approaches of testing software [Nie and Leung, 2011]. Due to the sheer size and complexity of real-world systems, a brute-force approach of analyzing all inputs in isolation is usually infeasible. Hence, developers typically analyze only a subset of variants, called the sample set, using off-the-shelf analysis tools. The idea is that, even though we cannot analyze all inputs individually, we can still strive for analyzing a representative sample set in order to be able to draw informed conclusions about

the entire set of inputs (e.g., concerning defect probability). Contrary to the original definition of sampling-based analysis, we look at a special case of sampling-based analysis, in which the inputs are configuration options and not arbitrary input values (e.g., program parameters) of a software system. So the output of a sampling approach is a sample set containing system variants.

The sample set is selected by a *sampling heuristics*, applied either by a domain expert or by an algorithm. Researchers and practitioners proposed different sampling heuristics, some of which require a sophisticated upfront analysis. We selected four heuristics that are common in practice (single configuration, random, code coverage, and pair-wise) and describe them in this section. Although the outcome of sampling-based analysis is inherently incomplete, the coverage criterion of a sampling heuristics allows developers to make reliable statements to some extent, e.g., that all combinations of two configuration options in a system are free of errors. For an overview of other sampling strategies, we refer to a recent survey [Nie and Leung, 2011].

header.h	main.c
<pre> 1 #ifndef A 2 int foo(int a) {...} 3 #else 4 int foo2(int a) {...} 5 #endif 6 #ifdef B 7 int bar(int i, int j) { 8 ... 9 } 10 #endif </pre>	<pre> 11 #include "header.h" 12 int main() { 13 #ifdef A 14 bar(2,3); 15 foo2(3); 16 #endif 17 #ifdef C 18 print("done"); 19 #endif 20 } </pre>

Figure 4.1: C code with preprocessor directives; the header file (left) contains one alternative and one optional definition; the C file (right) uses the definitions of the header file.

Single-configuration Heuristics

The simplest sampling heuristics, called *single configuration*, is to analyze only a single representative variant that enables most, if not all, configuration options of a configurable system. Typically, the variant is manually selected by a domain expert. The strength of this heuristics is that there is only a single variant that needs to be analyzed. Hence, it is fast. By selecting many configuration options, the heuristics covers a large part of the system's code base. However, it cannot cover mutually exclusive code fragments at the same time or intricate interactions specific to individual combinations of configuration options [Garvin and Cohen, 2011]. For the code snippet in Figure 4.1, we can create a configuration that enables all configuration options: {A, B, C}. Since code fragments in Line 2 and 4 are mutually exclusive, a single configuration will cover only one of them, leaving Line 4 uncovered.

According to Dietrich et al. [2012b], in the LINUX development community it is common to analyze only one predefined variant, called *allyesconfig*, with most configuration options

selected. Similarly, many software systems come with a default configuration which satisfies most users and which usually activates many configuration options.

Random Heuristics

A simple approach to select samples is to generate them randomly. For example, in a project with n configuration options, we could make n random, independent decisions (one for each option) of whether to enable them. In projects with dependencies between options, we would discard variants with invalid configurations and keep the remaining variants as our sample. Random sampling is simple and scales up to an arbitrary sample size. The developers of `BUSYBOX` and `LINUX` use random sampling during their development process. In both projects, decisions on configuration options are dependent, as random decisions are only applied to remaining, undecided options after constraints in the configuration system have been evaluated. While this procedure avoids creating invalid configurations, it renders the sampling approach uneven and, hence, contrary to the original definition of this sampling heuristics. Random sampling does not adhere to any specific coverage criterion, and analysis approaches based on it can continue until time or money runs out.

Code-coverage Heuristics

The code-coverage heuristics is inspired by the statement-coverage criterion used in software testing [Zhu et al., 1997]. In contrast to software testing, the code-coverage heuristics aims at variant generation, not code execution [Tartler et al., 2012]. The goal of this heuristics is to select a minimal sample set of variants, such that every lexical fragment of the system's code base is included in, at least, one variant. In contrast to the single-configuration heuristics, code-coverage heuristics covers mutually exclusive code fragments. However, note that including each code fragment at least once does not guarantee that *all* possible combinations of individual code fragments are considered. For the code snippet in Figure 4.1, the two configurations $\{A, B, C\}$ and $\{\}$ (the first selecting all options and the second deselecting them all) would be sufficient to include every code fragment in, at least, one variant. However, it would not help to detect the compilation error when calling `bar` when `A` is selected but `B` is not.

There is an algorithm to compute an optimal solution (a minimal set of variants) by reducing the problem to calculating the chromatic number of a graph, but it is NP-complete and does not scale up to our case studies.² Instead, we resort to the conservatively approximated solution by Tartler et al. [2012], which speeds up the computation of the sample set significantly at the cost of producing a sample set that is possibly larger than necessary.

A subtle problem of this heuristics arises from the issue of how to treat header files. When computing the sample set of two variants in our example, we implicitly assumed that we analyze coverage in the main file and the included header file together. Due to the common practice of including files that include other files themselves, a single `#include` directive in the source

²For more details on the optimal algorithm, confer to <https://github.com/ckaestne/OptimalCoverage/>.

code can bloat the code base of a single file easily by an order of magnitude. According to Kästner et al. [2011], file inclusion is frequently used in the LINUX kernel, in which, on average, 353 header files are included in each C file. In addition, header files often exhibit their own variability which is invisible in the C file without expanding macros. Furthermore, the `#include` directive for a header file may be annotated with an `#ifndef`, so that, for a complete analysis of all header code, sophisticated analysis mechanisms become necessary (e.g., symbolic execution of preprocessor code) [Hu et al., 2000; Latendresse, 2003; Kästner et al., 2011]. The resulting variant explosion can make complete analyses that include header files unpractical or infeasible, even with the approximate solution by Tartler et al. [2012]. Therefore, we distinguish two strategies for code coverage: (1) covering variability only in C files and (2) covering variability in C files and included header files. When analyzing the main file in Figure 4.1, the single configuration $\{A, C\}$ is sufficient to cover all code fragments of the main file. We use both strategies later in our evaluations.

Pair-wise Heuristics

The pair-wise heuristics is motivated by the hypothesis that many faults in software systems are caused by interactions of, at most, two configuration options [Kuhn et al., 2004; Calder and Miller, 2006; Perrouin et al., 2012; Siegmund et al., 2012]. Using pair-wise heuristics, the sample set contains a minimal number of samples that cover all pairs of configuration options, whereby each sample is likely to cover multiple pairs. For the code in Figure 4.1, with three optional and independent configuration options, a pair-wise sample set consists of four configurations: $\{A, B\}$, $\{A, C\}$, $\{B, C\}$, and $\{\}$.

The computation of pair-wise sample sets is not trivial if dependencies (such as A implies B or C) exist in the configuration model; in fact it is NP-complete (similar to the minimum set cover problem) [Johansen et al., 2011]. Hence, existing tools apply different conservative approximations to make the computation possible for large systems with many configuration options. In our experiments, we used SPLCATOOL³ by Johansen et al. [2011]. The computed sample set covers all pair-wise interactions that occur in a given system, but the set is not guaranteed to be minimal.

Summary

The single-configuration heuristics is simple and easy to apply, but covers only a fragment of a system's code base. Code-coverage heuristics and pair-wise heuristics outperform single configuration in covering a system's code base, but the algorithm for the computation of minimal sample sets is NP-complete. To make sample-set computation feasible for large systems, existing tools apply different heuristics to make the computation tractable at the expense of increased sample-set sizes.

³<http://heim.ifi.uio.no/martifag/splcatool/>

4.1.2 Variability-aware Analysis

Traditional analysis techniques do not incorporate configuration-knowledge in the form of `#ifdef` directives, so they are only suitable for the analysis of single system variants. Analyzing system variants separately, the analysis techniques do not exploit similarities between system variants, and in a worst-case scenario, parts of a configurable system (with n configuration options) have to be analyzed 2^n times. In contrast, variability-aware analysis (also known as family-based analysis [Thüm et al., 2014]) takes advantage of the similarities between the variants of a system in order to speed up the analysis process. Although individual variability-aware analyses differ in many details [Thüm et al., 2014], an idea all have in common is to analyze code that is shared by multiple variants only once. To this end, variability-aware analyses do not operate on generated variants, but on the raw code artifacts that still contain configuration knowledge.

There are many proposals for variability-aware analysis in literature, including data-flow analysis [Brabrand et al., 2013; Bodden et al., 2013], deductive verification [Thüm et al., 2012], model checking [Apel et al., 2013d; Classen et al., 2010, 2011; Apel et al., 2011; Lauenroth et al., 2009], parsing [Kästner et al., 2011], and type checking [Apel et al., 2010a; Kästner and Apel, 2008; Kästner et al., 2012b; Thaker et al., 2007]. However, while this work is promising, variability-aware analyses (beyond parsing) have not been applied to large-scale, real-world systems so far. Previous work either concentrated mostly on formal foundations or is of limited practicality (evaluated with academic case studies only). Despite the foundational previous work, it is unclear whether variability-aware analysis scales to large systems, as it considers all variations of a system simultaneously.

It is important to note that the development of a variability-aware analysis requires an upfront investment. Existing analysis techniques and tools have to be made variability-aware. That is, underlying data structures and algorithms need to be modified to incorporate configuration knowledge. Next we describe the data structures for type checking and liveness analysis that we use throughout our comparison of sampling-based analysis and variability-aware analysis. Specifically, we describe their generation and how they are processed by variability-aware algorithms.

All data structures and algorithms described in this section come with corresponding implementations as part of the TYPECHEF project (variability-aware parsing and analysis infrastructure).⁴ While variability-aware ASTs and variability-aware type checking were developed by others (Kästner et al. [2011] and Kästner et al. [2012b]), variability-aware CFGs and variability-aware liveness analysis are contributions of our own.

Variability-aware Abstract Syntax Tree

Many static analyses are performed on ASTs (cf. Section 2.2). As we want to analyze an entire configurable software system, we have to construct an AST that covers all variants of a system,

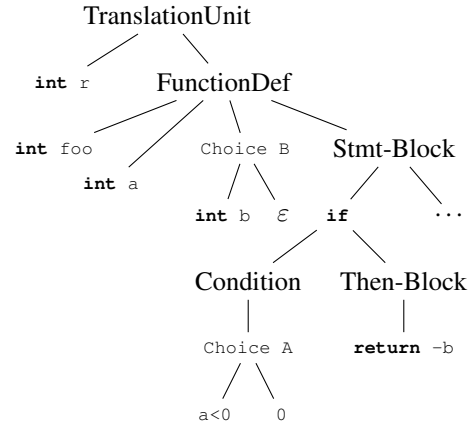
⁴<https://ckaestne.github.io/TypeChef/>


```

1 #ifndef A #define EXPR (a<0)
2 #else #define EXPR 0
3 #endif
4
5 int r;
6 int foo(int a #ifndef B, int b #endif) {
7     if (EXPR) {
8         return -b;
9     }
10    int c = a;
11    if (c) {
12        c += a;
13        #ifndef B c += b; #endif
14    }
15    return c;
16 }

```

(a) Source code with two configuration options that configure several code fragments (Line 1 to 3, 6, and 13); for compactness, we integrated `#ifndef` directives inside single code lines



(b) Excerpt of the corresponding variable AST

Figure 4.2: Running example for variability-aware analysis in C.

including the corresponding configuration knowledge from `#ifndef` annotations in the source code (cf. Section 2.1.1).

Previously, we discussed that the limitation of `#ifndef` variability to disciplined annotations enables a one-to-one mapping between `#ifndef` code fragments in source code and a variable representation in an AST (cf. Section 3.3). The resulting AST representation is similar to a standard AST, but configuration knowledge is attached to express compile-time variability. To do so, we extend our standard AST with variable nodes (or *Choice* nodes) that represent `#ifndef` variability of the source code in our AST representation. Similar to ambiguity nodes in Generalized LR parse forests [Tomita, 1984], choice nodes express the choice between two or more (using nested choice nodes) alternative subtrees. Explored formally in the choice calculus [Erwig and Walkingshaw, 2011], choice nodes are a general vehicle for expressing variability in data structures [Walkingshaw, 2013]. For example, *Choice*(A, $a < 0$, 0) (cf. Figure 4.2b) expresses the alternative of two expressions $a < 0$ and 0, controlled by configuration option A. The *Choice* node directly represents the configurable expression (including the `#ifndef` annotation A) in our running example (cf. Figure 4.2a; `#ifndef`s in Lines 1 to 3 and their use in Line 7). One alternative of a choice may be empty (cf. Figure 4.2b; ε), which makes the other, in fact, optional. In principle, we could use a single *Choice* node on top of the AST with one large branch per variant; but a configurable AST is more compact, because it shares parts that are common to multiple variants (e.g., in Figure 4.2b, we store only a single node for the declaration of `r` and a single node for the function name `foo`, which are shared by all variants).

To create variable ASTs for further processing, we use a variability-aware parsing framework

called TYPECHEF. TYPECHEF's parser incorporates `#ifdefs` during the parsing process and is able to handle undisciplined annotations automatically. To this end, the parser expands undisciplined annotations until they become disciplined and can be represented in the AST. In contrast to our definition of disciplined annotations mentioned previously (cf. Section 3.1.2), TYPECHEF's parser allows variability at a lower level of granularity. That is, besides declarations and statements, annotations at the level of function parameters, function specifiers, or even subexpressions and substatements are possible and find themselves represented as `Choice` nodes in the variable AST.

For more information about the parsing process, we refer to Kästner et al. [2011].

Variability-aware Type Checking

A classic type-checking algorithm for C traverses the AST, collects declarations in a symbol table, and attempts to assign proper types to all expressions (`getType: Map[Name, Type] → Expr → Type`). In principle, a variability-aware type checker works similarly, but covers all variants. For our comparison, we rely on the implementation of a variability-aware type checker on top of the TYPECHEF parsing infrastructure by Kästner et al. [2012b]. The type checker incorporates configuration knowledge in each of the three following steps.

First, a symbol (variable, function, etc.) may only be declared in some variants or may even have alternative types in different variants. To capture differently annotated variables, the symbol table is extended such that a symbol is no longer mapped to a single type (similar to the proposal of Aversano et al. [2002]), but to a conditional type (a choice of types or ε ; `CST = Map[Name, Choice[Type]]`). Table 4.1 illustrates a possible encoding of a conditional symbol table for our example, including the symbols' valid name binding in the program (scope). A symbol which is declared in all variants does not need `Choice` nodes (e.g., `r`). However, if a symbol is declared in a subtree of the AST that is only reachable given a certain presence condition, a presence condition is included in its type. Similarly, a symbol with different types can be declared in different variants. In our running example, the function `foo` has two alternative types (`int → int → int` vs `int → int`), depending on whether `B` is selected. Similar to variables and functions, type information for structures and enumerations in C has to be extended, too.

Second, during expression typing, the type checker assigns a variable type (choice of types) to each expression (`getType: CST → Expr → Choice[Type]`), by retrieving type information for symbols from the lookup table. In the context of configurable systems, the lookup may return a variable type. For example, when verifying that the condition of an `if` statement has a scalar type, the type checker checks whether all alternative choices of the variable type are scalar. If the check fails only for some alternative results, a type error is reported that points to a subset of variants, as characterized by a corresponding presence condition. Similarly, an assignment is only valid if the expected (variable) type is compatible with the provided (variable) type in *all* variants. Therein, an operation on two variable types can, in the worst case, result in the Cartesian product of the types in either case of the choice, resulting in a variable type with many alternatives. All other type rules are basically

Symbol	(Conditional) Type	Scope
r	int	0
foo	Choice(B, int → int → int , int → int)	0
a	int	1
b	Choice(B, int , ϵ)	1

Table 4.1: Conditional symbol table (CST) at Line 6 of our running example in Figure 4.2.

implemented along the same lines. In our running example, the type checker would report a type error in Line 8, because the symbol `b` cannot be resolved in variants without configuration option `B` activated (cf. Table 4.1).

Third, the type checker uses the configuration model of a system (if available) to filter all type errors that occur only in invalid variants. To this end, it simply checks whether the presence condition of each type error is satisfiable when conjoined with the configuration model (checked with a standard SAT solver). If so the type checker issues an error including the presence condition, otherwise it does not.

For more information about type checking in the presences of variability, we refer to Kästner et al. [2012b].

Variability-aware Control-flow Graphs

Most data-flow analyses require a CFG (cf. Section 2.2) that represents all possible execution paths of a program. Nodes of the CFG correspond to instructions in the AST, such as assignments and function calls; edges correspond to possible successor instructions according to the execution semantics of the programming language. A CFG is a conservative static approximation of the program’s actual behavior.

As with type checking, we need to make CFGs variable to cover all system variants. To create a CFG for a single program, we need to compute the successors of each node ($\text{succ} : \text{Node} \rightarrow \text{List}[\text{Node}]$). In the presence of variability, the successors of a node may diverge in different variants, so we need a variability-aware successor function that returns different successor lists for different variants. To encode such a function we use `Choice` nodes to represent varying successor elements. So for the successor function, we encode the result as $\text{succ} : \text{Node} \rightarrow \text{Choice}[\text{List}[\text{Node}]]$, or equivalently but with more sharing $\text{succ} : \text{Node} \rightarrow \text{List}[\text{Choice}[\text{Node}]]$. The higher sharing of analysis results in $\text{List}[\text{Choice}[\text{Node}]]$ arises from the fact that successor elements common to all variants are stored only once, instead of storing them multiple times in differently annotated successor lists ($\text{Choice}[\text{List}[\text{Node}]]$). Using the result of this successor function, we can determine presence conditions for all possible successors and add them as annotations to edges in the variability-aware CFG.

We illustrate variability-aware CFGs by means of the optional statement in Line 12 of our running example in Figure 4.2. In Figure 4.3, we show an excerpt of the corresponding variability-aware CFG (node numbers refer to line numbers in Figure 4.2a). The successor of the instruction `c += a` in Line 12 depends on the configuration option `B`: if `B` is selected, statement `c += b` in Line 13 is the direct successor; if `B` is not selected, `return c` in Line 15 is the (only) successor. Technically, we add further nodes to the result list of the successor function, until the conditions of the outgoing edges cover all possible variants, in which the source node is present (checked with a SAT solver or BDDs).

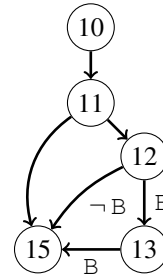


Figure 4.3: Excerpt of the variability-aware CFG of the running example in Figure 4.2.

Alternatively, we could drop presence conditions on edges and express variants of the control flow with `if` statements of C. Figure 4.4 illustrates the difference between these alternatives. The variable `a` is initialized with zero and incremented/decremented afterwards depending on the configuration options `A` and `B`, respectively. When both configuration options are mutually exclusive, the control-flow is limited to the increment and the decrement of `a`, because there is no control-flow path in between. By contrast, if we use `if` statements, we lose precision, because a normal CFG does not evaluate the `if` condition, but conservatively approximates the control flow by reporting both alternative branches as possible successor statements. For our example in Figure 4.4b there is a possible control-flow path from the increment in Line 3 to the decrement in Line 6. Such sound but incomplete approximation is standard practice to make static analysis tractable or decidable. However, using `if` statements, we lose precision for compile-time configurability, since the evaluation of configuration options is pushed to run time. Furthermore, we have only propositional formulae to decide between execution branches, which makes computations decidable and comparatively cheap. So we decided in favor of presence conditions on edges, which is in line with prior work on CFG in variable Java programs [Brabrand et al., 2012; Bodden et al., 2013].

We implemented the first variant of variability-aware CFGs (using presence conditions on control-flow edges) on top of TYPECHEF’s parsing infrastructure (cf. Section 4.1.2).

Variability-aware Liveness Analysis

Liveness analysis (or live-variable analysis) is a classic data-flow analysis for the computation of variables that are live (that may be read before being written again) for a given statement. A variable is live if there is a path from a definition to a use without any re-definition of the variable in between. The result of the analysis can be used, for example, to conservatively detect dead code. That is, if a variable is not live after its definition (with respect to the control flow), it can be eliminated. In real-world systems, warnings about dead code that occurs only in specific

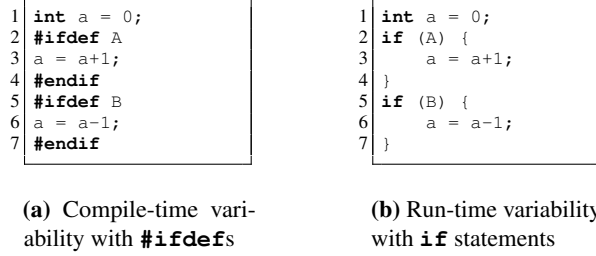


Figure 4.4: CFG representation with compile-time and run-time variability.

variants are important for maintainers; corresponding problems are regularly reported as bugs.⁵ Therefore, we want to incorporate variability in the results of the liveness analysis, too.

Liveness analysis is a fix-point computation of sets of variables that are live at (live_{in}) and live after (live_{out}) a given control-flow statement s of a function (cf. Equations 4.1 and 4.2). Both functions are repeatedly recomputed for all control-flow statements of a function until their outcomes stabilize at a fixed point, i.e., the result sets for all control-flow statements do not change anymore. Live variables are determined using the two functions gen (cf. Equation 4.3) and kill (cf. Equation 4.4), which compute all variables read and all variables written to, respectively. Liveness is a backward analysis, because analysis results are passed on successor elements (using succ in Equation 4.2) to its predecessors. While in traditional liveness analysis all functions return sets of variables, in variability-aware liveness analysis both return sets that may vary (a choice of sets or a set with optional entries), depending on the variability in the input representation.⁶ The results of live_{in} and live_{out} are variable as well, and the signatures of both functions change from $\text{Node} \rightarrow \text{Set}[\text{Id}]$ to $\text{Node} \rightarrow \text{Set}[\text{Choice}[\text{Id}]]$, where Id represents the identifier of a live variable.

$$\text{live}_{\text{in}}(s) = \text{gen}(s) \cup (\text{live}_{\text{out}}(s) \setminus \text{kill}(s)) \quad (4.1)$$

$$\text{live}_{\text{out}}(s) = \bigcup_{p \in \text{succ}(s)} \text{live}_{\text{in}}(p) \quad (4.2)$$

$$\text{gen}(y \leftarrow f(x_1, \dots, x_n)) = \{x_1, \dots, x_n\} \quad (4.3)$$

$$\text{kill}(y \leftarrow f(x_1, \dots, x_n)) = \{y\} \quad (4.4)$$

⁵e.g., https://bugzilla.kernel.org/show_bug.cgi?id=1664.

⁶Internally, we use formula maps ($\text{Map}[\text{Id}, \text{PresenceCondition}]$) that associate each value with a presence condition [Walkingshaw et al., 2014]. Formula maps generalize sets ($\text{Set}[\text{Id}]$), in which they annotate each set element with a condition, which controls the presence or absence of the element. In our case, the condition is a propositional formula constructed from configuration options. The diff and union operations of formula maps (cf. Equations 4.1) are straightforward extensions including BDD and SAT operations. In the standard case (no variability), PresenceCondition is equal to type Boolean .

Line	Uses	Defines	In	Out
10	{a}	{c}	{a, b _B }	{a, b _B , c}
11	{c}	{}	{a, b _B , c}	{a, b _B , c}
12	{a, c}	{c}	{a, b _B , c}	{b _B , c}
13	{b _B , c _B }	{c _B }	{b _B , c _B }	{c _B }
15	{c}	{}	{c}	{}

Table 4.2: Liveness-computation result of our running example in Figure 4.2.

In Table 4.2, we show the results of variability-aware liveness analysis for our running example. We show the result of each equation as a set of variables including their presence condition as subscript. For example, only variable `c` is live in the `return` statement in Line 15. Considering the control flow from Line 10 to 13 ($10 \rightarrow 11 \rightarrow 12 \rightarrow_B 13$), in the declaration statement in Line 10 the variable `a` is live, whereas `b` is only live if `B` is selected.

Our discussions on variability-aware liveness analysis are an important contribution to the development of variability-aware static analysis, and we implemented variability-aware liveness analysis as an extension of TYPECHEF’s analysis infrastructure on top of variability-aware CFGs (cf. Section 4.1.2).

4.2 Analysis Time

To evaluate the performance of sampling-based analysis and variability-aware analysis, we compare their analysis times in two full-fledged scenarios: type checking and liveness analysis. Both types of analysis are frequently used and are seamlessly integrated in modern Integrated Development Environments (IDEs) for on-the-fly error detection and coding assistance (e.g., auto-completion). Analysis times play an important role here, because results should be obtained quickly to give developers fast feedback and not to disrupt their workflow. So far it has been unknown whether sampling-based analysis and variability-aware analysis are feasible for the analysis of configurable systems and whether they scale in practice. To this end, we analyzed three real-world, large-scale systems, which increases external validity substantially compared to previous work that mostly concentrated on formal foundations, made limiting assumptions, or relied on comparatively small and academic case studies (cf. Section 4.6). We used state-of-the-art sampling heuristics (*single conf*, *code coverage* with and without headers, and *pair-wise*), as introduced in Section 4.1.1.

4.2.1 Empirical Study

Hypotheses and Research Questions

Based on the goals and properties of variability-aware and sampling-based analysis, we state two hypotheses and two research questions:

1. *Variability-aware vs single conf*: Analyzing all variants simultaneously using variability-aware analysis is most likely slower than analyzing a single variant that covers most configuration options. This is because the variable program representation covering all variants is larger than the program representation of any single variant, including the largest possible variant:

H₁ The execution times of variability-aware type checking and liveness analysis are *larger* than the corresponding times of analyzing the variants derived by single-configuration sampling.

2. *Variability-aware vs pair-wise*: While previous work showed that pair-wise sampling is a reasonable approximation for the analysis of all variants [Lochau et al., 2012], it can still generate quite large sample sets, as it considers the configuration space of the entire system in question [Johansen et al., 2012]. Hence, we expect variability-aware analysis to outperform pair-wise sampling:

H₂ The execution times of variability-aware type checking and liveness analysis are *smaller* than the corresponding times of analyzing the variants derived by pair-wise sampling.

3. *Variability-aware vs code coverage*: With respect to the comparison of variability-aware analysis and code-coverage sampling, we cannot make any informed guesses. Code-coverage sampling generates sample sets depending on the use of configuration options in the analyzed C files. As we do not know details of the code, we cannot predict how many variants will be generated and how large they will be. Hence, we pose a research question instead. Specifically, the influence of configuration options that occur in header files is unknown and, therefore, we look at two different variants of code coverage: one with header files and one without.

RQ₁ How do the execution times of variability-aware type checking and liveness analysis compare to the times for analyzing the variants derived by code-coverage sampling (with and without header files)?

4. *Scalability*: Finally, we pose the general question of the scalability regarding variability-aware analysis.

RQ₂ Does variability-aware analysis scale to systems with thousands of configuration options?

The motivation for questioning scalability is that variability-aware analysis reasons about variability by solving SAT problems or using BDDs during analysis. Generally, SAT is NP-complete, but previous work suggests that the problems that arise in variability-aware analyses are typically tractable by state-of-the-art SAT solvers [Mendonça et al., 2009] and BDDs [Czarnecki and Wąsowski, 2007], and that caching can be an effective optimization [Apel et al., 2010a].

Subject Systems

To evaluate our hypotheses and to answer our research questions, we selected three subject systems. We chose publicly available systems (for replicability) of substantial size, which are actively maintained by a community of developers and used in real-world scenarios. The systems consist of a substantial, variable code base, in which variability is implemented using CPP, and provide at least an informal configuration model that describes configuration options and their valid combinations [Liebig et al., 2010]. To facilitate the use of configuration models in these systems, we employed several tools for configuration-model extraction [Berger et al., 2010b; Tartler et al., 2011] and build-system analysis [Berger et al., 2010a].

- The `BUSYBOX` tool suite reimplements a subset of standard Unix tools for resource-constrained systems. With 792 configuration options, it is highly configurable, resulting in 1.26×10^{159} valid system variants. Most of the options refer to independent and optional subsystems and the configuration model in conjunctive normal form has 993 clauses. We used `BUSYBOX` version 1.18.5 (522 C files and 191 615 lines of source code).
- The `LINUX kernel` (x86 architecture, version 2.6.33.3) is an operating-system kernel with billions of installations worldwide, from high-end servers to mobile phones. With 6918 configuration options, it is highly configurable. In Chapter 3, we identified the `LINUX kernel` as one of the largest and (with respect to configuration options) most complex publicly available configurable software systems [Liebig et al., 2010]. It has 7691 source-code files with 6.5 million lines of code. Note that already the configuration model of `LINUX` is of substantial size: the corresponding extracted formula in conjunctive normal form has over 60 000 variables and nearly 300 000 clauses; a typical satisfiability check requires half a second on a standard computer. The sheer size of `LINUX`' configuration model rendered a computation of all valid system variants impossible. When trying to determine the number of variants, we ran out of memory on our well-equipped evaluation machine with 64 GB of RAM.
- The cryptographic library `OPENSSL` implements different protocols for secure Internet communication. `OPENSSL` can be tailored to many different platforms, and it provides a set of 589 configuration options, with which 6.5×10^{175} valid system variants can be generated. We analyzed `OPENSSL` version 1.0.1c with 733 files and 233 450 lines of code. Since `OPENSSL` does not come with a formal configuration model like `BUSYBOX` or

LINUX, we inferred a configuration model based on a manual analysis. Specifically, we analyzed syntax and type errors and used their corresponding erroneous configurations to build a configuration model. The resulting model has 15 clauses.

Experimental Setup

We used TYPECHEF as the underlying parsing framework. As explained in Section 4.1.2, TYPECHEF generates a variable AST per file, in which `Choice` nodes represent optional and alternative code. We reused the variability-aware type checking implementation of Kästner et al. [2012b] and implemented a variability-aware liveness analysis on top of variable ASTs. Both implementations are part of the TYPECHEF project.

To avoid bias due to different analysis implementations, we used our infrastructure for variability-aware analysis also for the sampling-based analysis. To this end, we generated individual variants (ASTs without configuration options) based on the sampling heuristics. We created an AST for a given configuration by pruning all irrelevant branches of the variable AST, so that no `Choice` nodes remained. As there were no configuration options in the remaining AST, the analyses never split, and there was no additional effort due to SAT solving, because the only possible presence condition was *true*.

As our liveness analysis is intra-procedural, it would be possible and more efficient to apply sampling to individual functions and not to files, as done by Brabrand et al. for Java product lines [2012; 2013]. Unfortunately, preprocessor macros in C rule out this strategy, as we cannot even parse functions individually without running the preprocessor first or without performing full variability-aware parsing. In our running example in Figure 4.2, we would not even notice that the function `foo` is affected by `A`, because variability arises from configurable macros defined outside the function. Configurable macros defined in header files are very common in C code [Kästner et al., 2011].

During liveness analysis, we did not take the configuration model of the analyzed systems into account. This is because we performed the data-flow analysis without posing an analysis question (e.g., “Which code is dead?”). In particular, during liveness analysis, our algorithms performed SAT checks without taking the configuration model into account, similar to the inter-procedural analysis for Java product lines by Bodden et al. [2013]. This way, the computation is faster and still complete, though false positives may occur. False positives can be eliminated easily after a subsequent refinement step (i.e., using the configuration model in SAT checks), so that only valid execution paths are taken into account. This strategy is commonly used in model checking [Clarke et al., 2003] to make the analysis of a complex system feasible in the first place.

In Figure 4.5, we illustrate our experimental setup. Depending on the sampling heuristics, one or multiple configurations are checked. For each file of the three subject systems, we measured the time spent in type checking and liveness analysis, each using the variability-aware approach and the three sampling heuristics (the latter consisting of multiple internal runs). In total, four to five analyses were conducted per subject system: a variability-aware one + three to four sampling-based (with and without header files for code coverage) ones.

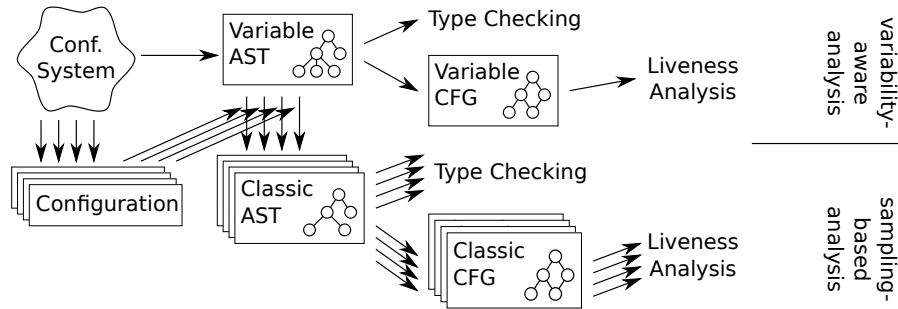


Figure 4.5: Experimental setup.

We ran all measurements on LINUX machines (Ubuntu 14.04) with Intel(R) Xeon(R) CPU E5-2690 v2 (with 3.0 GHz) and 64 GB RAM. We configured the Java JVM with up to 4 GB RAM for memory allocation.

Results

Table 4.3 shows the measurement results for each analysis and subject system together with the speedups, showing the relative performance improvement of variability-aware analysis compared to sampling-based analyses. We report performance measurements as total times for the entire analysis. Execution was sequential, though parallelization would be possible in all cases, as each file was analyzed in isolation. Furthermore, we illustrate the distribution of analysis times for individual files using notched box-plots on a logarithmic scale. Figures 4.6, 4.7, and 4.8 show the plots for BUSYBOX, LINUX, and OPENSSEL, respectively. In the plots, we highlight the median (over all files) of the variability-aware analysis with a vertical line, to simplify comparison with the medians of the sampling-based analysis. Apart from that, we provide the number of analyzed configurations for each of the sampling-based analyses (below the name of the analysis, ‘configurations per file’ or in short ‘c.p.f.’). Single-configuration heuristics requires the same number of variants for each file (because it is based on global knowledge of the configuration model only), whereas code coverage and pair-wise heuristics⁷ require different numbers of variants in different files, which we provide in terms of mean \pm standard deviation. We evaluated all research hypotheses using paired, one-sided t-tests at a confidence level of 95 % (including the Bonferonni correction to oppose the problem of multiple hypothesis testing [Salkind, 2007]).

⁷In addition to the given configuration model, the build system of LINUX defines presence conditions for individual files (cf. Section 2.1.1). So, as for LINUX, each file has a configuration model of its own. Computing pair-wise sample sets for each file was infeasible, as a single pair-wise sample-set computation took more than 20 h on our machine. Therefore, we use the global configuration model for the computation of pair-wise sample sets. As the generated sample may contain configurations that do not satisfy a file’s presence condition, the overall number of analyzed configurations for a file decreases.

System	Analysis approach	Type checking		Liveness analysis	
		time in s	speedup	time in s	speedup
BUSYBOX	Single configuration	673	0.92	274	0.88
	Code coverage NH	979	1.34	430	1.39
	Code coverage	3 850	5.25	2 760	8.90
	Pair-wise	1 540	2.10	661	2.13
	Variability-aware	733		310	
LINUX	Single configuration	25 300	0.39	14 000	0.53
	Code coverage NH	72 100	1.10	48 300	1.84
	Pair-wise	891 000	13.60	817 000	31.22
	Variability-aware	65 500		26 200	
OPENSSL	Single configuration	513	0.61	246	0.64
	Code coverage NH	952	1.13	448	1.18
	Code coverage	2 380	2.81	1 610	4.24
	Pair-wise	2 160	2.55	1 360	3.58
	Variability-aware	846		380	

Table 4.3: Total times for analyzing the subject systems with each approach (time in seconds, with three significant digits) and speedups, showing the relative performance improvement of variability-aware analysis compared to sampling-based analyses; a speedup lower than 1.0 reflects a slowdown.

In all subject systems and for both type checking and liveness analysis, variability-aware analysis is slower than single-configuration sampling (H_1 ; statistically significant) and faster than pair-wise sampling (H_2 ; statistically significant). The results regarding code-coverage sampling (RQ_1) are along the same lines. Type checking and liveness analysis with variability-aware analysis are faster than code-coverage sampling in *BUSYBOX*, *LINUX*, and *OPENSSL* (statistically significant). We observe that code coverage without header files (NH) is often faster than with header files, and sometimes it even outperforms single-configuration sampling. This is because many `#ifdefs` occur in header files, something that is neglected in code-coverage sampling NH. Single-configuration sampling considers variability in header files; consequently, it may result in a system variant that is more expensive to analyze than analyzing the sampling configurations determined with code-coverage NH.

To put our results into perspective, let us illustrate in Figure 4.9 the trade-off between variability-aware analysis and sampling-based analysis by means of the example of liveness analysis for *BUSYBOX*. The x-axis shows the average number of variants sampled for the respective sampling heuristics. The y-axis shows the average analysis times for the variants analyzed. Clearly, the more variants are analyzed, the longer the analysis takes, which illustrates the trade-off between analysis coverage and analysis time. The interesting spot is the break-even point, at which variability-aware analysis becomes faster than sampling (dashed line).

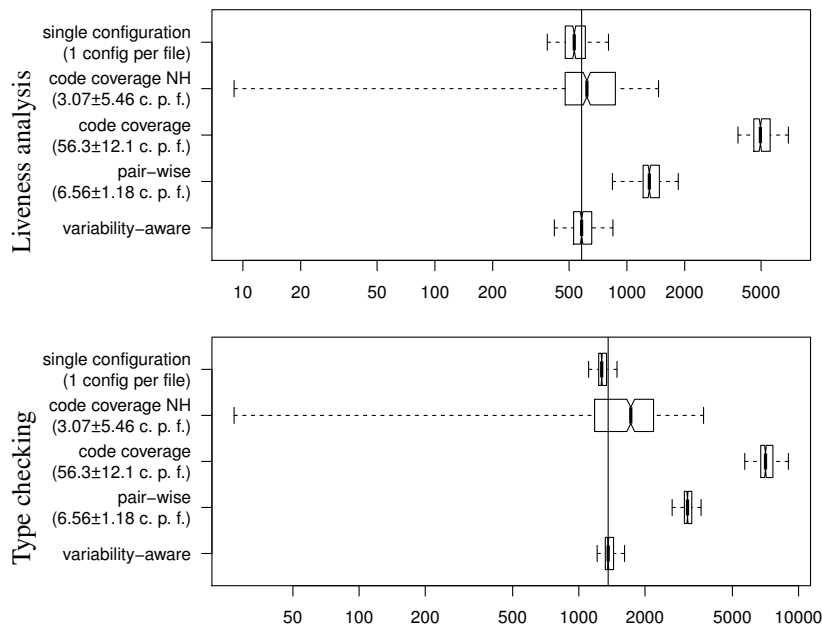


Figure 4.6: Distribution of analysis times for BUSYBOX (times in milliseconds; logarithmic scale).

But recall that, even though sampling is faster below this line, it comes at the price of losing information due to a limited analysis coverage.

Discussion and Lessons learned

Our experiments support hypotheses H_1 and H_2 : in all three subject systems, variability-aware analysis is faster than sampling-based analysis when using the pair-wise heuristics, but slower than using the single-configuration heuristics. With respect to research question RQ_1 , there is no clear picture. The performance of code-coverage sampling depends on the variability implementations in the respective files; the number of sampled variants and the performance results considerably differ between files inside each subject system (cf. Figures 4.6, 4.7, and 4.8). So, performance of the code-coverage heuristics is hard to predict and strongly depends on certain implementation patterns. Nevertheless, in our experiments variability-aware analysis is faster than code-coverage sampling.

A further observation is that variability-aware analysis results in higher speedups in liveness analysis than in type checking. This can be explained by the fact that liveness analysis is intra-procedural, whereas type checking considers entire compilation units. Exploring the performance of variability-aware inter-procedural analysis of large scale systems is an

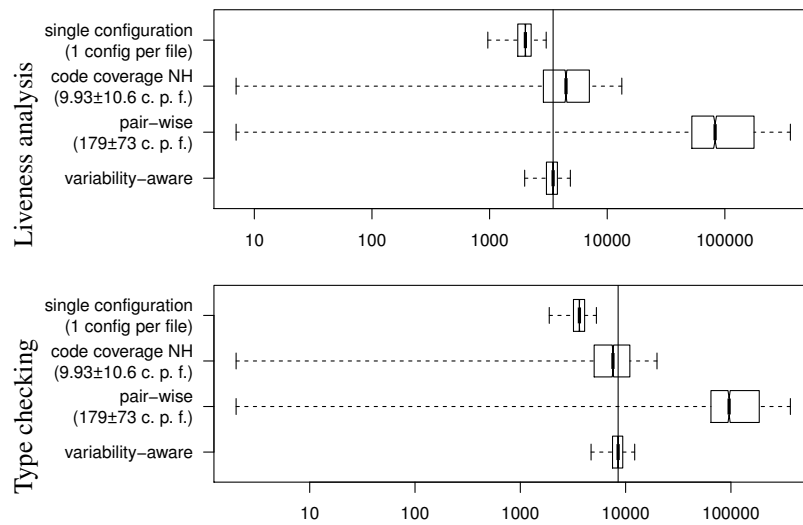


Figure 4.7: Distribution of analysis times for LINUX (times in milliseconds; logarithmic scale).

interesting avenue for further work. Additionally, for liveness analysis, we did not employ a system's configuration model in SAT checks. This, most likely, decreases the time for computing analysis results.

The experimental results for BUSYBOX, LINUX, and OPENSLL demonstrate that variability-aware analysis is in the range of the execution times of sampling with multiple samples (code coverage and pair-wise). Thus, with regard to question RQ₂, we conclude that variability-aware analysis is practical for large-scale systems. An important finding is that the additional effort induced by solving SAT problems during the analysis is not a bottleneck, not even for large systems such as the LINUX kernel. Overall, variability-aware type checking (compared to single configuration) in BUSYBOX takes as much time as checking 2 variants (3 variants in LINUX and 2 variants in OPENSLL). For liveness analysis, the break-even point is at 2 variants for all three subject systems. That is, if a sampling heuristics (including random sampling) produces a sampling set larger than that, variability-aware analysis is faster and, in addition, complete. All values are very low compared to the number of possible variants of the respective system. We can conclude that a complete analysis is possible at the cost of an incomplete sampling heuristics.

Threats to Validity

- A threat to internal validity is that our implementations of variability-aware type checking and liveness analysis support ISO/IEC C, but not all GNU C extensions used in the subject systems (especially LINUX). Our analyses simply ignore corresponding code constructs. Also, due to the textual and verbose nature of the C standard, the implementation does

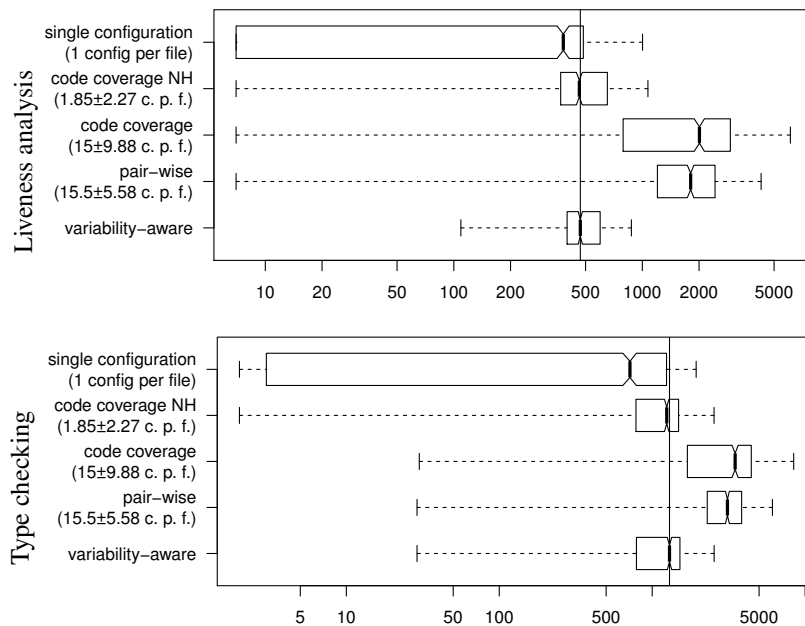


Figure 4.8: Distribution of analysis times for OPENSLL (times in milliseconds; logarithmic scale).

not entirely align with the behavior of the GNU C compiler. Due to these technical problems, we excluded 1 file of BUSYBOX and 1 file of OPENSLL from our study. All numbers presented in this work have been obtained after excluding the problematic files. Still, the comparatively large numbers of 521 files for BUSYBOX, 7691 files for LINUX, and 732 files for OPENSLL let us conclude that the approach is practical and that our evaluation is representative.

- The variants generated by the sampling heuristics represent only a small subset of possible variants (which is the idea of sampling). But for pair-wise sampling, it may happen that some variants of a file are very similar, as the difference in the respective variant configurations affect the content of a file to a minor or no extent. However, we argue that our conclusions are still valid, as this is in the nature of the sampling heuristics, and all heuristics we used are common in practice.
- A (common) threat to external validity is that we considered only three subject systems. We argue that this threat is largely compensated by their size, the fact that many different developers and companies contributed to the development of these systems, and that all these systems are widely used in practice.

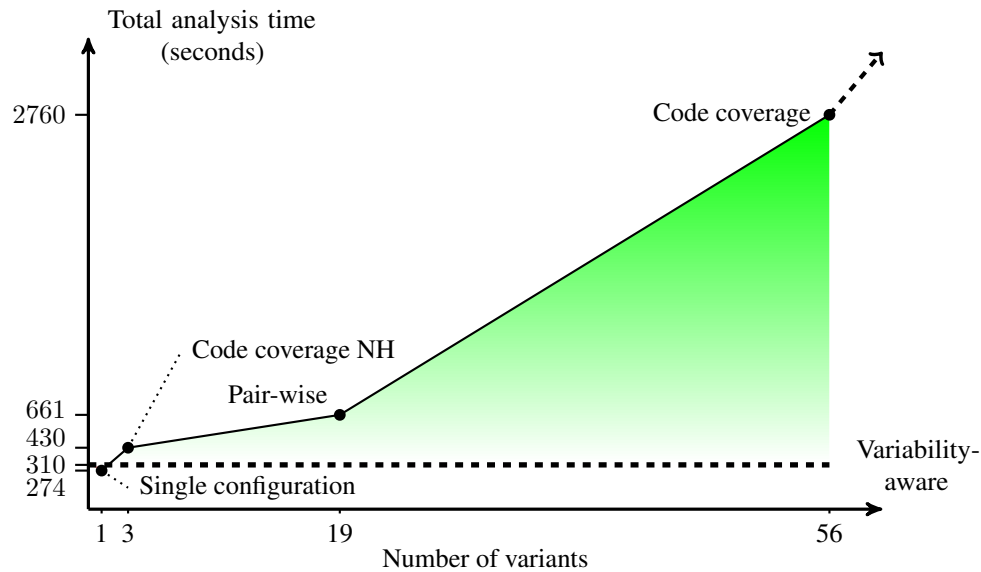


Figure 4.9: Number of variants vs analysis time for liveness analysis of BUSYBOX.

Summary

In our experiments, we found that the performance of variability-aware analysis scales to large software systems, such as the LINUX kernel, and even outperforms some of the sampling heuristics, while still being complete.

Next, we take a look at the technical setup of sampling-based analysis and variability-aware analysis in more detail.

4.3 Upfront Investment

Both analysis approaches require a certain upfront investment before they can be pursued in a real setting. For sampling, we need to compute the sample set of valid configurations of a system before the analysis takes place. Using a configuration and the configurable system's generator, we can quickly generate individual variants of a system on demand, which can be processed afterwards using off-the-shelf analysis tools. Optimal solutions for sampling heuristics such as pair-wise sampling are known, and lookup tables containing optimal sample sets are available [Kuhn et al., 2013]. Employing such sample sets is easy, because there is no computational effort, and analysis approaches simply search lookup tables for their respective solution. However, such sets can only be applied when no configuration constraints exist. When constraints exist, they pose a huge challenge on the development of scalable techniques for the

generation of sample sets. In fact, sample-set generation is far from trivial and still subject to active research [Johansen et al., 2012; Tartler et al., 2012; Henard et al., 2014; Perrouin et al., 2010]. This is mainly because some sampling strategies rely on algorithms that are NP-complete, and sophisticated heuristics are necessary to speed up the sampling process. Nevertheless, sampling-based analysis benefits from the fact that it facilitates reuse of existing analysis techniques.

In contrast to sampling-based analysis, we cannot reuse existing analysis techniques with variability-aware analysis. Its application usually requires a reimplementaion. This includes adaptations of data structures and algorithms to empower analysis techniques to work with a variable code base. This approach has been pursued to adapt existing type-checking, model-checking, and testing techniques to configurable systems [Lauenroth et al., 2009; Classen et al., 2010; Apel et al., 2010a; Kästner et al., 2012a,c,b]. However, so far, there is no standard procedure for developing variability-aware analysis, and most variability-aware analysis tools have been developed from scratch. Unfortunately, redeveloping variability-aware analysis takes a considerable amount of time. The development of the infrastructure necessary for variability-aware control-flow and liveness analysis took us several months, and others reported similar times for developing variability-aware analysis techniques [Kästner et al., 2011, 2012b]. At the same time, variability-aware analyses usually lack the maturity of existing analyses and tools, some of which are developed for ages.

In this section, we summarize our experience with the generation of sample sets using different sampling heuristics, and we provide insights into the development of variability-aware analysis. In particular, we outline the essence of variability-aware analysis and develop a variability-aware data-flow framework, which simplifies the development of static analyses significantly.

4.3.1 Experience with Sampling

We expected that contemporary sampling tools can quickly compute representative sample sets. However, for pair-wise sampling, the time for the sample-set computation takes up to several hours (e.g., >20 h for LINUX; cf. Table 4.4). This high computation time was the reason why we did not generate sample sets for each file of LINUX individually. As most LINUX files exhibit build-system variability (cf. Section 2.1.1), each file may have a different configuration model. Computing individual sampling sets for LINUX files is impossible, as the high sample-set computation times rule out this form of treatment. Furthermore, due to the size of the input problem, sampling heuristics often generate a considerably high number of configurations, which we could not analyze in reasonable time (e.g., code coverage for LINUX).

The single-configuration heuristics worked well. LINUX has a commonly used configuration, called *allyesconfig*, which is maintained by the community and frequently used for analysis purposes.⁸ For BUSYBOX and OPENSLL, we created single configurations by selecting as many configuration options as possible.

⁸<http://kernel.org/doc/Documentation/kbuild/kconfig.txt>

Random sampling has proved problematic. Both `BUSYBOX` and `LINUX` have configuration models with many constraints. In 1 000 000 random configurations, there was not a single one that satisfied all constraints of the input configuration model. Random sampling was only a possibility for `OPENSSL`, which has a comparatively sparse configuration model ($\sim 3\%$ of randomly generated configurations were valid). `BUSYBOX` and `LINUX` developers actually use a skewed form of random sampling (`RANDCONFIG`), in which random values are selected one by one for every configuration option whose activation/deactivation is not enforced by constraints of other options. This approach strongly depends on the ordering of configuration options and violates a developer's intuition about random selection.

In contrast to all other heuristics, heuristics based on code coverage need to investigate every file individually (and optionally all included header files). We reimplemented the conservative algorithm of Tartler et al. [2012] for this task in two variants: one including header files and one excluding them. When headers are included and macros are considered, an analysis must process several megabytes of source code per C file [Kästner et al., 2011]. Code-coverage heuristics often generates a huge number of valid configurations for a given input. We tried to improve the situation by filtering configurations that are covered by others and, thus, overall reduce the size of the number of configurations in question. However, this reduction requires many SAT checks, is computationally very expensive, and cannot be conducted for `LINUX` in a reasonable amount of time. Hence, we left the code-coverage sample sets untouched and omitted the analysis of `LINUX`.

Although the sampling algorithms contain many sophisticated heuristics for speeding up the sample-set computation, we observe that the computation still takes a considerable amount of time (cf. Table 4.4). This is because of the large number of SAT problems, which need to be solved. Solving these problems is a necessity for the sample-set generation. By contrast, in variability-aware analysis, SAT problems are only solved on demand (by determining the satisfiability of computed analysis results in a particular configuration space). For practical applications, this can lead to a higher number of SAT problems for sampling-based analysis than for variability-aware analysis and consequently to higher analysis times. For example, the times for generating sample sets with code-coverage sampling heuristics exceeded the times for performing variability-aware liveness analysis in `BUSYBOX`. In `LINUX` the generation even exceeded the analysis times for type checking and liveness analysis.

Although pair-wise sampling is frequently used and there are several proposals for efficient sample-set computation [Johansen et al., 2012; Henard et al., 2014; Perrouin et al., 2010], we found only one research tool (`SPLCATOOL`)⁹ that was able to compute complete sample sets of pair-wise configurations for a given configuration model at the scale of the `LINUX` kernel. `SPLCATOOL` performed reasonably well for `BUSYBOX` and `OPENSSL`, but `LINUX`' larger configuration space made the sample-set computation very expensive. In `BUSYBOX` and `LINUX` the computation time exceeded the times for performing variability-aware type checking and liveness analysis. We could have relaxed the pair-wise heuristics by not covering all pair-wise configurations of a system. A corresponding approach (including a tool implementation;

⁹<http://heim.ifi.uio.no/martifag/splcatool/>

PLEDGE)¹⁰ was proposed by Henard et al. [2014]. This sampling approach provides a parameter for limiting the sample set to a predefined size. Internally, the tool uses a search-based approach, which stops when the limit is reached. Although the application of PLEDGE would speed up the generation time of the sample set significantly, we did not use this approach. PLEDGE’s sample set is incomplete with respect to all possible pair-wise configurations and, hence, an erroneous configuration may not be part of the computed sample set. Similarly, Kowal et al. [2013] proposed a filtering algorithm to reduce the number of variants of pair-wise sample sets generated with the ICPL algorithm (SPLCATOOL uses ICPL). The algorithm precomputes a set of ‘interesting’ configuration options that serves as an input for pair-wise sampling heuristics. The sampling algorithm takes the lower number of configuration options into account and possibly generates a smaller sample set in less time. Both approaches [Henard et al., 2014; Kowal et al., 2013] reduce the number of system variants that need to be analyzed significantly. However, general conclusions, such as the analyzed system being free of errors that may occur in any pair-wise combination of configuration options, are impossible.

System	Single configuration	Code coverage NH	Code coverage	Pair-wise
BUSYBOX	92	180	440	322
LINUX	9 360	81 200	n/a	1 280 000
OPENSLL	63	127	203	127

Table 4.4: Times for sample-set computation (time in seconds, with three significant digits).

Of course, we could increase a sample set’s size to cover more system variants by applying a different sampling heuristics. For instance, we could apply t -wise sampling, the generalization of pair-wise sampling (where $t = 2$). The parameter t represents the number of configuration options that are considered together. So, in the case of $t = 1$ (or feature-wise), each configuration option is considered in, at least, one configuration of the sample set; for $t = 3$ (or triple-wise), all combinations of three options are considered, and so forth. Including a system’s configuration knowledge, the size of the sample set usually increases with a higher value of t . So, for BUSYBOX, the respective sizes of the sample sets for 1-wise and 2-wise are 4 and 31.¹¹ For BUSYBOX, the times (in seconds) for creating 1-wise and 2-wise sample sets are 1 and 16. At the same time, the computational effort increases for computing sample sets with a higher t -value and was impossible to master for most systems we used (BUSYBOX and LINUX). So covering more configurations by computing 3-wise sample sets was impossible for larger configurable systems in a reasonable amount of time. For example, for BUSYBOX, we terminated the generation after two hours due to the lack of progress. Furthermore, the elapsed time exceeded analysis times for variability-aware type checking (733 s) and liveness analysis (310 s) considerably.

¹⁰<http://research.henard.net/SPL/>

¹¹Computed with SPLCATOOL.

4.3.2 Patterns of Variability-aware Analyses

The main success factor of variability-aware analysis over variant-based analysis techniques, such as brute-force or sampling, is the sharing of analysis results between similar variants of the configurable system. Researchers proposed different strategies to maximize sharing of (intermediate) analysis results and to reason about configuration knowledge efficiently [Thüm et al., 2014]. For example, TYPECHEF’s variability-aware parser preserves sharing by creating annotated AST nodes that multiple system variants have in common. Code without `#ifdef` directives is represented only once, because it is shared by all variants. The parser creates `Choice` nodes to represent local variations in the input source code only if necessary. Throughout our analyses, we preserve sharing of results as far as possible, by using compact variability-aware data structures and by adapting analysis algorithms accordingly. Specifically, three patterns emerged that maximize sharing: *late splitting*, *early joining*, and *local variability representation* as illustrated in Figure 4.10. The key observation is: keep variability local.

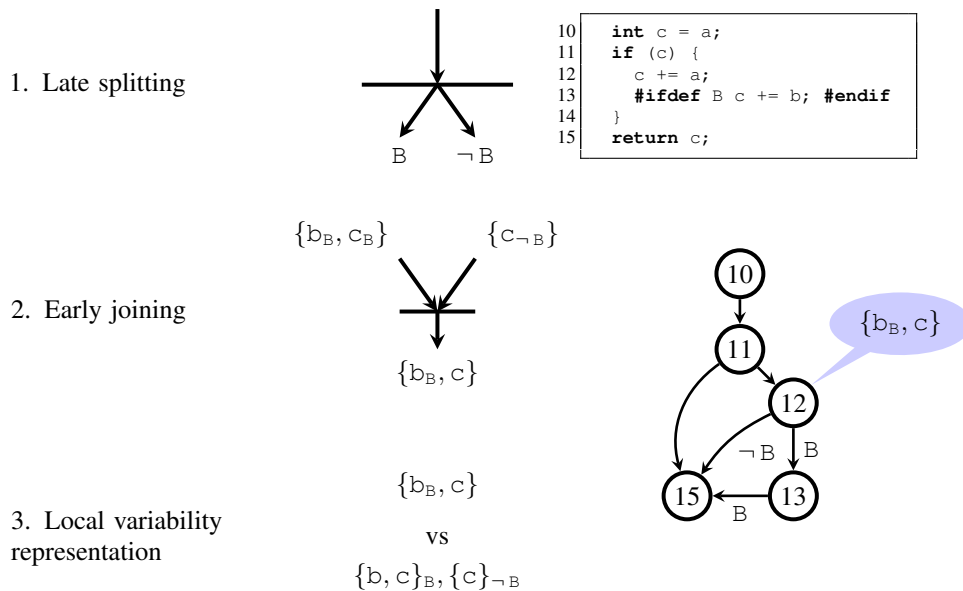


Figure 4.10: Patterns of variability-aware analysis illustrated using liveness analysis for an excerpt of our running example in Figure 4.2, including the variability-aware CFG.

First, *late splitting* means that we perform the analysis without variability until we encounter it. For example, in liveness computation, the analysis splits at differently annotated control-flow statements. Thus, liveness computation for `c += a` in our example splits because of configuration option `B`, resulting in two intermediate results: (1) following path `12 →B 13` and returning $\{b_B, c_B\}$ and (2) following path `12 →¬B 15` and returning $\{c_{\neg B}\}$.

Second, *early joining* attempts to *join* intermediate results early, often as early as possible. For example, if we have two result sets from different control-flow paths, we join them using the set union operation \cup . If we have a variable with different annotations (c_B and $c_{\neg B}$), we can simply join them to c (presence condition `true`) for further processing. Hence, even if the control flow ‘explodes’ due to different annotations, we can often join the results to a compact representation. This way, variability passes from parts of the AST into other parts, if and only if variability *actually makes a difference* in the internal representations of variables or other forms of analysis results. Additionally, we need to consider only combinations of configuration options that occur in different parts of the AST if they actually generate different (intermediate) results when combined; otherwise the results remain orthogonal.

Third, *local variability representation* aims at keeping variability local in intermediate results. For example, instead of distinguishing between different result sets in our liveness analysis, we have only one set with annotated variables that are live at the point given. Technically, we use the type `Set [Choice [Id]]` instead of the type `Choice [Set [Id]]` to achieve this locality. Liveness computation of `c += a` in our example returns a set $\{b_B, c\}$, in which the identifiers are variable, instead of the entire set. This preserves sharing between variants. Even after conditional control-flow, we store each identifier only once. Alternatively, we could store the results in two sets $\{b, c\}_B$ and $\{c\}_{\neg B}$ (`Choice [Set [Id]]`). This, however, is inefficient, because the variable c occurs twice. Although this is not a problem in our simple example, in practice, variables may occur in many different variants, causing redundant storage of variables. This is a serious problem, because an analysis tool may run out of main memory. Furthermore, maintaining all variants is computationally more expensive than maintaining our compact representation. For a detailed discussion about variability-aware data structures and their trade-offs, we refer the interested reader to Walkingshaw et al. [2014].

The three patterns of late splitting, early joining, and local variability representation are applicable to any kind of variability-aware analysis. Although not always made explicit, these patterns can also be observed in other variability-aware analyses [Apel et al., 2013d; Kästner et al., 2012c, 2011; Brabrand et al., 2012]. Next, we use these patterns to simplify the development of static analyses.

4.3.3 Variability-aware Intra-procedural Data-flow Framework

To ease the development of static analyses in general, we revisit the three patterns of late splitting, early joining, and local variability representation and develop a variability-aware framework for intra-procedural data-flow analysis. The framework considerably simplifies the development of different static analyses, which we use for detecting programming errors later (cf. Section 4.4).

In Section 4.1.2, we introduced variability-aware liveness analysis. Liveness analysis is an instance of general intra-procedural frameworks for solving data-flow equations, called MONOTONE FRAMEWORKS [Nielson et al., 1999]. MONOTONE FRAMEWORKS provide general abstractions for exploiting similarities between different data-flow analyses. Apart from liveness, other instances of the frameworks are *reaching definitions*, *available expressions*, and *very*

busy expressions, all of which can be used to solve many practical analysis problems, including program optimization (e.g., [Aho et al., 2006]) and programming-error detection (e.g., [Cherem et al., 2007]). For example, *double free* ensures that dynamically allocated memory is freed only once. A pointer variable passed to the function `free` to deallocate previously allocated memory should not be passed to another `free` call without having assigned other dynamically allocated memory to the pointer variable. We can define *double free* as a reaching-definition problem. Reaching definition determines all definitions (assignments of variables) that reach a given point in the CFG without being overridden. Using reaching definitions, we track all pointers of freed variables until they get a reassignment with different dynamically allocated memory. If there is an attempt to free an already freed pointer variable, we issue an error. Similar properties can also be stated for other static analyses, such as uninitialized variables, freeing of statically allocated memory, and checking of return values of standard library functions for errors.

Before we outline our extensions of the framework to make it variability-aware, we describe the framework's central concepts and explain differences between our formal description and the original one proposed by Nielson et al. [1999].

General Definition

A central element of MONOTONE FRAMEWORKS is the *lattice* L , which represents data-flow properties/information and the *combination operator* $\sqcup : \mathcal{P}(L) \rightarrow L$ (with $\sqcup : L \times L \rightarrow L$) which joins intermediate results from different paths (F) in the control-flow representation. An analysis computed by MONOTONE can be either *forward* or *backward* with respect to the CFG. In forward analysis, analysis results are forwarded to successor elements, and, in backward analysis, to predecessor elements (flow and flow^R, respectively). For both kinds of analyses we use the standard operations `pred` and `succ` for traversing the CFG. The data-flow computation relies on the two functions `Analysiso` and `Analysis•`, that use each other and represent a fix-point computation to determine the analysis results of a given control-flow statement l . During the fix-point computation, the framework computes for each CFG element a transfer function f_l , which simulates actual program behavior with respect to a given analysis question, i.e., a given instance of MONOTONE.

For reasons of simplification, MONOTONE FRAMEWORKS are formulated for a simple imperative language called WHILE [Nielson et al., 1999]. WHILE was designed to illustrate different aspects of program analysis. It shares common characteristics of imperative languages such as statement sequences, control structures (**if-then-else**), and repetitive computations (**while**-loop). However, it neglects many abstractions and programming constructs present in C, e.g., pointers, user-defined types, and functions. The motivation behind using WHILE was to teach students common patterns of data-flow analysis at textbook level. The basic principles of the analysis patterns remain the same (general framework for data-flow equations and forwarding of data-flow properties) for a fully-fledged language such as C, but have to be adapted to some extent. Specifically, the computation of single data-flow properties has to be adapted, since C provides more programming constructs. Furthermore, since C is a more expressive programming language compared to WHILE (e.g., pointer, arrays, and user-defined

types), our analysis framework faces several limitations (cf. Section 4.5).

Equations 4.5-4.7 summarize the definitions of MONOTONE FRAMEWORKS. Using them, we can define liveness analysis with: $\sqcup = \cup$, $\sqcap = \cap$, $\perp = \emptyset$, $\iota = \emptyset$, $F = \text{flow}^R$, and gen / kill as defined in Section 4.1.2.

$$\text{Analysis}_o(l) = \sqcup \{ \text{Analysis}_\bullet(l') \mid l' \in F(l) \} \sqcup \iota_E^1 \quad (4.5)$$

$$\text{where } \iota_E^1 = \begin{cases} \iota & \text{if } l \in E \\ \perp & \text{if } l \notin E \end{cases}$$

$$\text{Analysis}_\bullet(l) = f_l(\text{Analysis}_o(l)) \quad (4.6)$$

$$f_l(l) = (l \setminus \text{kill}(l)) \cup \text{gen}(l) \quad (4.7)$$

\sqcup is \cap or \cup (and \sqcap is \cup or \cap)

F is either flow or flow^R

E denotes the entry or the exit, respectively, of the intra-procedural CFG

ι_E^1 represents the initial (ι) and final (\perp) value of the lattice

f_l is the transfer function associated with l

Differences

The original definition of MONOTONE FRAMEWORKS operates on WHILE programs, which are labeled representations of program code. Each control-flow element in a WHILE program has a unique label, which is used for identification during program analysis.¹² Operations in MONOTONE FRAMEWORKS make frequent use of several auxiliary functions (blocks, labels, init, and final) that provide means to determine control-flow for program elements. In our definition of MONOTONE, we omit labels and work on control-flow elements of the CFG directly, because we can distinguish AST elements in our infrastructure internally.

Toward a Variability-aware Data-flow Framework

To enrich the variability-unaware MONOTONE FRAMEWORKS with variability, we need to adapt the lattice L at first. The lattice stores analysis elements in a generic set ($\text{Set } [T]$), which represents the type of elements we want to track in a specific instance of MONOTONE. For liveness analysis, the generic type T is Id (cf. Section 4.1.2); for available expressions, T is Expr . To represent analysis results with local variations, we change the variability representation from $\text{Set } [T]$ to $\text{Set } [\text{Choice } [T]]$ to incorporate variability. Similar to the definition of liveness analysis, this representation ensures a redundancy-free storage of analysis

¹²Although WHILE labels are part of the syntax, they are only used for identification and do not represent any control-flow-like execution semantics. Therefore, they should not to be confused with labels in C, which provide jump targets for control-transfer statements (`goto`).

results, because each element of \mathbb{T} is stored only once, including the result's presence condition that controls its presence in different system variants [Walkingshaw et al., 2014]. Traversing the CFG with flow and flow^R ensures late splitting. The analysis splits if and only if variability makes a difference in the input representation. We merge intermediate results directly using the early joining pattern in Equations 4.5 and 4.7. As the respective joining operations in the framework (\sqcup , \sqcap , \cup , and \setminus) also need to handle presence conditions, we apply BDD and SAT operations (\neg , \wedge , and \vee) accordingly.

Summary

The upfront investment for both analysis approaches is different. For sampling, the generation of sample sets is expensive, because the underlying algorithms are NP-complete, and, as a result, sample-set computation takes a considerable amount of time of the entire analysis.

A variability-aware analysis is often developed from scratch. Fortunately, developing a variability-aware analysis is straightforward, because the support for variability in analysis algorithms is orthogonal to the analysis algorithm in question, and variability-aware data structures and algorithms to handle variability seamlessly integrate into analysis approaches.

After having discussed the performance and the upfront investment of sampling-based and variability-aware analysis, we continue with the comparison of sampling-based analysis with variability-aware analysis when detecting programming errors in a real setting.

4.4 Error Detection

With our third criterion (error detection), we investigate the performance of sampling-based analysis to detect programming errors in the presence of variability. In particular, we employ different, variability-aware static analyses to detect serious programming errors in all system variants and investigate whether sampling-based analysis would find these errors, too. As described in Section 4.1.1, sampling-based analysis uses heuristics to reduce the number of variants that are going to be analyzed with a traditional analysis or of-the-shelf analysis tools. Although this analysis approach is necessarily incomplete, sampling-based analysis may still be sufficient to analyze configurable systems for two reasons.

First, for some systems, the number of configurations, although very high, is not required by end users. Consequently it is sufficient to concentrate on desired system variants. For example, HP's printer firmware Owen has more than 2000 configuration options implemented with the C preprocessor [Refstrup, 2009; Pearse and Oman, 1997]. Nevertheless, only 100 variants are generated and compiled frequently, including a static analysis which the compiler incorporates to detect programming errors [Refstrup, 2009]. The desired configuration space is so small that sampling-based analysis still scales in practice, although many, possibly redundant computations have to be made.

Second, there are studies that indicate that sampling heuristics are sufficient to cover most errors that occur in a configurable system [Kuhn et al., 2004; Steffens et al., 2012]. Kuhn et al.

[2004] showed in an empirical study with different t -wise sampling heuristics that using sample sets of 1-wise, 2-wise, and 3-wise sampling are sufficient to find an average of 50, 70, and 95 % of all programming errors, respectively. Steffens et al. [2012] applied pair-wise testing in an industrial case study (Danfoss Automation Drive) with 432 valid system configurations. Using pair-wise sampling, the authors created a sample set with 57 configurations, with which they could find 97.5 % of the system's errors. Both studies show that, depending on the sampling approach, a large number of programming errors can still be detected with sampling. Nevertheless, the studies also show that the sampling approaches used never reached an error coverage of 100 %. A serious threat to both studies is that they are very specific regarding their given application scenario. Thus, it is impossible to draw conclusions for real-world, large-scale configurable systems.

Variability-aware analysis to detect programming errors is beneficial, because for each error, the analysis exhibits only one message that precisely describes the problem including a configuration constraint. The constraint represents the valid configuration space, in which the programming error occurs, and it can be used to derive a valid system variant. By contrast, a sample set may contain (if at all) multiple configurations, in which a programming error is triggered, and clustering techniques may be necessary to infer distinct programming errors, and to narrow down an error to the minimal configuration that causes it.

To evaluate the effectiveness of sampling approaches for the detection of programming errors in `#ifdef`-based, configurable systems, we develop a set of static analyses based on our variability-aware CFG (cf. Section 4.1.2) and on our variability-aware implementation of MONOTONE FRAMEWORKS (cf. Section 4.3.3). Based on them, we determine the error coverage for all sampling heuristics that we used previously (cf. Section 4.1.1).

4.4.1 Evaluation

For the error-coverage evaluation, we developed a set of eight analyses, capturing certain properties that help to detect programming errors in configurable systems. These analyses enable the detection of (serious) programming errors that can be exploited by attackers or that lead to undefined/unexpected program behavior as specified by the C standard.

We do not aim at implementing state-of-the-art program-analysis techniques that are both efficient with respect to analysis time and precise with respect to analysis results. This is because existing analysis tools for error detection have been developed for years to be fast and to report as few incorrect error warnings as possible. We did not have the resources to improve our analyses to the level of common analysis tools, so the outcome of our analysis is likely to contain many false positives and false negatives, some of which we confirmed manually. False positives and false negatives occur for two reasons.

First, all of our analyses are only intra-procedural. That is, analysis properties are only computed on a per-function basis. This limitation means that programming errors that cut across multiple functions, e.g., freeing the same memory location in different functions, cannot be detected (*double-free* detection), or using variables without initialization code, which are assigned a proper value in a different function (*uninitialized-variables* detection).

Second, we did not integrate a pointer-alias analysis in our algorithms. Pointer-alias analysis attempts to determine run-time values of pointers (memory locations), an information which is central to many static analyses [Hind, 2001]. Statically determining a pointer alias is undecidable in general [Landi, 1992; Ramalingam, 1994], so existing approaches make conservative assumptions about pointer accesses for modifications and references [Hind, 2001]. In our analysis implementations, we did not include any of the proposed algorithms, which leads to decreased analysis precision and is likely to produce a lot of false positives (i.e., assuming two pointer alias, which they do not), and false negatives (i.e., neglecting references of different pointers to the same memory location). We discuss the benefit of integrating a pointer-alias analysis in our algorithms in Section 4.5. Nevertheless, our analyses respect the basics of control-flow and data-flow computations for C and, consequently, provide a conservative approximation of programming errors in configurable systems.

Next, we describe the eight different variability-aware analyses that we use for comparing the error coverage of different sampling heuristics (cf. Section 4.1.1):

Dead Store. Assignments to local variables that are not used in subsequent program code are called *dead stores*. Although dead stores do not particularly threaten program security, they are usually the result of a logical programming error and should be removed. A dead store is a particular instance of *dead code*, the existence of unnecessary code that leads to an increase in program size and an increase in program run-time. Our example in Figure 4.11 contains a dead store in Line 2. The assignment to the variable `a` with a value of 2 is overridden in Line 4 without being used in the meantime. To detect dead stores, we use our variability-aware liveness analysis (cf. Section 4.1.2) and determine whether assignments to variables „live out“, i.e., whether variables are read in subsequent program code. If a variable is not used in subsequent code, we issue an error.

```

1 void foo() {
2   int a = 2; // dead store
3   int b = 2;
4   a = 3;
5 }

```

Figure 4.11: Example of a dead store.

Error Handling. High-level constructs for exception or error handling are not available in C. As a result, a common approach is to encode the corresponding information (e.g., error code) in the form of a return value. For example, for dynamically allocating memory, programmers use the standard library function `malloc` in Figure 4.12. Given the size of the desired memory chunk, the function returns either the reserved memory as a pointer or zero (`NULL` or `(void*) 0`) as return code to indicate that the memory allocation failed. To avoid undefined program behavior, the return value of a call to `malloc` should be checked for this error code before it is used. In our example (cf. Figure 4.12), if the configuration option `DEBUG`

```

1 void foo() {
2   int* p = malloc(
3     15*sizeof(int));
4
5   #ifdef DEBUG
6     *p = 1;
7   #endif
8   if (p != NULL) { ... }
9 }

```

Figure 4.12: Example of a missing error handling.

is selected, the pointer `p` is dereferenced before having been checked for the error-return code of `malloc` properly. The C standard defines this error-encoding approach for many standard library functions, including functions for opening and closing files, for formatting input strings, and many more. To determine missing error-code checks, our analysis tracks the results of 31 standard-library calls with a variant of reaching-definition analysis. In particular, the analysis computes whether variables assigned with the result of standard-library calls reach in control structures, checking them for predefined error codes.

Double Free. In C, the programmer is responsible for the management of dynamically allocated memory. To this end, the C standard defines a group of standard-library functions for memory allocation (e.g., `malloc`) and memory deallocation (e.g., `free`). To avoid memory leaks, unneeded memory should be freed. Dynamically allocated memory should be freed only once, since freeing memory multiple times leads to undefined behavior, which can be exploited by attackers. Our example in Figure 4.13 contains a double-free error. If the configuration option `A` is not selected, the allocated memory in Line 2 is freed twice (in Lines 7 and 9), because the variable `a` in Line 5, which changes the binding of variables with the name `a`, is not available. To identify double-free errors, we use a variant of reaching definitions. More concretely, our analysis determines whether a pointer variable passed as an argument to the function `free` is passed to another `free` call without any reassignments of memory.

```

1 int foo() {
2     int* a = malloc(2);
3     if (a) {
4         #ifdef A
5             int* a = malloc(3);
6         #endif
7         free(a);
8     }
9     free(a); // double free
10    return 0;
11 }

```

Figure 4.13: Example of a double free.

Freeing of Static Memory. Freeing memory that was not allocated dynamically using memory-management functions can result in serious errors, e.g., heap corruption or abnormal program termination. For this reason, only pointer variables that contain memory previously allocated with memory-management functions, such as `malloc` or `realloc`, should be freed with `free`. Our example in Figure 4.14 contains such a freeing static-memory error. The variable `s` in Line 2 is assigned only to dynamically allocated memory in Line 4 in the context of an `if`-branch. As the initialization of `s` with static memory in Line 6 is part of a valid CFG path, passing the variable to `free` in Line 8 is a freeing-of-static-memory error. Our analysis tracks pointer variables that are not initialized with dynamic memory, and it determines whether they are passed to memory-management functions.

```

1 int foo(int l) {
2     char *s;
3     if (l == 2) {
4         s = (char *)malloc(12);
5     } else {
6         s = "usage: ... ";
7     }
8     free(s); // freeing st. mem.
9     return 0;
10 }

```

Figure 4.14: Example of freeing a variable, which was allocated statically.

Uninitialized Variables. Variables that are not initialized hold arbitrary values. The use of these variables can lead to serious issues, such as unexpected or undefined program behavior. Therefore, programmers have to guarantee a correct initialization of variables before their first use. In our example in Figure 4.15, the variable `a` in Line 2 is not initialized but passed as an argument to the function `bar` in Line 4. Once again, we use a variant of reaching definitions for the computation of uninitialized variables. In particular, we determine whether uninitialized variables are used before being initialized. We neglect pointer variables in this analysis, because just like the other seven analyses, it is only intra-procedural, and pointer variables are often passed to other functions for initialization.

```

1 void foo() {
2   int a;
3   ...
4   bar(a); // uninit. var.
5 }

```

Figure 4.15: Example of an uninitialized variable being used.

In addition to the five data-flow analyses discussed so far, we also implemented three analyses that solely work on variability-aware CFGs (cf. Section 4.1.2):

Terminate case Blocks with break Statements. The conditional statement `switch` consists of an expression, several `case` labels, and an optional `default` label. Each label is followed by a series of statements and should be ended using a `break` statement (by convention), so that control flow jumps beyond the `switch`. The `break` statement is optional. Hence, control flow can *fall through* the next `case` in a `switch` statement executing further statements of the `switch` body. Omitting the `break` statement may lead to unintended control flows and should be avoided. Figure 4.16 shows a `switch` statement with two `case` blocks. If the configuration option `A` is selected, `case 1` contains code without a corresponding `break`, for which our analysis issues an error.

```

1 void foo(int a) {
2   switch (a) {
3     case 1:
4     #ifdef A
5       a = 2;
6     #endif
7     case 0: a = 1;
8       break;
9   }
10 }

```

Figure 4.16: Example of a `case` block without a terminating `break` statement.

Control Flow in Non-void Functions. According to the C standard, the control flow of all functions with a non-`void` return type should execute a `return` statement. Using the return value of a function without executing a `return` leads to undefined behavior. Figure 4.17 shows an example of such an error. The function `foo` with return type `int` contains a control-flow path without executing the `return` statement in Line 3. If the condition in Line 2 is not satisfied, the function returns to its call site with an incorrect return value. The result of `foo` is used in `bar`, which may lead to undefined behavior. Our analysis checks whether there is a path in the CFG of a

```

1 int foo(int x) {
2   if (x > 0)
3     return 1;
4 }
5
6 void bar(int y) {
7   if (foo(y+2)) ...
8 }

```

Figure 4.17: Example of a non-`void` function with a missing `return` statement.

function with a non-**void** return type that does not end in a **return** statement.

Dangling switch Code. Depending on the condition's value, control flow in **switch** statements jumps to the body of the **switch**, to one of several **case** labels, to the optional **default** label, or it jumps beyond the **switch** statement. A **switch** body may contain any sequence of statements including declarations of variables. If a programmer places code before the first **case** label, the code is never executed, i.e., it dangles in the **switch** statement. Depending on the kind of code (e.g., initialization of variables), dangling switch code may result in unexpected or undefined behavior. Figure 4.18 shows an example of dangling **switch** code. If the configuration option **A** is not selected, the initialization and the increment of variable **b** (Lines 6 and 7) dangle from the **switch** statement. Our analysis computes CFGs for **switch** bodies and checks whether, apart from declarations without initialization code, any control-flow statements occur that are not guarded in any configuration by **case** or **default** labels.

```
1 void f(int a) {  
2     switch (a) {  
3 #ifdef A  
4     case 0:  
5 #endif  
6     int b;  
7     b++;  
8     case 1:  
9     default: a+3;  
10 }  
11 }
```

Figure 4.18: Example of dangling **switch** code.

All eight static analyses are variability-aware and, as such, determine programming errors in all system variants that can possibly be derived from a configurable system. So we can use their outcome to determine the effectiveness of sampling heuristics with respect to error coverage. We describe the process of error-coverage determination and our setup for this experiment in the following.

4.4.2 Experiment Setup

A sampling set, determined by a sampling heuristics, covers only a subset of the entire configuration space of a configurable system. To determine the error coverage of sampling approaches, we run all eight variability-aware static analyses on the variable code base. If a programming error occurs, our analyses issue a warning, including the position of the error in the source code and a constraint. The constraint represents the configuration space, in which the programming error occurs. For all errors and for each sampling heuristics, we determine whether, at least, one configuration in a sample set satisfies the error constraint. If so, the sampling approach covers the error; otherwise, it does not. For this experiment, we reused the sampling heuristics introduced previously (cf. Section 4.1.1) and measured their error coverage for the same subject systems (BUSYBOX, LINUX, and OPENSLL). We used the same setup as before for all measurements (cf. Section 4.2.1).

4.4.3 Results and Discussion

Table 4.5 shows the number of programming errors for each of the eight static analyses with respect to the three subject systems. Along with the total number of errors found by each analysis, we also show the percentage of covered errors for each sampling heuristics and for each programming error.

For the evaluation, we use all issued errors including false positives as well false negatives of our analyses without any (manual) filtering. As static analysis problems are often undecidable, our analyses are unsound and usually require manual inspections to confirm or reject identified programming errors. So the total number of errors serves as a representative of the effort for developers to review identified errors. Furthermore, even though the actual number of programming errors is most likely to be lower, it still correlates with the number of issued errors by our analyses. Hence, our analysis results are still valuable for the discussion on error detection.

System	Errors	EH	DS	DF	UV	NF	TC	CF	DC	Σ
BUSYBOX	Total	1 312	416	194	4 158	440	405	158	0	7 083
	Single configuration	85.21	79.57	84.02	66.11	91.14	80.00	89.87	100.00	73.81
	Code coverage (NH)	99.92	100.00	100.00	99.49	100.00	100.00	99.37	100.00	99.68
	Code coverage	99.92	100.00	100.00	99.52	100.00	100.00	100.00	100.00	99.70
	Pair-wise	88.87	96.15	88.14	84.27	90.91	90.12	88.61	100.00	86.77
LINUX	Total	1 247	428 550	3 387	970 037	7 233	957 252	9 015	45	2 376 766
	Single configuration	85.49	54.81	88.40	53.91	89.27	34.34	31.92	0.00	46.28
	Code coverage (NH)	100.00	47.60	63.51	81.16	99.99	69.89	67.21	84.44	70.56
	Code coverage	100.00	47.60	63.51	81.16	99.99	69.89	67.21	84.44	70.56
	Pair-wise	100.00	95.87	100.00	98.24	99.99	99.58	95.90	100.00	98.35
OPENSSL	Total	154	1 153	145	15 433	439	1 015	23	0	18 364
	Single configuration	67.53	49.52	13.79	8.60	64.92	32.41	52.00	100.00	14.43
	Code coverage (NH)	100.00	100.00	100.00	86.46	100.00	99.90	100.00	100.00	88.62
	Code coverage	100.00	100.00	100.00	99.67	100.00	100.00	100.00	100.00	99.72
	Pair-wise	100.00	99.74	100.00	74.46	100.00	100.00	100.00	100.00	78.52

EH: error handling; DS: dead store; DF: double free; UV: uninitialized variables; NF: freeing of static memory; TC: terminate `case` blocks with `break` statements; CF: control-flow in non-`void` function; DC: dangling `switch` code

Table 4.5: Error coverage of selected sampling approaches.

The results show that none of the sampling heuristics covers all detected programming errors. Single-configuration sampling has the lowest error-coverage percentage (between 14.43 and 73.81 %, on average). Single-configuration sampling never reaches full coverage in the eight error analyses. This is because this sampling heuristics analyzes only a single configuration and, hence, covers only a small subset of the system's configuration space. Although in our setup most configuration options have been selected, single-configuration sampling lacks the ability to analyze source code with mutual exclusive configuration options. The reason for high coverage rates of single-configuration sampling in some programming errors (e.g., freeing of static memory in BUSYBOX) is that the detected errors often occur in every system variant (they are independent from variability). Since our static analyses operate on variability-aware CFGs, a sampling heuristics should include combinations of presence

conditions that arise from the presence of optional and alternative code fragments.

Code-coverage sampling with and without header files worked surprisingly well and covered most errors in the three subject systems (between 70.56 and 99.72 %, on average). The small difference (0.02 %) between both code-coverage sampling heuristics in BUSYBOX is again the result of the high number of programming errors that are configuration-independent. It is not surprising that code-coverage sampling outperforms code-coverage sampling without header files, because the latter only covers a smaller configuration space. Both sampling heuristics cover all programming errors for some of the eight analyses, and when code-coverage sampling covers all errors, so does code-coverage without headers (except for uninitialized variables and control-flow in non-`void` functions in BUSYBOX).

The results of pair-wise sampling are in the range of previous studies [Kuhn et al., 2004; Steffens et al., 2012] (between 78.52 and 98.35 %, on average). Interestingly, while pair-wise sampling reached full error coverage for some programming errors (e.g., error handling in the LINUX kernel and in OPENSLL), it does not for BUSYBOX, in which the heuristics reaches 88.87 %, at most. This may be because that we computed a pair-wise sample set for the entire BUSYBOX case study using the configuration knowledge gained from the configuration model (cf. Section 2.1.1). The generated sample set contained configurations that were invalid for some files (additional dependencies in the build system rendered them unsatisfiable). The remaining configurations only covered a subset of the configuration space in a single file and, therefore, detected only a smaller number of programming errors. Using pair-wise sampling at the level of files will most likely improve the coverage rate. However, this approach is more expensive (cf. Section 4.3.1).

Next, we put our results into perspective and discuss possible improvements for both analysis approaches.

4.5 Perspectives

Even though variability-aware analysis scales well to large-scale configurable systems, there is still much room for improvement. Although we reported sequential times for analyzing the case studies, we can easily parallelize the analysis for both approaches, as all files are analyzed in isolation.

During variability-aware analysis, we can avoid many redundant computations by analyzing source code from header files only once. C's reuse mechanism for source code is based on the header-file inclusion. Developers usually outsource common source code into header files and include it in C source files if necessary. According to Kästner et al. [2011], each C file in LINUX includes 353 header files, on average, and, therefore, source code of header files is analyzed (parsing, type checking, and data-flow analysis) again and again. A reuse mechanism for analysis results (AST, typing information, data-flow information) for common, reoccurring source code, as proposed by Garrido [2005], is likely to reduce analysis times significantly.

The most expensive operations during variability-aware analysis arise from SAT solving. SAT is NP-complete, but problems that arise in the context of analyzing configurable systems

are tractable with current SAT-solvers [Thüm et al., 2009]. Furthermore, caching can be an effective means to speed up analysis [Apel et al., 2010b; Kolesnikov et al., 2013]. Still, solving a single SAT problem in our LINUX case study takes half a second, on average, on standard hardware. While progress in the development of SAT solvers reduced the time for solving SAT problems in the context of configurable systems considerably (e.g., [Johansen, 2013]), we observed that it still takes a significant amount of time to solve a high number of SAT problems. To speed up our variability-aware analyses, we already made extensive use of SAT caching. Before passing a SAT problem to the SAT solver, we consulted a cache that stores the results of previous SAT calls. So far, we have not used a persistent storage for SAT results. Subsequent or reoccurring analyses would greatly benefit from such a storage, as the number of distinct SAT calls decreases over time.

We have already highlighted that our data-flow analyses suffer from two limitations: restriction to intra-procedural analysis and lack of pointer-alias information. Integrating a variability-aware pointer analysis will reduce the number of false positives substantially and exhibit errors that our analyses currently miss (false negatives). This integration will increase the number of SAT checks on the one hand, because two pointers may be in an alias relation in a certain configuration only, which needs to be checked. On the other hand, ruling out many impossible errors due to pointer-alias information will reduce the number of SAT calls for checking error conditions of potential programming errors. With respect to inter-procedural analysis, we can extend our intra-procedural CFGs by resolving function calls to their (potentially varying) function definitions. By adapting the tracking of data-flow properties, i.e., mapping parameters of the function call to formal arguments of the function definition, we are able to track data-flow properties across function boundaries.

Variability-aware analysis is applicable also to alternative implementation techniques, such as aspects or feature modules. Apel et al. [2010a] and Kolesnikov et al. [2013] developed variability-aware type checkers for feature-oriented extensions of Featherweight Java and Java, respectively. Technically, the authors also used variability-aware data structures and algorithms for handling variability. Although the representations and algorithms are different, a representation with variability-aware ASTs and CFGs to support variability-aware type checking and data-flow analysis is possible. In experiments with both type checkers, the authors observed that variability-aware analysis is superior in comparison to traditional analysis approaches. Although the authors' experiments do not reach the size of our experiments in terms of variable code base, of available configuration options, and of the existing configuration knowledge, the results are promising, and variability-aware analysis for alternative implementation techniques will most likely scale to large-scale configurable systems, too.

4.6 Related Work

There is a large body of work on the analysis of configurable systems, including the detection of programming errors. We focus on sampling-based and variability-aware analysis approaches and review them with respect to analysis time, upfront investment, and error detection.

Analysis Time

Variability-aware type systems have been studied for some time, including systems for feature-oriented programming [Thaker et al., 2007; Kim et al., 2008; Apel et al., 2010a; Kolesnikov et al., 2013] and configurable systems developed with preprocessors [Aversano et al., 2002; Post and Sinz, 2008; Kenner et al., 2010; Kästner et al., 2012b; Chen et al., 2014]. While type systems for both development approaches are different, researchers experienced enormous speedups in analysis times when applying variability-aware type checking compared to brute-force analysis. For example, Kolesnikov et al. [2013] compared variability-aware type checking with a variability-unaware type-checking approach for 12 feature-oriented systems. For systems with a comparatively large configuration space, the speedup of variability-aware analysis over the brute-force approach reaches two orders of magnitude, and the break-even point, at which variability-aware analysis supersedes a traditional analysis, is quickly reached (two to eight system variants). We showed that the break-even-point for annotation-based systems is similar, even though all three systems that we analyzed contain billions of valid system variants. Hence, obtainable speedups are even higher.

There are several proposals for variability-aware data-flow analyses [Bodden et al., 2013; Brabrand et al., 2013] that implement data-flow analysis for Java programs, in which variability is expressed with annotations in Java comments using a preprocessor similar to CPP. Brabrand et al. [2013] systematically explored different variants of variability-aware analysis. The variants differ in handling variability internally (CFG, a lattice storing intermediate results, and the transfer functions). A comparison of the analysis variants with brute-force analysis shows that variability-aware analysis, in general, outperforms brute-force analysis, if a system has a large configuration space. The authors omit the variant that is similar to our data-flow analyses (\mathcal{A}_4 : analysis with sharing and merging) during their comparative evaluation. This analysis variant is particularly advantageous, because it ensures fast analysis computation, including redundancy-free storage of analysis results.

Bodden et al. [2013] extended a framework for inter-procedural data-flow analysis, with support for variability. Using an inter-procedural CFG, analysis problems are formulated as graph-reachability problems; data-flow properties are forwarded along the CFG using flow functions. To account for variability in the input representation, the authors solely enriched the CFG to make the data-flow analysis variability-aware. In a set of experiments with three data-flow analyses, the authors could show that their variability-aware analysis outperforms traditional analysis by several orders of magnitude.

Both approaches [Brabrand et al., 2013; Bodden et al., 2013] make limiting assumptions about the form of annotations, in particular, the limitation regarding type uniformity [Kästner et al., 2012a] and annotation discipline [Kästner et al., 2008; Liebig et al., 2011]. We have already stressed that variables in C may have different types in different configurations, which has to be accounted for in the analysis. Technically, both approaches do not support variables with different types, which limits their application for real-world systems substantially. Furthermore, the authors used comparatively small and academic case studies in their experiments, so the generalizability of the results for large systems is questionable. In contrast, we demonstrated

the feasibility and scalability of data-flow analysis regarding software systems at the size of the LINUX kernel.

Apart from annotation-based systems developed with preprocessors, such as CPP, researchers proposed and implemented product-line type systems for feature-oriented programming (e.g., [Apel et al., 2010a; Kolesnikov et al., 2013]). For example, Kolesnikov et al. [2013] implemented a type checker for a feature-oriented extension of Java, called FUJI [Apel et al., 2012]. From a technical point of view, FUJI works similar to TYPECHEF; source code of the configurable system is parsed and represented with variability-aware ASTs. FUJI's variability-aware type checker traverses these ASTs and incorporates configuration knowledge during type checking. In a case study of 12 (academic) product lines, the authors observed that variability-aware type checking outperforms a naïve brute-force approach, i.e., type checking all system variants in isolation. Interestingly, the authors determined that the break-even point of variability-aware analysis lies between two to eight configurations. Our results show a similar picture. In a similar way, Apel et al. [2010a] proposed Feature Featherweight Java, an extension of Featherweight Java [Igarashi et al., 2001] with support for feature-oriented programming. The authors extended the formal syntax definitions of Featherweight Java and developed a variability-aware type checker on top of it, including a soundness proof. The variability-aware type checker has never been intended to give a realistic picture of checking type correctness in the presence of variability, but to discuss possible strategies for the development of variability-aware type checkers. The authors' work influenced the development of our variability-aware analyse, and our results show that variability-aware analysis scales to real-world, large-scale configurable systems.

Upfront Investment

Sampling-based analysis and variability-aware analysis are both in the center of active research.

With respect to the development of variability-aware analysis, researchers proposed different procedures to enrich a traditional analysis approach or reported from experiences in variability-aware analyses' development [Kästner et al., 2012a; Apel et al., 2010a; Chen et al., 2014; Brabrand et al., 2013; Bodden et al., 2013; Midtgaard et al., 2014].

For example, Bodden et al. [2013] enriched a framework for inter-procedural data-flow analysis with variability by solely adding presence conditions to the CFG representation, on which the framework operates. Existing data-flow analyses for non-variable programs can be reused without changing analysis algorithms, because the framework instantiates and runs data-flow analyses with variability.

Brabrand et al. [2013] developed five different variants of variability-aware data-flow analyses by systematically adding configuration knowledge to the involved data structures and algorithms. In a way, the variants represent the progress of creating variability-aware analysis, from an initial variability-unaware analysis to an analysis variant incorporating our three analysis patterns late splitting, early joining, and local variability representation (cf. Section 4.3.2).

We can observe the three patterns of late splitting, early joining, and local variability representation in different analysis approaches. For example, when parsing `#ifdef` code with

TYPECHEF's variability-aware parser [Kästner et al., 2011], configuration knowledge occurs in the form of `#ifdef` annotated tokens in the input stream (i.e., the output of lexical analysis) of the parsing process. The parser maintains a context, storing the configuration knowledge of the currently processed input. During parsing, TYPECHEF performs late splitting when the parser context changes after configuration knowledge from the input has been added to it (parsing an alternative, such as `#ifdef A-#else-#endif`). In this case, TYPECHEF parses both branches separately with different parsers, each maintaining a separate context. TYPECHEF performs early joining when different parser instances reach the same input, i.e., the same position in the input token stream. To represent variability in the parsing output, TYPECHEF uses variable AST nodes (cf. Section 4.1.2), which represent local variations in the input source code.

In a similar way, variability-aware model checking and abstract interpretation make also use of the three patterns of variability-aware analysis [Apel et al., 2013d; Kästner et al., 2012c]. During program analysis, configuration knowledge is added as context to the program state, which contains elements, such as the program counter and values of variables. Configuration knowledge is stored as a context to program states in the form of presence conditions (local variability representation). Program instructions manipulate the program state and create new ones when the program analysis proceeds. The analysis approaches create program states in different contexts if and only if they actually differ (late splitting). Equal program states (same values for variables and the same program counter) can early be joined by an or-conjunction of the program-states presence conditions, similar to the union operation in our liveness implementation (cf. Section 4.1.2).

The three patterns are a good starting point when enriching analysis approaches with variability. From a theoretical point of view, there are many different applications for variability-aware analysis, including type inference for configurable systems [Chen et al., 2014] and the definition and use of variability-aware data structures [Walkingshaw et al., 2014], such as variability-aware graphs [Erwig et al., 2013].

Research on sampling approaches mainly focuses on the improvement of sampling heuristics with respect to the sample-set generation time and the number of generated configurations. The algorithms of sampling heuristics are often NP-complete (cf. Section 4.3), and their computation requires powerful hardware, heavy tuning, and sophisticated heuristics [Johansen et al., 2011; Henard et al., 2014]. Nevertheless, sample-set generation takes a considerable amount of time and is sometimes not even possible in a given scenario. Johansen [2013] proposed a new algorithm for pair-wise sample-set computation as part of the tool SPLCATOOL, which we also used in Section 4.3.1, and he compared it to other pair-wise sampling algorithms. Although SPLCATOOL performs best in the comparison, its application to configurable systems with a large number of configuration options requires a system with a lot of memory (up to 128 GB). In contrast, when solving propositional formulae as part of variability-aware analysis, a standard computer with 2 to 8 GB of memory is sufficient.

Error Coverage

Sampling-based analysis is de facto the standard in detecting programming errors in configurable systems. There are several studies discussing the application of sampling heuristics for error detection [Kuhn et al., 2004; Steffens et al., 2012; Garvin and Cohen, 2011; Tartler et al., 2012, 2014].

Kuhn et al. [2004] analyzed bug reports for several large software systems. Each bug was triggered by a combination of different program parameters. The authors determined that an exhaustive test of all combinations of up to six program parameters is sufficient to find all errors. Analyzing all combinations of six parameters is equivalent to 6-wise sampling (cf. Section 4.4). Although we consider configuration options at compile time, setting program parameters for program execution is a similar problem. Generating sample sets is often very expensive from a computational perspective, and it remains unclear whether 6-wise sample sets can efficiently be generated for large configurable systems with current sampling algorithms [Johansen, 2013].

Steffens et al. [2012] investigated the effectiveness of pair-wise sampling in a configurable system from industry (Danfoss Automation Drive). To this end, they determined the error coverage of pair-wise sampling after an exhaustive test of all 432 system variants, which could be derived from the system. For comparison, the authors mutated parts of the source code, which was tested using the system's test suite for errors afterwards. The generated pair-wise sample-set (with 57 system configurations) covers around 97 % of all errors.

In contrast to both studies, in which the application of pair-wise sampling was crucial to detect errors, Garvin and Cohen [2011] analyzed configuration-dependent errors in two software systems (GCC and FIREFOX). The authors observed that only 3 of 28 identified errors are dependent upon system configuration, and, as such, the usefulness of sampling heuristics is rather limited. Besides the issue of sample-set generation, there is no guarantee that the number of configuration options, which are involved in a programming error, is fixed (e.g., six). Kuhn et al. [2013] report from application scenarios, in which nine system options are involved. Apel et al. [2013c] even report from software systems, in which up to 33 options are involved. So in the worst case, all configuration options of a system may be involved in an error's configuration constraint, and, therefore, more system variants need to be analyzed to guarantee system correctness.

In our context, the application of code-coverage sampling proved most successful. This sampling approach was proposed in the context of configurable systems (in particular for LINUX) [Tartler et al., 2012, 2014]. Using this sampling heuristics, the authors could increase the code coverage from an initial 70 % (using the *alloyesconfig*) to 84 %. The authors integrated code-coverage sampling in their tool VAMPYR,¹³ which automatically derives code-coverage sampling sets and applies static analysis tools, such as GCC, to derived system variants in order to find bugs. This approach, although incomplete, proved very successful, because it revealed many warnings and errors previously unknown [Tartler et al., 2014]. In contrast to single-configuration sampling and pair-wise sampling, code-coverage sampling operates on the input source code directly, and it generates system configurations with an improved

¹³<http://vamos.informatik.uni-erlangen.de/trac/undertaker/>

error coverage. However, none of the sampling heuristics takes the source code into account. Sampling heuristics may perform differently, depending on the static analysis in question. For example, both control-flow and data-flow analyses are different with respect to variability. When computing CFGs, the successor/predecessor computation interrelates one AST node, possibly annotated, with another. A sampling heuristic that takes such characteristics into account generates a sample set, which is closer to system variants required by end users. It is an interesting endeavor to investigate alternative sampling approaches that operate on static analysis information and to measure their performance in a similar setup as ours.

Only recently, Abal et al. [2014] presented a qualitative study of 42 variability bugs in the LINUX kernel. The authors collected bug-fixing commits from the kernel repository and classified confirmed bugs (by kernel developers) with respect to common programming errors in C. They observed that variability bugs occur in any location, are not bound by particular, “error-prone” configuration options, and are not limited to any kind of programming error. This result emphasizes the necessity of comprehensive analysis approaches for error detection. Although most errors require different static analyses than the ones we developed (e.g., *assertion violation* or *memory leakage*) or are inter-procedural, our variability-aware static analyses (cf. Section 4.4.1) would find three errors.¹⁴ Additionally, the authors investigated the error condition of each identified bug and observed that 12 bugs involve a condition of three or more configuration options. This is an interesting observation, because the commonly applied pair-wise sampling heuristics would find these bugs only by accident. With respect to error detection, the authors’ bug collection is an ideal test bed for ground research on variability-aware analysis and can improve static analysis (including ours) significantly.

In the same line of research, Coker et al. [2014] analyzed integer issues in C source code of large-scale, configurable systems. Integer issues pose a threat to system security, as integer vulnerabilities can be exploited by attackers. In particular, they investigated signedness issues (mixing signed and unsigned variables) as well as overflow and underflow issues. The authors used variability-aware analysis to detect both issues in versions of BUSYBOX and LINUX that we also used. They observed that integer issues occur in configuration-option dependencies, in which up to 11 options are involved. This confirms the necessity of comprehensive analysis approaches, as, once again, sampling-based analysis would find these issues more by accident than by design.

4.7 Summary

Analyzing large-scale, configurable systems is challenging, as contemporary systems often have billions of valid system variants. Sampling-based analysis as well as variability-aware analysis reduce the effort of analyzing configurable systems in different ways, either by lowering

¹⁴As each error’s presence condition is comparatively simple, sampling-based analysis would most likely have found the errors, too. The corresponding bug fixes are available on <https://github.com/torvalds/linux/commit/36855dc>, <https://github.com/torvalds/linux/commit/7acf6cd>, and <https://github.com/torvalds/linux/commit/e39363a>.

the number of variants to be analyzed or by avoiding redundant computations of analysis results for similar variants by incorporating configuration knowledge in the analysis process. In an extensive comparison of both analysis approaches using three criteria (analysis time, upfront investment, and error coverage), we determined the strengths and weaknesses of both approaches.

Based on heuristics, sampling-based analysis determines a sample set of valid configurations for analysis and analyzes them using a traditional, variability-unaware analysis. While analyzing a single variant with this approach is not expensive, analysis results are incomplete, because only a subset is analyzed. In contrast, variability-aware analysis incorporates configuration knowledge in the analysis process and analyzes common source code only once. Even though this analysis approach is more expensive than sampling-based analysis, variability-aware analysis can outperform some sampling heuristics with respect to analysis time. In our experience, the break-even point for analyzing configurable systems is between two and four. That is, when analyzing more than four variants, one should use variability-aware analysis rather than sampling-based analysis.

Considering the upfront investment of both analysis approaches, the application of sampling heuristics is dominated by the effort made for the sample-set computation, for which some sampling heuristics employ algorithms that are NP-complete. In our experience, the computation of sample sets is sometimes very expensive and may even take longer than the subsequent analysis of system variants. Existing variability-aware analyses are often from-scratch implementations of existing analysis techniques/tools and usually lack the maturity of traditional analyses. To improve the situation, we derived three patterns of variability-aware analyses that capture the essential ingredients of efficient analysis computation. An analysis splits when it encounters changes of configuration knowledge in the input representation and joins intermediate analysis results as soon as it reaches a common state in the input again. Local variations in analysis results are efficiently stored in compact representations that only store differences between variants. We transferred the insights of these patterns to the development of a variability-aware data-flow framework. The framework simplifies the development of static analyses and makes error detection available to a large number of configurable systems at one stroke.

With variability-aware analysis, we analyzed three large-scale, real-world configurable systems for eight different programming errors (control-flow and data-flow errors) to determine the error coverage of sampling heuristics. Our analysis included the detection of serious programming errors, such as the use of uninitialized variables, which may lead to unexpected/undefined system behavior. In a nutshell, we determined that each sampling heuristics fails to detect all programming errors that we detected using variability-aware analysis. The common practice of analyzing only a single, yet common system variant results in the detection of between 14 and 74 % of all errors, on average, and never reached full error coverage for the analyzed errors. Alternative sampling approaches (code-coverage sampling and pair-wise sampling) that are not limited to the analysis of a single variant were able to detect a higher number of programming errors. Only for a subset of analyses, both sampling approaches detected all errors. That is, to be certain about detecting programming errors, one should strive for a complete analysis by using variability-aware analysis techniques.

There are many interesting opportunities for future work. First of all, extending intra-procedural analysis to make it inter-procedural enables developers to detect more programming errors. Our implementation of variability-aware CFGs already allows such an extension and gives rise to the creation of further analysis frameworks, incorporating variability, which brings variability-aware analysis to many real-world systems at a single stroke. Furthermore, with the idea of evolving systems in mind, an interesting endeavor is to explore possibilities to reuse analysis results in subsequent analyses, including analyses after source-code changes. To this end, the main challenge is to determine code changes and their impact on analysis results. Integrating reuse of analysis results into variability-aware analysis will greatly improve the analysis performance.

5 Refactoring C Code with #ifdefs

This chapter shares material with the ICSE'15 paper „Morpheus: Variability-Aware Refactoring in the Wild“ [Liebig et al., 2015].

As most software systems, configurable systems evolve. *Refactoring* is an important approach to deal with software evolution [Mens and Tourwé, 2004]. Although refactoring has been studied thoroughly in academia and proved successful in practice [Mens and Tourwé, 2004], its applicability to configurable systems is a problematic case. This is because the variability of configurable systems adds a new dimension of complexity that has not been tamed so far, as we will illustrate. A key challenge is to ensure *behavior preservation* not only of a single system, but of all system variants that can possibly be derived from a configurable system. This turns out to be problematic because of the possibly huge configuration space (cf. Chapters 3 and 4).

Existing refactoring approaches and tools use heuristics to reason about variability (which does not guarantee behavior preservation) [Garrido, 2005; Padioleau et al., 2008], employ a brute-force strategy to process all variants individually (which does not scale to realistic systems) [Vitteck, 2003; Waddington and Yao, 2007; Spinellis, 2010], or limit the use of variability (which makes many systems unrefactorable) [McCloskey and Brewer, 2005; Baxter et al., 2004; Hafiz and Overbey, 2012; Platoff et al., 1991]. To make matters worse, state-of-the-art refactoring engines of widely used IDEs, such as ECLIPSE and XCODE, even produce erroneous code for standard refactorings in the presence of preprocessor directives (e.g., RENAME IDENTIFIER or EXTRACT FUNCTION).

To improve refactoring of configurable systems beyond the state of the art, we strive for a variability-aware refactoring solution that preserves the behavior of all variants, that is general (does not rule out large sets of configurable systems), and that scales to systems of substantial size (hundred thousands of lines of code). Similar to our discussions on analyzing C code with preprocessor annotations (cf. Chapter 4), we rely on variability-aware data structures and variability-aware analyses to make *variability-aware refactoring* possible in the first place.

While there were some proposals for variability-aware refactoring, based on academic languages and tools [Schulze et al., 2013b,c,d], we go beyond that by supporting the full power of C and CPP as well as applications in real settings. To this end, we developed the refactoring engine MORPHEUS, which implements the three standard refactorings RENAME IDENTIFIER, EXTRACT FUNCTION, and INLINE FUNCTION as a proof of concept. We applied MORPHEUS to the three substantial, real-world systems BUSYBOX, OPENSLL, and SQLITE to assess its correctness and scalability. Although the engine internally relies on solving many SAT problems, it scales far beyond state-of-the-art tools (that guarantee behavior preservation): the response time of a standard refactoring is, on average, less than a second and so in the range

of standard refactoring tools such as ECLIPSE [Gligoric et al., 2013].

For each subject system, we used a substantial test suite to provide evidence for the correctness of our refactoring engine. To this end, we extended a standard approach of testing refactoring engines [Gligoric et al., 2013] with support for variability.

5.1 State of the Art

Before we introduce our approach of variability-aware refactoring, we review the refactoring capabilities of state-of-the-art IDEs for C. We outline their operation principles and discuss shortcomings in the presence of preprocessor directives. First, we establish the terminology that we use throughout this chapter. Refactoring is “the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure” [Fowler, 1999]. Code transformations employed by developers usually follow a set of *refactoring patterns*, such as RENAME IDENTIFIER or EXTRACT FUNCTION, which have been documented by practitioners and researchers [Fowler, 1999]. A *refactoring engine* implements these patterns as (semi-)automatic code transformations. We call the application (including preparation and execution) of a particular pattern within a refactoring engine a *refactoring task*. If it is clear from the context, we simply use the term refactoring. Next, we review a number of publicly available IDEs for C and their refactoring engines, including commercial tools, open-source tools, and research prototypes. Most IDEs lack a refactoring engine and provide only a simple textual search-and-replacement functionality. Since such functionality is barely a compensation for a missing refactoring engine and only of limited usability, even for simple refactoring patterns, such as RENAME IDENTIFIER, we omit them in our discussion and focus on IDEs with dedicated refactoring engines. Such engines follow one of four operation principles: no variability support, variant-based, disciplined subset, and heuristics.

No Variability Support

Common refactoring tools, such as ECLIPSE and XCODE, provide a set of basic refactorings, including RENAME IDENTIFIER, EXTRACT FUNCTION, and INLINE FUNCTION that are usually not variability-aware. To handle variability induced by preprocessor directives, these tools employ engines that evaluate CPP directives implicitly using default values of a configurable system’s project setup. Both ECLIPSE and XCODE basically operate on only a default variant, which represents a single system variant of possibly billions. In practice, this can easily lead to errors. For example, in Figure 5.1 we depict the application of two refactorings: RENAME IDENTIFIER in XCODE and EXTRACT FUNCTION in ECLIPSE. We were able to apply both refactorings without any negative feedback from the refactoring engines. Unfortunately, after the application, the transformed code contained errors for some system variants. As for RENAME IDENTIFIER, not all depending identifiers had been renamed, causing a type error if a particular variant was compiled (the variant A in Figure 5.1c). As for EXTRACT FUNCTION, after selecting and extracting of a list of statements had been over, the resulting code

compiled, but had an altered behavior due to a change in the statement order (cf. Figure 5.1d vs Figure 5.1e).

```

1 #ifndef A
2 int global = 1;
3 #else
4 int global = 0;
5 #endif
6
7 int foo() {
8     int local = global;
9     return local;
10 }

```

(a) Before Rename identifier `global`

```

1 #ifndef A
2 int global = 1;
3 #else
4 int activated = 0;
5 #endif
6
7 int foo() {
8     int local = activated;
9     return local;
10 }

```

(b) After Rename identifier `global`

```

1 [A] file xcode.c:8:16--file xcode.c:8:25
2 activated undeclared (only under condition !A)

```

(c) Type error: identifier `activated` not defined in variant A

```

1 #include <stdio.h>
2 #define DEBUG 1
3
4 int main() {
5     if (DEBUG) {
6         printf("Debug mode entered.\n");
7         #ifdef A
8             printf("Option A enabled.\n");
9         #endif
10        printf("Debug mode left.\n");
11    }
12    return 0;
13 }

```

Output of variant A:

```

1 Debug mode entered.
2 Option A enabled.
3 Debug mode left.

```

(d) Before extracting function `foo`

```

1 #include <stdio.h>
2 #define DEBUG 1
3
4 void foo() {
5     printf("Debug mode entered.\n");
6     printf("Debug mode left.\n");
7 }
8
9 int main() {
10    if (DEBUG) {
11        foo();
12    }
13    #ifdef A
14        printf("Option A enabled.\n");
15    }
16    return 0;
17 }

```

Output of variant A:

```

1 Debug mode entered.
2 Debug mode left.
3 Option A enabled.

```

(e) After extracting function `foo`

Figure 5.1: Before (cf. Figure 5.1a) and after (cf. Figure 5.1b) applying `RENAME IDENTIFIER` in `XCODE`; type error after renaming (cf. Figure 5.1c); before (cf. Figure 5.1d) and after (cf. Figure 5.1e) applying `EXTRACT FUNCTION` in `ECLIPSE` with the corresponding program outputs.

Variant-based

Some refactoring engines cannot handle variability induced by `#if` directives directly [Vitek, 2003; Spinellis, 2003, 2010; Waddington and Yao, 2007]. Instead, they employ a variant-based approach. That is, they generate all system variants that are affected by a refactoring, apply a particular refactoring task to each variant independently, and lead the result back to the variable code base. To this end, a developer has to specify one or more configurations, which serve as an input for the generation of system variants. During configuration specification, developers assign each configuration option of the configurable system a *true/false* value for the option's selection or deselection, respectively. Even though the specification process is sometimes supported by the aid of a tool, a major drawback is that specifying system configurations remains a tedious and error prone task (in particular if done manually). Furthermore, errors in the specification process may easily lead to incorrect code.

The variant-based approach rests on two assumptions. First, a system's number of valid configurations is often low, so, configuration specification can be handled manually. Second, the complexity induced by `#if` directives cannot be handled by refactoring algorithms in practice. In particular, checking the satisfiability of configurations, which is a frequent task when refactoring C code, is difficult. Both assumptions do not hold in practice. First, as identified in Chapter 3 and Chapter 4, configurable systems usually have a huge number of configuration options leading to billions of valid configurations. Second, we as well as others observed that reasoning about configuration knowledge is tractable even for large software systems using BDDs or SAT solvers [Chen et al., 2009; Thüm et al., 2009; Liebig et al., 2013]. Additionally, variant-based approaches face a severe limitation. Since refactoring tasks are solely applied to individual variants of a system, all transformed variants have to be merged—a problem that is challenging in its own right [Mens, 2002]. To make the merging process tractable, existing engines often only support `RENAME IDENTIFIER`, for which merging is easy to apply in comparison to merging, for example, different variants of a function after applying `EXTRACT FUNCTION`.

Disciplined Subset

In Section 3.3, we discussed the benefit of disciplined `#if` annotations for the development of tool support. Since refactoring engines require structured input, usually in the form of an AST, a common idea is not to allow arbitrary annotations of code [Baxter and Mehlich, 2001; Platoff et al., 1991], but to limit the developer to a subset of disciplined annotations (cf. Section 3.1.3): annotations on entire functions, type definitions, and statements. If developers solely use disciplined annotations, `#if`-annotated source code can be parsed based on preprocessor-enriched grammars [Baxter and Mehlich, 2001; Liebig et al., 2011; Kästner et al., 2011; Gazillo and Grimm, 2012], and an AST with variability information can be used for further processing. To apply such a disciplined-subset approach, arbitrary, undisciplined annotations have to be transformed (manually) to disciplined annotations. Although researchers experienced that such manual labor is feasible and scales up to medium-sized software systems [Baxter

and Mehlich, 2001; Platoff et al., 1991], we observed that undisciplined annotations occur frequently in practice (cf. Section 3.3.1). Consequently, manually disciplining annotations for such systems is a tedious and error-prone task that will hardly be adopted in practice. Similarly, the substitution of CPP by a new language for source-code preprocessing also involves manual code disciplining [Boucher et al., 2010; McCloskey and Brewer, 2005]. Even after undisciplined annotations have been disciplined, the creation of a variability-aware AST is particularly challenging, because complex interconnections between `#ifdef` directives and `#define` macros have to be considered [Kästner et al., 2011]. Existing approaches fail to handle interconnections properly, as they do not employ a sound and complete parsing approach.

Heuristics

Several approaches use heuristics to avoid the manual labor of transforming undisciplined annotations to disciplined annotations [Garrido and Johnson, 2005; Padioleau, 2009]. Similar to the disciplined-subset approach, an engine uses an `#ifdef`-enriched grammar for the creation of ASTs with variability information. Heuristics either automatically rewrite `#ifdef` annotations that do not align with the grammar specification, or report problematic code, which cannot be parsed, to the developer for manual rewrites [Garrido and Johnson, 2005; Padioleau, 2009]. But an AST created with unsound heuristics introduces an additional source of error on top of the refactoring challenge. There are some approaches using heuristics, which were successfully applied for code transformations. For example, Padioleau et al. [2008] developed COCCINELLE, a program-matching and transformation engine based on semantic patches. A semantic patch is a declarative specification of a generic program transformation, similar to a refactoring pattern, which can be exploited for the application of refactoring tasks, too. The developers of COCCINELLE successfully applied a set of semantic patches to the LINUX kernel [Padioleau et al., 2008]. An alternative tool, solely aiming at refactorings is CREFACTORY [Garrido and Johnson, 2003; Garrido, 2005]. CREFACTORY provides a set of simple refactoring patterns (e.g., DELETE UNUSED VARIABLE, MOVE VARIABLE TO STRUCTURE DEFINITION, and RENAME IDENTIFIER), neglecting complex patterns (e.g., EXTRACT FUNCTION). In addition to the unsound heuristics during parsing, CREFACTORY employs heuristics for reasoning about configuration options: the engine comes without a SAT solver or BDDs for answering configuration-related questions (e.g., whether a code fragment is still selectable after applying FUNCTION INLINE). To the best of our knowledge, CREFACTORY was only applied in refactoring tasks of small, manageable software systems. Thus, its scalability is unclear [Garrido, 2005].

Summary

Existing engines fail to refactor C code properly for different reasons: they provide only facilities for simple textual search and replacement, do not support variability, are incomplete and use heuristics, or do not scale. A textual search-and-replacement approach supports only

RENAME IDENTIFIER and is an inadequate substitute for a proper refactoring engine that employs consistency checks. There are two reasons for incompleteness. First, an engine handles only a single (often standard) configuration neglecting multiple configurations due to its inability for handling `#ifdef` directives. Second, as configuration knowledge is not fully handled by an engine (it neglects a system's configuration model or uses heuristics), the engine may introduce programming errors in some system variants. In the end, most approaches have scalability issues, since they internally apply a variant-based approach, either partially on selected parts of the source code or on entire files. Table 5.1 summarizes the findings of our investigation of IDEs and their refactoring capabilities.

5.2 Variability-aware Refactorings with Morpheus

To overcome the limitations of state-of-the-art refactoring engines, we built MORPHEUS on top of variability-aware ASTs (cf. Section 4.1.2) and derived data structures such as variability-aware CFGs (cf. Section 4.1.2). These data structures serve as an input for variability-aware algorithms to compute static analysis information that is required during the refactoring process. For example, most refactoring engines exploit type and reference information. We exploit TYPECHECKER's variability-aware type checker (cf. Section 4.1.2) to get reference information, e.g., references between variable declarations and variable usages and vice versa (`RefInf=Map[Id, List[Choice[Id]]]`). For example, Figure 5.2 shows reference information for the variable `global` of our RENAME-IDENTIFIER example (cf. Figure 5.1a). All references of the variable `global` in Line 8 are linked to their original declarations including presence conditions (`List[Choice[Id]]`). In a similar fashion, we created and used variability-aware CFGs (cf. Section 4.1.2). For example, the successor of statement `printf("Debug mode entered.\n")` in Line 6 in our EXTRACT FUNCTION example (cf. Figure 5.1d) is either `printf("Option A enabled.\n")` (Line 8) or `printf("Debug mode left.\n")` (Line 10), depending on the selection of the configuration option A. Such information is crucial for the definition of variability-aware refactoring patterns, such as EXTRACT FUNCTION or INLINE FUNCTION [Schäfer et al., 2009].

Closest to our work is the project OPENREFACTORY/C [Hafiz and Overbey, 2012; Hafiz et al., 2013], which, similar to our proposal, employs variability-aware data structures and algorithms to cope with multiple system configurations. Up to now, the available prototype is not ready for production use and suffers from several limitations, e.g., support for only one configuration [Hafiz et al., 2013] and missing support for preprocessor directives `#define` and `#include` in the source code.

5.2.1 Specification of Refactorings

As representative and widely used refactoring patterns [Murphy-Hill et al., 2012], we selected RENAME IDENTIFIER, EXTRACT FUNCTION, and INLINE FUNCTION. For the definition and implementation of the refactorings, we abstract from the underlying variability-aware analysis

Refactoring engine	Version/Reference	Textual Search & Replacement	No Variability Support	Variant-based	Disciplined Subset Heuristics
COCCINELLE	[Padioleau et al., 2008]				✓
CODE::BLOCKS	10.05 ¹	✓			
CODELITE	2.8.0 ¹	✓			
CREFACTORY	[Garrido, 2005]				✓
CSCOUT	[Spinellis, 2010, 2003]		✓		
DMS	[Baxter et al., 2004]			✓	
ECLIPSE CDT	8.2.1 ²	✓			
GEANY	0.21 ¹	✓			
GNAT GPS	5.0-6 ¹	✓			
KDEVELOP	4.3.1 ¹	✓			
MONODEVELOP	2.8.6.3 ¹	✓			
NETBEANS IDE	7.4 ¹		✓		
PROTEUS	[Waddington and Yao, 2007]			✓	
PTT	[Platoff et al., 1991]				✓
VISUAL STUDIO	2013 Prof. ⁴	✓ ⁵			
XCODE	5 ³		✓		
XREFACTORY	[Vitteck, 2003]			✓	

¹<http://freecode.com>; ²<http://eclipse.org/cdt/>;
³<http://developer.apple.com/xcode/>; ⁴<http://microsoft.com/visualstudio/>;
⁵by default support only via one of several, proprietary extensions

Table 5.1: Classification of refactoring support in integrated development environments.

framework and rely on an interface, as illustrated in Figure 5.3. Note that the signature of the interface incorporates variability (use of the `Choice` type constructor; cf. Section 4.1.2).

Rename Identifier

The challenge of specifying `RENAME IDENTIFIER` correctly is that all identifiers in a configurable program (e.g., function names, function parameters, local or global variables, and

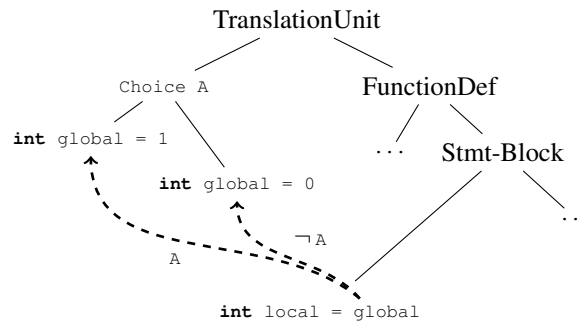


Figure 5.2: AST representation enriched with reference information of the RENAME-IDENTIFIER example in Figure 5.1a; Choice A represents a variable AST node providing a selection of two different definitions of variable `global`.

user-defined data types) may vary depending on some configuration options. In our RENAME-IDENTIFIER example (cf. Figure 5.1a), the variable `global` is defined twice (annotated with `A` and `¬A`, respectively). For consistent renaming of such identifiers, which can possibly be scattered across multiple source files, we employ reference information (for the same file using `RefInf` and across files using `CProgram`). If we select one identifier for renaming, we rename also all dependent references, even across multiple files.

The refactoring expects the following input: a selected identifier (*oid*), a variable AST (*tunit*), a global linking interface (*li*), and a name for the new identifier (*nid*). MORPHEUS applies the refactoring as follows: after checking that *nid* conforms to the C standard of identifiers (`isValidId`), the engine applies variability-aware type checking on the variable input AST (`typeCheck`). As a result, we get a type environment (*te*) including all identifiers and their (possibly variable) types (`Map[Id, List[Choice[Type]]]`) and reference information (*ri*) of declarations/uses of variables to their corresponding uses/declarations (`Map[Id, List[Choice[Id]]]`). While the former map is not needed for this refactoring pattern, the latter map helps to preserve name binding—the crucial property of this refactoring pattern. For example, when renaming the variable `global` in Line 8 in our RENAME-IDENTIFIER example (cf. Figure 5.1a), the engine determines the transitive closure of identifier uses and their declarations (`getModuleReferences`). The resulting list contains all variable identifiers including their presence condition: $rid = List(Choice(A, global, global))$ (cf. Figure 5.2). MORPHEUS checks the list for three conditions.

First, MORPHEUS rules out impossible renamings of identifiers in system-header files (e.g., renaming of the function `printf` in `stdio.h`)—files affected by the refactoring must be writable (`isWritable`). Although simple, this missing condition was not checked by ECLIPSE’s refactoring engine for C.¹ Second, to preserve binding and visibility of identifiers, the engine

¹https://bugs.eclipse.org/bugs/show_bug.cgi?id=396361

```

addFDef : Choice[AST] × Choice[FDef] → Choice[AST]
compatibleCFG : Choice[AST] × Choice[FDef] → Boolean
genFCall : Choice[AST] × List[Choice[Stmt]] × Id → Choice[Stmt]
genFDef : Choice[AST] × List[Choice[Stmt]] × Id → Choice[FDef]
genFPro : Choice[FDef] → Choice[Stmt]
getDefs : RefInf × List[Choice[Id]] → List[Choice[Id]]
getPC : List[Choice[Stmt]] → PC
getUses : RefInf × List[Choice[Id]] → List[Choice[Id]]
getModuleReferences : RefInf × Id → List[Choice[Id]]
getProgramReferences : CProgram × Id → List[(Choice[AST], List[Choice[Id]])]
insertBefore : Choice[AST] × Choice[FDef] × Choice[Stmt] → Choice[AST]
isFunctionCall : Choice[AST] × Id → Boolean
isRecursive : Choice[AST] × Choice[FDef] → Boolean
isValidId : Id → Boolean
isValidInModule : TypeEnv × List[Choice[Id]] × Id → Boolean
isValidInProgram : CProgram × Choice[Id] × Id → Boolean
isValidSelection : Choice[AST] × List[Choice[Stmt]] → Boolean
isWritable : Choice[Id] → Boolean
replaceFCalls : Choice[AST] × List[Id] × List[Choice[FDef]] → Choice[AST]
replacelds : Choice[AST] × List[Choice[Id]] × Id → Choice[AST]
replaceStmts : Choice[AST] × List[Choice[Stmt]] × Choice[Stmt] → Choice[AST]
typeCheck : Choice[AST] → (TypeEnv, RefInf)

```

Figure 5.3: Auxiliary functions of MORPHEUS that provide the interface to underlying variability-aware analyses and transformations.

checks possible violations of C’s scoping rules for each identifier (`isValidInModule`) that is going to be replaced with *nid*. Third, `RENAME IDENTIFIER` may affect not only the source file (module), on which the developer currently operates, but it may also affect depending identifiers. Renaming a function declaration or function call may require renamings of corresponding calls or declarations in other modules. To support refactorings with a ‘global’ effect, we rely on a data structure for module interfaces (`CProgram`), that is, a map of all modules and their imported/exported symbols (function declarations),² including presence conditions [Kästner et al., 2012b]. Using `li` (`CProgram`), MORPHEUS determines defined symbols (limited to declarations determined with `getDefs`) in conflict to *nid* (`isValidInProgram`), and terminates if there are any. If all premises (`isValidId`, `isWritable`, `isValidInModule`, and `isValidInProgram`) hold, we replace all references of *oid* (using *rid*) in the variable AST *tunit* with *nid* (using `replacelds`). The `RENAME-IDENTIFIER` specification in Figure 5.4 does not

²Global variables that are externally visible are currently not supported.

include renamings of files with linked identifiers. To support their renamings, MORPHEUS uses `getProgramReferences` to fetch depending variable ASTs and linked identifiers of variables. The engine uses both information to apply renamings in depending files in the same manner as `rename` (cf. Figure 5.4).

$$\begin{array}{c}
 \boxed{\text{rename: } C\text{Program} \times \text{Choice}[\text{AST}] \times \text{Id} \times \text{Id} \rightarrow \text{Choice}[\text{AST}]} \\
 \text{isValidId}(nid) \quad (te, ri) = \text{typeCheck}(tunit) \quad rid = \text{getModuleReferences}(ri, oid) \\
 \forall r : r \in rid : \text{isWritable}(r) \quad \forall r : r \in rid : \text{isValidInModule}(te, r, nid) \\
 \forall d : d \in \text{getDefs}(ri, rid) : \text{isValidInProgram}(li, d, nid) \quad tunit' = \text{replacelds}(tunit, rid, nid) \\
 \hline
 \text{rename}(li, tunit, oid, nid) \rightarrow tunit'
 \end{array}$$

Figure 5.4: Specification of RENAME IDENTIFIER.

Extract Function

We have already seen in example Figure 5.1a that `EXTRACT FUNCTION` can be problematic. A program's control flow has to be preserved while different code transformations are performed (e.g., generating a function with required parameters). To face this challenge, we employ variability-aware CFGs and reference information. Given is a variability-aware AST ($tunit$), a selection of statements ($lstmt$), a global linking interface (li), and a function name ($fname$), we apply `EXTRACT FUNCTION` as defined in Figure 5.5. First, MORPHEUS validates the accordance of the given function name with the C standard of identifiers (`isValidId`). Second, the engine determines whether $lstmt$ is a valid statement selection for extraction (`isValidSelection`). In particular, MORPHEUS computes a variability-aware CFG and determines whether the selection contains elements that will disrupt the control flow after extraction. To check this property, MORPHEUS traverses the CFG (using the variability-aware successor relation) and ensures that jump targets of problematic code constructs (e.g., **break**, **continue**, and **goto**) belong to the input selection in any variant. Third, as function identifiers can be variable too, MORPHEUS checks $fname$ for violations of C's typing rules within the same module (`isValidInModule`) and across all modules of the program (`isValidInProgram`) for all system variants using the auxiliary function `getPC` to get the common annotation of the selected statements. This check enables turning down the extraction of the selected statements in Figure 5.1d into a function named `main`, as the same symbol is already applied in Line 4. Both checks require SAT checks to determine conflicting declarations. If both checks pass, MORPHEUS replaces $lstmt$ with the appropriate function call ($fcall$ generated with `genFCall`), and introduces the new function declaration ($fdef$ generated with `genFDef`) and its prototype ($fpro$ generated with `genFPro`) with the auxiliary functions `addFDef` and `insertBefore`, respectively.

$\text{extract: } C\text{Program} \times \text{Choice}[\text{AST}] \times \text{List}[\text{Choice}[\text{Stmt}]] \times \text{Id} \rightarrow \text{Choice}[\text{AST}]$

$$\begin{aligned}
&\text{isValidId}(fname) \quad \text{isValidSelection}(tunit, lstmt) \quad (te, ri) = \text{typeCheck}(tunit) \quad pc = \text{getPC}(lstmt) \\
&\text{isValidInModule}(te, tunit, fname) \quad \text{isValidInProgram}(li, \text{Choice}(pc, fname, \text{empty}), fname) \\
&\quad fcall = \text{genFCall}(tunit, lstmt, fname) \quad fdef = \text{genFDef}(tunit, lstmt, fname) \\
&\quad tunit' = \text{replaceStmt}(tunit, lstmt, fcall) \quad tunit'' = \text{addFDef}(tunit', fdef) \\
&\quad fpro = \text{genFPro}(fdef) \quad tunit''' = \text{insertBefore}(tunit'', fdef, fpro) \\
\hline
&\text{extract}(li, tunit, lstmt, fname) \rightarrow tunit'''
\end{aligned}$$

Figure 5.5: Specification of EXTRACT FUNCTION.

Inline Function

Similar to EXTRACT FUNCTION, we need to check control-flow properties and reference information for this refactoring pattern. INLINE FUNCTION requires merging the control flow of the caller function and the callee function, which may result in a disrupted control flow. This merge operation is particularly challenging, because each function call selected to be inlined may have a different context (e.g., available identifiers) and may occur in a different presence condition. To apply INLINE FUNCTION properly, MORPHEUS has to check a set of premises that involve solving SAT problems (cf. Figure 5.6).

At first, the engine uses auxiliary function `isFunctionCall` to validate that the selected identifier (`fcall`) is a function call. After that, MORPHEUS type checks the variability-aware input AST to infer the type environment (`te`) and the reference information (`ri`). The former information is necessary to determine conflicting identifiers of variables that need to be renamed before the function can be inlined. The latter information is necessary to determine all function declarations available for this refactoring. Again, since source code is annotated with presence conditions, a single function call may reference different function declarations that need to be inlined (if possible), including their context. Using the reference information, MORPHEUS determines all depending identifiers (`getModuleReferences`) and separates them afterwards into function calls (`getUses`) and function declarations (`getDefs`).

For each function definition in `fdefs`, the engine makes sure that the function is not recursive (recursive functions cannot be inlined), and that it has a compatible, non-disruptive control flow. If both checks pass, MORPHEUS inlines each function call incrementally (`replaceFCalls`). During each inline operation, the engine repeatedly determines reference information for variables and renames them on demand if they violate C's scoping rules. This occurs especially when multiple function calls that are located next to each other are being inlined. When accessing variability-aware data structures, such as reference information (`ri`), type environment (`te`), and variability-aware CFGs, MORPHEUS solves many SAT problems on demand.

$$\begin{array}{c}
\boxed{\text{inline: Choice[AST]} \times \text{Id} \rightarrow \text{Choice[AST]}} \\
\text{isFunctionCall}(tunit, fcall) \quad (te, ri) = \text{typeCheck}(tunit) \quad rid = \text{getModuleReferences}(ri, fcall) \\
\quad fcalls = \text{getUses}(ri, rid) \quad fdefs = \text{getDefs}(ri, rid) \quad \nexists fd : fd \in fdefs : \text{isRecursive}(tunit, fd) \\
\quad \forall fd : fd \in fdefs : \text{compatibleCFG}(tunit, fd) \quad tunit' = \text{replaceFCalls}(tunit, fcalls, fdefs) \\
\hline
\text{inline}(tunit, fcall) \rightarrow tunit'
\end{array}$$

Figure 5.6: Specification of INLINE FUNCTION.

5.3 Experiments

To show that variability-aware refactoring is feasible in practice, we applied MORPHEUS to the three, real-world subject systems BUSYBOX, OPENSLL, and SQLITE. All three refactorings rely on variability-aware data structures to represent control flow and reference information. To determine whether elements of these data structures are valid in a given context, we employ BDDs and SAT-solving technology. In particular, MORPHEUS validates variant configurations to be satisfiable, contradictory, or tautological. As SAT is NP-complete, a crucial question is whether applying variability-aware refactoring scales in practice. A refactoring engine should be able to process large amounts of source code quickly.

Next, we introduce the three subject systems and our experiment setup. Then, we present our measurement results and reflect on our experience with applying variability-aware refactoring in practice.

5.3.1 Subject Systems

For our experiments, we selected three software systems of substantial size, of which billions of variants can be generated. Besides variability, it was important that each subject system is shipped with a test suite. We reused BUSYBOX and OPENSLL of our previous experiments regarding variability-aware type checking and static analysis in Section 4.2.1. Our third subject system was SQLITE.

SQLITE³ is a library implementing a relational database-management system. The library gained much attention in software development due to a number of desirable properties (e.g., zero-configuration, cross-platform, transactions, and small footprint) and is considered the most widespread database-management system worldwide, with installations as part of ANDROID, MOZILLA FIREFOX, and PHP. To ease embedding SQLITE in other software systems, SQLITE's code base consists of two source-code files (amalgamation version; one header file and one code file). We use a recent version of SQLITE (3.8.1) with 143 614 lines of C code, which can be configured using 93 configuration options (1.02×10^{39} variants).

³<http://sqlite.org>

5.3.2 Experiment Setup

To create a variability-aware AST of C code containing `#ifdef` directives, we used TYPE-CHEF’s variability-aware parsing infrastructure (cf. Section 4.1.2) [Kästner et al., 2011]. Based on the AST representation, we employed TYPE-CHEF’s analysis facilities for type checking and control-flow analysis (cf. Section 4.1.2) [Kästner et al., 2012b; Liebig et al., 2013]. After parsing and analyzing the C code, we applied the three refactorings (RENAME IDENTIFIER, EXTRACT FUNCTION, and INLINE FUNCTION) to each file of all systems. As refactoring tasks we selected code fragments (an identifier, a list of statements, and a function call) randomly and applied the appropriate refactoring. Since we wanted to measure the effect of variability, we preferably selected code fragments that contain variability. That is, MORPHEUS analyzed the variable input AST (`Choice [AST]`) and automatically scanned its nodes for elements that have a non-trivial presence condition (different from *true/false*). Our refactorings do still work when the selected code fragments do not contain any variability. In this case, the created variability-aware data structures remain invariable (the presence conditions of `Choice [A]` elements are all *true*) and no satisfiability problems arise. As a result, MORPHEUS behaves like a standard refactoring engine. Finally, for rigid performance tests, we parametrize each refactoring pattern based on a previous test setup for refactoring engines as follows [Gligoric et al., 2013]:

Rename Identifier

For each file, we randomly selected up to 50 identifiers with presence conditions (e.g., function names, local or global identifiers, user-defined data types), and renamed them using a predefined name. When the renaming had a global effect, we employed a global module interface for consistent renamings of depending identifiers in other files (cf. Section 5.2.1). Overall, we renamed 5832 identifiers of BUSYBOX, 5186 of OPENSSL, and 50 of SQLITE.

Extract Function

For each file, we attempted to extract one list of statements and put them into a function with a predefined name. To this end, we randomly selected sequences of statements (up to 100 selections similar to the case study of Gligoric et al. [2013]) from a function’s implementation that preferably contained, at least, one statement with a non-trivial presence condition. Many files contain only a small number of functions with a small function body, for which we were not able to apply EXTRACT FUNCTION. Overall, we extracted 61 functions of BUSYBOX, 172 of OPENSSL, and 1 of SQLITE.

Inline Function

Much like EXTRACT FUNCTION, INLINE FUNCTION is not always applicable, and, therefore, for each file MORPHEUS scanned the source code for possible function definitions that can be inlined and that preferably contained variability. Overall, we inlined 50 functions of BUSYBOX,

126 of OPENSLL, and 1 of SQLITE.

While checking all premises and applying all transformation operations (cf. Equations 5.4 to 5.6), MORPHEUS solves many SAT problems. In particular, accesses to the global module interface, reference information, and control-flow require many satisfiability checks. To avoid expensive satisfiability checks, we cache the outcome of satisfiability problems that are already solved. This strategy was successfully applied in variability-aware analyses in the past [Apel et al., 2010a,b]. MORPHEUS and the underlying parsing and analysis infrastructure do also make use of caching, and we are interested in knowing whether variability-aware refactorings can benefit from caching, too.

5.3.3 Performance Results

We ran the experiments for OPENSLL on a LINUX machine with AMD Opteron 8356 CPUs, 2.3 GHz, and 64 GB RAM. For BUSYBOX and SQLITE, we used a LINUX machine with a Intel Core2 Quad Q6600 CPU, 2.4 GHz, and 8 GB RAM. We configured the Java JVM with 2 GB RAM for memory allocation.

In Table 5.2, we show the measurement results for each refactoring and subject system. We report refactoring times and affected configurations with the mean \pm standard deviation as well as the maximum for a single refactoring task. Additionally, we use box-plots to visualize the distribution of time measurements and the number of affected configurations of the refactoring results per subject system. The numbers do not include times for parsing and type checking the input C code, because both tasks are usually done in background processing in IDEs, and their results (AST and reference information) are used for other tasks beside refactoring (e.g., syntax highlighting and static analysis for error checking).

Overall, the results show that variability-aware refactoring is feasible in practice. For BUSYBOX and OPENSLL, the refactoring times are less than one second (RENAME IDENTIFIER and EXTRACT FUNCTION), on average. Applying INLINE FUNCTION is a little more expensive, since we need to update reference information each time we inline a function call.

For SQLITE, the results are different. Due to the nature of the subject system (in particular, the fact that the source code was merged in a single file), a refactoring on the resulting AST takes a comparatively large amount of time. But, this is *not* caused by variability, nor by shortcomings of our approach. A single INLINE FUNCTION may take up to 85 seconds, on maximum, because functions selected for an inlining task are called up to a hundred times in the source code and for each inlined function call, the variability-aware data structures, e.g. reference information, need to be traversed and updated; two operations that have not been optimized by us.

5.3.4 Testing the Refactoring Engine

To validate that our refactorings are behavior-preserving, we employed a standard testing approach for refactoring engines [Daniel et al., 2007; Gligoric et al., 2013]. To this end, we







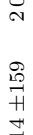
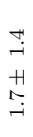

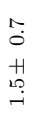
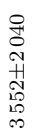
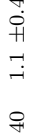

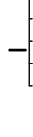
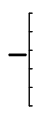
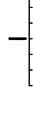
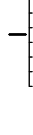
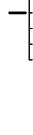
System	RENAME IDENTIFIER			EXTRACT FUNCTION			INLINE FUNCTION			
	time in ms	mean \pm sd	# configs	time in ms	mean \pm sd	# configs	time in ms	mean \pm sd	# configs	
BUSYBOX	72 \pm 77.8	15.9 \pm 20.5	52	410 \pm 99.6	10.6 \pm 18.1	52	4 049 \pm 2 976	19 282	4.1 \pm 8.2	52
										
OPENSSL	114 \pm 159	1.7 \pm 1.4	11	1 065 \pm 316	1.5 \pm 0.7	7	3 552 \pm 2 040	13 140	1.1 \pm 0.4	3
										
SQLITE	476 \pm 98.4	56 \pm 0	56	1 350 \pm 0	4 \pm 0	4	85 378 \pm 0	85 378	8 \pm 0	8
										

Table 5.2: Measurement results for RENAME IDENTIFIER, EXTRACT FUNCTION, and INLINE FUNCTION for all three subject systems (BUSYBOX, OPENSSL, and SQLITE); for each refactoring we give the mean \pm standard deviation, the maximum of refactoring times in milliseconds, and the number of affected configurations; box-plots show the corresponding distributions (excluding outliers).

used two test oracles that automatically checked whether a refactoring’s code transformation was correct. Our oracles were: (1) the source code of our subject systems still compiles and (2) the results of all system tests (post-refactoring vs pre-refactoring) do not vary. In contrast to existing test suites, which do not incorporate variability, we determined which system configurations are affected and tested them against both oracles automatically. This way we ensured that MORPHEUS did not introduce any variability-related errors to a system’s code base.

For our three subject systems, we used the following system tests:

- BUSYBOX comes with a test suite of 410 single test cases in 74 files, of which 46 tests fail (which we ignored during our evaluation). Each test checks the correctness of a single component of BUSYBOX’ tool collection.
- OPENSSEL’s test suite is delivered with its source and checks individual components of OPENSSEL’s implementation, including the correct implementation of hashing algorithms (e.g., MD5 and SHA-256), key-generation algorithms (e.g., NIST prime-curve P-192), as well as message encryption and message decryption with cryptographic algorithms. The test suite runs as a whole, and test success is indicated with ‘*ALL TESTS SUCCESSFUL*’ output.
- For testing SQLITE, we used the proprietary test suite TH3.⁴ TH3 is a test program generator and provides a full coverage test of the compiled object code.

The test setup was as follows. During the application of a refactoring task, MORPHEUS determined all affected presence conditions, based on accesses to variability-aware data structures. For example, renaming the variable `global` in Line 8 in Figure 5.1a has an impact on the presence conditions A and $\neg A$. After collecting affected presence conditions for all refactoring tasks (e.g., applying up to 50 times RENAMING REFACTORING), MORPHEUS computes valid system configurations for the configurable system. For each configuration, the engine compiles and applies all system tests before and after the refactoring and compares their results.

During our experiments, our selected test oracles did not show a difference before and after a refactoring’s application. That is, MORPHEUS did not introduce any variability-related bugs when applying a refactoring’s transformations.

5.3.5 Perspectives of Variability-aware Refactoring

Despite the encouraging results, there are two engineering issues that need to be solved before MORPHEUS can be applied in an industrial setting. First, the parsing infrastructure TYPECHEF applies partial preprocessing of source code before the parser creates the variability-aware AST [Kästner et al., 2011; Kenner et al., 2010]. That is, the preprocessor directives `#define` and `#include` are resolved using automatic macro expansion and file inclusion. Both resolutions are necessary to be able to parse C code at all, as both directives manipulate the token stream—even conditionally if annotated with `#ifdef` directives—that serves as the input to the parser. As a result, the recreation (i.e., pretty printing) of source code of the variable-

⁴<http://sqlite.org/th3.html>

aware ASTs requires the additional effort of reversing macro expansion and file inclusion. Existing refactoring engines, such as CREFACTORY, preserve partial-preprocessing information in AST nodes so that the pretty printer can use them when recreating source code. This approach can also be used in MORPHEUS to support the entire process (parsing→refactoring→pretty-printing).

Second, setting up TYPECHEF and MORPHEUS for a new software system is a non-trivial task. The main burden is the creation of a proper setup for the parsing, type-checking, and linking infrastructure. As TYPECHEF solely works on the basis of C files, the possibly complex setup of a software system (e.g., configuration scripts, library dependencies, and build-system setup) has to be made explicit to enable successful work on a system's source code. In principle, it is possible to use TYPECHEF as a compiler replacement, with the downside that a user has to specify additional information for variability (e.g., which configuration options should be considered variable and which dependencies exist between configuration options). So far, we make use of additional tools for the extraction of configuration knowledge from build systems [Tartler et al., 2011; Berger et al., 2010a] and configuration models [Berger et al., 2010b]. Further approaches, such as the automatic interference of a configurable system's configuration knowledge [Nadi et al., 2014], will simplify the project setup further.

We have not exploited every option to speed up the transformation process yet. In particular, there is one optimization possibility that is likely to improve the refactoring times substantially. With the idea of continuous development of a software system in mind, there is some potential for the improvement of transformation times by reusing analysis results in subsequent transformations. We can facilitate reuse by using a persistent storage for analysis results, such as type-checking, linking, and control-flow information. Using such a cache, it is sufficient to recompute information that changed between consecutive transformation runs by inferring the delta between the versions in question. Along the same line, we can reuse the results of solving SAT problems in subsequent transformation runs. At the moment, we only cache SAT-solver results within a single refactoring experiment.

5.3.6 Threats to Validity

First of all, our selection of only three subject systems threatens external validity, because refactoring tasks in these systems may be particularly easy, and a significant performance loss may only occur when using MORPHEUS with different software systems. We do not consider this to be a problem, since we selected systems with a significant code base (containing several thousand lines of source code), which have been developed over decades by many different developers, and which are well-received in practice. Furthermore, from our experience, the key performance factor of variability-aware refactoring is the time for solving satisfiability problems. In general, the time to resolve a single SAT call depends on the number of configuration options and the number of dependencies between them. While existing systems usually have many configuration options [Liebig et al., 2010], the number of option dependencies is usually small so that satisfiability problems can be solved efficiently. Additionally, recent advances in SAT-solver technology enable the solution of large problems with thousands of configuration

options efficiently [Thüm et al., 2009].

Second, for our experiments, we could not rely on existing refactoring tasks, which one could extract from the system’s version history (construct validity). Hence, our tasks may not be representative of practical refactoring tasks applied by developers in the wild. However, we believe that our large random selection of code fragments for refactoring tasks largely compensates this threat, and that the results provide a reasonable overview of the refactoring performance in practice.

Third, the application of the selected refactoring patterns may be particularly easy to apply, even for configurable systems, and as such, they may not be representative for refactoring in general. We focussed on the three refactoring patterns `RENAME IDENTIFIER`, `EXTRACT FUNCTION`, and `INLINE FUNCTION`, because they are among the most important refactorings that developers use in practice [Murphy-Hill et al., 2012]. Furthermore, other refactorings, such as `DELETE UNREFERENCED VARIABLE` or `REORDER FUNCTION PARAMETERS`, have a similar complexity, so, we expect a similar performance. For example, for `DELETE UNREFERENCED VARIABLE`, we only need to traverse reference information and determine for each definition of a variable if it has corresponding uses. In the same vein, `REORDER FUNCTION PARAMETERS` requires reference information of a function definition to its corresponding function calls. Based on this information, we can easily reorder parameters and arguments accordingly, and we only need to incorporate the SAT solver if presence conditions of function arguments and function parameters differ in the function definitions and the function calls.

Last, one technical problem is that the variability-aware parsing and analysis infrastructure `TYPECHEF` does not fully support the ISO/IEC standard for C (external validity). In particular, the infrastructure implements only a subset of the C standard and extensions of GNU C that are used in the subject systems. During analysis, `TYPECHEF` ignores unsupported constructs, so we had to exclude 6 files for `BUSYBOX` and 11 for `OPENSSL` in our experiments. Furthermore, we changed the source code of 9 files that was not in line with the C standard. In particular, we transformed incorrect representations of hexadecimal numbers to correct ones (e.g., `\x8` to `\x08`). For `OPENSSL`, we had to exclude 127 identifiers (e.g., references to unsupported function-pointer constructs). `SQLITE`’s source code remained untouched. All numbers and plots in this thesis were generated after excluding problematic files. However, we believe that our experiments are representative for variability-aware refactoring and show that our approach is feasible in practice.

5.4 Related Work

Beside refactoring, which we already discussed in Section 5.1, there are three areas of related work: variability-aware code transformations, testing refactoring engines, and testing configurable systems.

Variability-aware Code Transformations

As researchers often discourage the use of preprocessors for the development of configurable systems [Kästner, 2010], they have proposed alternative implementation mechanisms, such as aspects [Kiczales et al., 1997] and feature modules [Batory et al., 2004], for variability implementation. There has been some effort to transform preprocessor-based implementations into aspects [Reynolds et al., 2008; Bruntink et al., 2007; Adams et al., 2009] and feature modules [Kästner et al., 2009a]. This transformation usually rests on the identification of typical patterns of preprocessor use [Adams et al., 2009; Liebig et al., 2010] and the definition of appropriate transformation rules for code extraction [Kästner et al., 2009a]. Prior to code extractions, developers often have to prepare the source code manually [Reynolds et al., 2008], which however can partially be automated by using MORPHEUS. While the objectives of variability-aware transformations (e.g., separation of concerns) are different, the variant-preservation challenge is the same. To ensure behavior preservation, developers sometimes use run-time tests [Lohmann et al., 2006; Kästner et al., 2009a], but in general, they do not employ a systematic testing approach as we do.

Testing Refactoring Engines

Testing is a common approach to detect errors in a refactoring engine's implementation [Daniel et al., 2007; Soares et al., 2013; Gligoric et al., 2013]. Existing approaches usually generate input programs [Soares et al., 2013; Daniel et al., 2007] or use real software projects as test input [Gligoric et al., 2013]. Testing procedures usually involve an automatic check of one or more test oracles. Our testing approach incorporates variability: we check test oracles not only for a single configuration, but for all configurations that have been affected by a refactoring.

Testing Configurable Systems

Since the number of valid configurations of a system can exponentially grow with the number of configuration options, testing all variants individually is elusive [Thüm et al., 2014]. We and others have employed sampling (i.e., reducing the number of configurations to an interesting subset by means of a given sampling heuristic) to take variability into account [Liebig et al., 2013]. Sampling has been applied successfully in different contexts, enabling the detection of errors in a reasonable amount of time. Fortunately, variability-aware refactoring has mostly a local effect on source code. That is, in our subject systems, `RENAME IDENTIFIER`, `EXTRACT FUNCTION`, and `INLINE FUNCTION` usually affected only a couple of configuration options, which enabled exhaustive testing of all variants. Nevertheless, dynamic testing approaches, which employ an analysis of execution traces to infer affected configurations (e.g., SPLAT [Kim et al., 2013]), could reduce the number of configurations to be tested further.

5.5 Summary

Refactoring C code with preprocessor directives is challenging, because a single refactoring may affect not only a single but a multitude of system variants that can be derived from a configurable system. We proposed an approach for scalable refactoring of C code with preprocessor directives, accompanied by a specific refactoring-engine implementation named MORPHEUS. A comparative analysis of state-of-the-art refactoring engines for C revealed that most refactoring engines suffer from one of several shortcomings. They cannot handle variability at all, provide only limited support for variability-aware refactoring, or make use of unsound heuristics. Based on variability-aware data structures and static analysis, we specified and implemented sound variability-aware instances of common refactorings (RENAME IDENTIFIER, EXTRACT FUNCTION, and INLINE FUNCTION). We demonstrated the feasibility of variability-aware refactoring with our variability-aware refactoring engine MORPHEUS and applied it to the three real-world systems BUSYBOX, OPENSBL, and SQLITE with a total number of 11 479 refactorings. Our empirical results show that MORPHEUS performs well, especially compared to the state of the art. The average transformation time is in the order of milliseconds, performing sound refactorings on real, variable C code.

To verify that our refactorings are indeed semantics-preserving, we extended a standard testing approach with support for variability. We were able to show that all variants of our subject systems still compiled and conformed to the systems' test suites after applying the refactorings.

On top of our interface specification of variability-aware analysis and transformation, further refactorings, such as DELETE UNREFERENCED VARIABLE or REORDER FUNCTION PARAMETERS, are possible, making our refactoring engine MORPHEUS an ideal test bed for experiments of variability-aware refactoring for other researchers. Overall, we demonstrated that sound refactoring engines for C including the C preprocessor are in reach, and that variability-aware refactorings, including analysis and transformation, scale to substantial, real-world software systems. MORPHEUS closes a gap in tool support for the development of software systems written in C with CPP. Such tool support, which has been available for other mainstream languages, such as Java or C#, for some time, simplifies the development of C code significantly and increases the confidence in a software system's development process.

6 Conclusion and Future Work

Tool support for efficient analysis and transformation of source code is an important asset for developers when implementing software systems. To facilitate reuse, in practice, many software systems are developed as configurable systems to support different requirements of application scenarios and use cases. Developers of configurable systems often encode the corresponding variability with compile-time configuration options, enabling end users to create a specific variant of the configurable system tailored to their requirements. That is, given a set of configuration options, an end user can activate/deactivate configuration options and so enable/disable optional and alternative functionalities in the source code. Using this approach, software developers often create configurable systems that enable the generation of billions of system variants.

While configurable systems are beneficial from the point of view of software customization, existing tool support is insufficient for the wealth-of-variants challenge in configurable systems' development. This is because existing analysis and transformation approaches target the development of single system variants, including only rudimentary (and often broken) tool support for configurable systems. As a consequence, developing configurable systems becomes prone to programming errors.

To improve the current situation, we developed analysis and transformation techniques with direct support for variability. With such tool support variability in the source code is no longer neglected, thus, enabling efficient reasoning about source code even in the presence of possibly billions of variants.

Specifically, we made the following contributions:

1. In chapter *Understanding Preprocessor Annotations*, we investigated basic information required to develop efficient program-analysis and program-transformation tools in order to support the development of configurable systems with CPP. By means of a set of metrics, we captured invaluable information about configuration knowledge in a system's code base. With a study of 42 open-source software systems, we identified the current practice of CPP usage in practice and, given the requirements of tool support, inferred guidelines for preprocessor usage. Existing systems often provide hundreds of configuration options, which are scattered across the entire variable code base to implement system variability.

Although the annotation concept is easy to understand, developers sometimes write complex `#ifdef` code. One particular form of complexity is the nesting of `#ifdef` directives, which developers use moderately in most cases and which reaches a hard-to-manage maximum of 24 nested `#ifdefs` only in rare cases.

Functions, structure definitions, and statements are the most frequently annotated elements in the source code, and expression annotations or function-parameter annotations occur only infrequently. This is particularly interesting, because it bears the potential of `#ifdef`-based variability being transformed to alternative variability representations. In this context, we observed that developers often annotate reasonable elements of the programming language C (disciplined annotations) and usually avoid undisciplined annotations (annotations of single tokens, such as an opening brace). The main benefit of disciplined annotations is that they enable the development of sound tool support (variability-aware program analyses and program transformations). Based on a study of 30 million lines of C code, we were able to show that enforcing preprocessor discipline is justifiable and does not place a significant burden on software developers.

2. By comparing state-of-the-art analysis techniques for configurable systems (sampling-based analysis vs variability-aware analysis) with respect to three criteria (analysis time, upfront investment, and error coverage), we were able to show that variability-aware analysis is competitive to (and sometimes even outperforms) sampling heuristics (which represent the state of the art in practice) when *Analyzing C Code with Preprocessor Annotations*. While being more expensive to apply, the additional effort of variability-aware analysis due to solving SAT problems plays a minor role compared to repetitive computations of analysis results for similar variants when using sampling-based analysis. The break-even point at which variability-aware analysis is faster than sampling is between two and three variants in our experience.

In our studies, we identified three patterns that can serve as a guideline for developers and researchers when designing and implementing scalable analyses for configurable systems. The late-splitting pattern entails that an analysis remains variability-unaware until variability is encountered in the input representation. If that is the case, an analysis splits up into several analysis processes, each handling one of the varying inputs individually. When a common point in the input representation is reached, the early-joining pattern joins individual analyses' results that stem from a previous split. For efficient handling of analysis results, the local variability-representation pattern defines compact and redundancy-free representations of analysis results. Based on the three patterns, we enriched the intra-procedural data-flow analysis framework (MONOTONE FRAMEWORKS) with variability. The enriched frameworks enable the rapid development of static analyses for program optimization and error detection in configurable systems.

On top of the variability-aware CFGs and MONOTONE FRAMEWORKS, we implemented eight variability-aware static analyses for error detection. Not only were we able to demonstrate their usefulness of detecting real bugs in LINUX, we also evaluated that in terms of error coverage sampling-based analysis cannot measure up to the standard of variability-aware analysis.

3. Based on our results on variability-aware data structures and variability-aware static analysis, we developed the variability-aware refactoring engine MORPHEUS (Chapter

Refactoring C Code with #ifdefs). The engine provides three common refactorings for C (RENAME IDENTIFIER, EXTRACT FUNCTION, and INLINE FUNCTION). In contrast to previous refactoring engines, MORPHEUS incorporates configuration knowledge during the transformation process. So our engine respects all system variants that can possibly be derived from a configurable system. Despite the increased transformation effort, MORPHEUS is still very fast and provides an instantaneous user experience. To validate system correctness, we enriched a standard testing approach for refactoring engines with variability and proved that MORPHEUS is behavior-preserving.

With variability-aware analysis and transformation we can detect and avoid programming errors early in the development process. The major benefit is that developers can guarantee system correctness of all variants to customers before the system is being delivered.

Future Work

So far, our assessment of configuration options regarding variability and complexity as well as the analysis and transformation of `#ifdef`-based source code have worked on a single version of a configurable system only. That is, our results represent only a snapshot of the current state of the system's implementation. Naturally, configurable systems evolve; so using our techniques and specific tool implementations is beneficial for continuous development, too. Regarding the assessment of configuration options, developing system variants in a configurable system is more complex than developing an unconfigurable system. In this vein, Zhang et al. [2013] evaluated our complexity metrics to determine the degeneration of a configurable system's implementation. In particular, they measured successive versions of a configurable system from industry and predicted future values for the metrics. According to the authors, the metrics were helpful to detect problematic spots in the source code, and analysis results triggered corresponding refactorings in the past. Future work should investigate this direction in more depth in order to validate the effectiveness of our proposed metrics in real software systems with experienced developers.

Although we were able to demonstrate that variability-aware tool support scales to large-scale systems, there is still room for improvement. So far, we have recomputed variability-aware data structures and static-analysis results again and again. With the idea of continuous integration and software evolution in mind, we should adapt variability-aware analysis and transformation to changes of the evolving system. An interesting avenue for future work is to determine efficient recomputation strategies for analysis and transformation results based on code-artifact changes. The resulting incremental variability-aware analysis and transformation techniques will most likely improve the performance further, and tame even fast growing configurable systems, such as the LINUX kernel [Tartler et al., 2011].

Our implementation of variability-aware static analysis and refactoring paves the way for a variety of applications, which can be applied to many large-scale, real-world systems for the first time. In particular, variability-aware control-flow graphs and variability-aware MONOTONE

FRAMEWORKS are valuable assets for the development of further static analyses. Extending intra-procedural to inter-procedural data-flow analysis can help to detect more programming errors in existing configurable systems. Our implementation of variability-aware control-flow graphs enables such extensions, but the main question is whether inter-procedural analysis scales to realistic systems, too. There is some experience with inter-procedural analysis in the context of configurable systems written in Java (using a different preprocessor) [Bodden et al., 2013]. The authors enriched an inter-procedural data-flow analysis framework with variability, enabling the seamless analysis of variant-rich systems. A similar framework for C source code with `#ifdefs` would help to detect programming errors across functions and, hence, improve programming-error detection even further.

We provided evidence that variability-aware data structures are beneficial for analyzing and transforming configurable systems developed with compile-time variability (with preprocessors, aspects, or feature modules). However, sometimes a transformation between different variability representations is beneficial, if developers change the programming paradigm in a system [Adams et al., 2009; Lohmann et al., 2009] or if tool support is not yet available. There were some attempts to transform `#ifdef`-annotated source code into alternative representations [Adams et al., 2009; Lohmann et al., 2009; Post and Sinz, 2008]. However, being limited to general proof-of-concept studies [Adams et al., 2009; Lohmann et al., 2009] or by being limited to the use of tedious and error-prone manual transformations [Post and Sinz, 2008], a (semi-)automatic solution can supersede the necessity to stick to a particular development approach. We have already started to pursue an automatic transformation of compile-time variability with `#ifdefs` to run-time variability with `if` statements, as proposed by Post and Sinz [2008]. The idea of this process, called *variability encoding* [Apel et al., 2011], is to create a single variant (also known as *product simulator*), which is able to simulate all system variants. The resulting product simulator can be analyzed with existing tool support directly. Hence, it does not place a burden on developers adapting an existing analysis approach to make it variability-aware. The major challenge of variability encoding is to preserve as much sharing between system variants as possible, so that an analysis tool can still analyze the simulator efficiently. While early results are promising, there are still a few engineering issues left regarding the transformation.

Our contributions have the potential to change the way configurable systems are going to be developed in the future. Variability-aware tool support brings configurable-system development in line with the development of unconfigurable systems. Scalable, variability-aware analysis techniques and transformation techniques not only enable efficient development, but also enable changing variability representations automatically, thus, making different development approaches for variability interchangeable at will.

Bibliography

- I. Abal, C. Brabrand, and A. Wąsowski. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 421–432. ACM, 2014.
- M. Acher, P. Heymans, P. Collet, C. Quinton, P. Lahire, and P. Merle. Feature Model Differences. In *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 629–645. Springer, 2012.
- B. Adams, W. De Meuter, H. Tromp, and A. Hassan. Can we Refactor Conditional Compilation into Aspects? In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 243–254. ACM, 2009.
- A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson, 2006.
- S. Apel. *The Role of Features and Aspects in Software Development*. PhD thesis, University of Magdeburg, 2007.
- S. Apel. How AspectJ is Used: An Analysis of Eleven AspectJ Programs. *Journal of Object Technology*, 9(1):117–142, 2010.
- S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(5):49–84, 2009.
- S. Apel and C. Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *Proceedings of the International Symposium on Software Composition (SC)*, pages 20–35, 2008.
- S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Transactions on Software Engineering*, 34(2):162–180, 2008.
- S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-Independent, Automated Software Composition. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 221–231. IEEE, 2009.
- S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type Safety for Feature-Oriented Product Lines. *Automated Software Engineering*, 17(3):251–300, 2010a.

- S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Language-Independent Reference Checking in Software Product Lines. In *Proceedings of the International Workshop Feature-Oriented Software Development (FOSD)*, pages 65–71. ACM, 2010b.
- S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of Feature Interactions using Feature-Aware Verification. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 372–375. IEEE, 2011.
- S. Apel, S. Kolesnikov, J. Liebig, C. Kästner, M. Kuhlemann, and T. Leich. Access Control in Feature-Oriented Programming. *Science of Computer Programming (SCP)*, 77(3):174–187, 2012.
- S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013a.
- S. Apel, C. Kästner, and C. Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Transactions on Software Engineering*, 39(1):63–79, 2013b.
- S. Apel, S. Kolesnikov, N. Siegmund, C. Kästner, and B. Garvin. Exploring Feature Interactions in the Wild: The New Feature-interaction Challenge. In *Proceedings of the International Workshop Feature-Oriented Software Development (FOSD)*, pages 1–8. ACM, 2013c.
- S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 482–491. IEEE, 2013d.
- D. Atkins, T. Ball, T. Graves, and A. Mockus. Using Version Control Data to Evaluate the Impact of Software Tools: A Case Study of the Version Editor. *IEEE Transactions on Software Engineering*, 28(7):625–637, 2002.
- L. Aversano, L. Di Penta, and I. Baxter. Handling Preprocessor-Conditioned Declarations. In *Proceedings of the Working Conference on Source Code Management and Manipulation (SCAM)*, pages 83–92. IEEE, 2002.
- G. Badros and D. Notkin. A Framework for Preprocessor-Aware C Source Code Analyses. *Software: Practice and Experience*, 30(8):907–924, 2000.
- T. Ball and S. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*, pages 1–3. ACM, 2002.
- V. Basili, G. Caldiera, and D. Rombach. *The Goal Question Metric Approach*, volume 1, pages 528–532. John Wiley & Sons, 1994.

-
- L. Bass, P. Clements, S. Cohen, L. Northrop, and J. Withey. Product Line Practice Workshop Report. Technical Report CMU/SEI-97-TR-003, Software Engineering Institute, 1997.
- L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003.
- D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 7–20. Springer, 2005.
- D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.
- I. Baxter and M. Mehlich. Preprocessor Conditional Removal by Simple Partial Evaluation. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 281–290. IEEE, 2001.
- I. Baxter, C. Pidgeon, and M. Mehlich. DMS[®]: Program Transformations for Practical Scalable Software Evolution. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 625–634. IEEE, 2004.
- D. Benavides. *On the Automated Analysis of Software Product Lines using Feature Models: A Framework for Developing Automated Tool Support*. PhD thesis, University of Seville, 2007.
- D. Benavides, S. Segura, P. Martín-Arroyo, and A. Cortés. Using Java CSP Solvers in the Automated Analyses of Feature Models. In *Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, pages 399–408. Springer, 2005.
- D. Benavides, S. Segura, and A. Ruiz-Corés. Automated Analysis of Feature Models 20 Year Later: A Literature Review. *Information Systems*, 35(6):615–636, 2010.
- T. Berger. *Variability Modeling in the Real*. PhD thesis, University of Leipzig, 2013.
- T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wasowski. Feature-to-Code Mapping in Two Large Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 498–499. Springer, 2010a.
- T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. Variability Modelling in the Real: A Perspective from the Operating Systems Domain. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 73–82. ACM, 2010b.
- A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.
- T. Biggerstaff. A Perspective of Generative Reuse. *Journal of Annals of Software Engineering*, 1(5):169–226, 1998.

- E. Bodden, M. Mezini, C. Brabrand, T. Tolêdo, M. Ribeiro, and P. Borba. SPL^{LIFT} — Statically Analyzing Software Product Lines in Minutes Instead of Years. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 355–364. ACM, 2013.
- Q. Boucher, A. Classen, P. Heymans, A. Bourdoux, and L. Demonceau. Tag and Prune: A Pragmatic Approach to Software Product Line Implementation. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 333–336. ACM, 2010.
- C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba. Intraprocedural Dataflow Analysis for Software Product Lines. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 13–24. ACM, 2012.
- C. Brabrand, M. Ribeiro, T. Tolêdo, J. Winther, and P. Borba. Intraprocedural Dataflow Analysis for Software Product Lines. *Transactions on Aspect-Oriented Programming*, 10:73–108, 2013.
- M. Bruntink, A. van Deursen, M. D’Hondt, and T. Tourwé. Simple Crosscutting Concerns Are Not So Simple: Analysing Variability in Large-Scale Idioms-Based Implementations. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 199–211. ACM, 2007.
- M. Calder and A. Miller. Feature Interaction Detection by Pairwise Analysis of LTL Properties: A Case Study. *Formal Methods in System Design*, 28(3):213–261, 2006.
- L. Chen, M. Babar, and N. Ali. Variability Management in Software Product Lines: A Systematic Review. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 81–90. Carnegie Mellon University, 2009.
- S. Chen, M. Erwig, and E. Walkingshaw. Extending Type Inference to Variational Programs. *ACM Transactions on Programming Languages and Systems*, 36(1):1–54, 2014.
- S. Cherem, L. Princehouse, and R. Rugina. Practical Memory Leak Detection Using Guarded Value-flow Analysis. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 480–491. ACM, 2007.
- E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided Abstraction Refinement for Symbolic Model Checking. *Journal of the ACM*, 50(5):752–794, 2003.
- A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 335–344. ACM, 2010.

- A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic Model Checking of Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 321–330. ACM, 2011.
- Z. Coker, S. Hasan, J. Overbey, M. Hafiz, and C. Kästner. Integers In C: An Open Invitation to Security Attacks? Technical Report CSSE14-01, College of Engineering, Auburn University, 2014.
- B. Cole, D. Hakim, D. Hovemeyer, R. Lazarus, W. Pugh, and K. Stephens. Improving Your Software Using Static Analysis to Find Bugs. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 673–674, 2006.
- M. Collard, M. Decker, and J. Maletic. srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 516–519. IEEE, 2013.
- A. Colyer and A. Clement. Large-Scale AOSD for Middleware. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 56–65. ACM, 2004.
- K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- K. Czarnecki and A. Wąsowski. Feature Diagrams and Logics: There and Back Again. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 23–34. IEEE, 2007.
- K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wąsowski. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 171–182. ACM, 2012.
- B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated Testing of Refactoring Engines. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 185–194. ACM, 2007.
- C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann. A Robust Approach for Variability Extraction from the Linux Build System. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 21–30. ACM, 2012a.
- C. Dietrich, R. Tartler, W. Schröder-Preikshat, and D. Lohmann. Understanding Linux Feature Distribution. In *Proceedings of the AOSD Workshop on Modularity in Systems Software (MISS)*, pages 15–20. ACM, 2012b.

- D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks. Software Architecture Themes in JPL's Mission Data System. In *Proceedings of Aerospace Conference*, pages 259–268. IEEE Aerospace Conferences, 2000.
- C. Elsner, D. Lohmann, and W. Schröder-Preikschat. Fixing Configuration Inconsistencies Across File Type Boundaries. In *Proceedings of the EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 116–123. IEEE, 2011.
- M. Ernst, G. Badros, and D. Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002.
- M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Transactions on Software Engineering and Methodology*, 21(1):6:1–6:27, 2011.
- M. Erwig, E. Walkingshaw, and S. Chen. An Abstract Representation of Variational Graphs. In *Proceedings of the International Workshop Feature-Oriented Software Development (FOSD)*, pages 25–32. ACM, 2013.
- J.-M. Favre. The CPP Paradox. In *Proceedings of the European Workshop Software Maintenance*, 1995.
- J.-M. Favre. Preprocessors from an Abstract Point of View. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 329–339. IEEE, 1996.
- J.-M. Favre. Understanding-In-The-Large. In *Proceedings of the International Workshop on Program Comprehension (IWPC)*, pages 29–38. IEEE, 1997.
- M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- D. Ganesan, M. Lindvall, C. Ackermann, D. McComas, and M. Bartholomew. Verifying Architectural Design Rules of the Flight Software Product Line. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 161–170. Carnegie Mellon University, 2009.
- A. Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, University of Illinois, 2005.
- A. Garrido and R. Johnson. Refactoring C with Conditional Compilation. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 323–326. IEEE, 2003.
- A. Garrido and R. Johnson. Analyzing Multiple Configurations of a C Program. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 379–388. IEEE, 2005.

- B. Garvin and M. Cohen. Feature Interaction Faults Revisited: An Exploratory Study. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 90–99. IEEE, 2011.
- P. Gazillo and R. Grimm. SuperC: Parsing All of C by Taming the Preprocessor. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 323–334. ACM, 2012.
- M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov. Systematic Testing of Refactoring Engines on Real Software Projects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 629–653. Springer, 2013.
- W. Gong and H.-A. Jacobsen. AspeCt-Oriented C Language Specification, January 2008. Version 0.8; <http://www.aspectc.net>.
- M. Hafiz and J. Overbey. OpenRefactory/C: An Infrastructure for Developing Program Transformations for C Programs. In *Proceedings of the Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH)*, pages 27–28. ACM, 2012.
- M. Hafiz, J. Overbey, F. Behrang, and J. Hall. OpenRefactory/C: An Infrastructure for Building Correct and Complex C Transformations. In *Proceedings of Workshop on Refactoring Tools (WRT)*, pages 1–4. ACM, 2013.
- F. Heidenreich, I. Savga, and C. Wende. On Controlled Visualizations in Software Product Line Engineering. In *Proceedings of the International Workshop on Visualisation in Software Product Line Engineering (ViSPL)*, pages 303–313. Lero International Science Centre, University of Limerick, 2008.
- C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-wise Test Configurations for Software Product Lines. *IEEE Transactions on Software Engineering*, 40(7):650–670, 2014.
- M. Hind. Pointer Analysis: Haven’t We Solved This Problem Yet? In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 54–61. ACM, 2001.
- Y. Hu, E. Merlo, M. Dagenais, and B. Lagüe. C/C++ Conditional Compilation Analysis using Symbolic Execution. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 196–206. IEEE, 2000.
- A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

- D. Janzen and K. De Volder. Programming with Crosscutting Effective Views. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 195–218. Springer, 2004.
- H. Jepsen and D. Beuche. Running a Software Product Line: Standing Still is Going Backwards. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 101–110. ACM, 2009.
- M. Johansen. *Testing Product Lines of Industrial Size: Advancements in Combinatorial Interaction Testing*. PhD thesis, University of Oslo, 2013.
- M. Johansen, Ø. Haugen, and F. Fleurey. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 638–652. Springer, 2011.
- M. Johansen, Ø. Haugen, and F. Fleurey. An Algorithm for Generating t-Wise Covering Arrays from Large Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 46–55. ACM, 2012.
- B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why Don't Software Developers Use Static Analysis Tools to Find Bugs? In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE Software, 2013.
- R. Johnson and B. Foote. Designing Reuseable Classes. *Journal of Object-Oriented Programming*, 2(1):22–35, 1988.
- E. Jürgens, F. Deissenböck, B. Hummel, and S. Wagner. Do Code Clones Matter? In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 485–495. IEEE, 2009.
- K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, 1990.
- C. Kapser and M. Godfrey. “Cloning Considered Harmful” Considered Harmful. *Empirical Software Engineering*, 13(6):645–692, 2008.
- C. Kästner. *Virtual Separation of Concerns: Preprocessor 2.0*. PhD thesis, University of Magdeburg, 2010.
- C. Kästner and S. Apel. Type-Checking Software Product Lines - A Formal Approach. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 258–267. IEEE, 2008.
- C. Kästner and S. Apel. Virtual Separation of Concerns - A Second Chance for Preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009.

-
- C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Feature Using AspectJ. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 223–232. IEEE, 2007.
- C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 311–320. ACM, 2008.
- C. Kästner, S. Apel, and M. Kuhlemann. A Model of Refactoring Physically and Virtually Separated Features. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 157–166. ACM, 2009a.
- C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach. In *Proceedings of the International Conference on Objects, Models, Components, Patterns (TOOLS EUROPE)*, pages 174–194. Springer, 2009b.
- C. Kästner, P. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 805–824. ACM, 2011.
- C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-Based Product Lines. *ACM Transactions on Software Engineering and Methodology*, 21(3):1–39, 2012a.
- C. Kästner, K. Ostermann, and S. Erdweg. A Variability-Aware Module System. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 773–792. ACM, 2012b.
- C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward Variability-Aware Testing. In *Proceedings of the International Workshop Feature-Oriented Software Development (FOSD)*, pages 1–8. ACM, 2012c.
- M. Kendall and B. Babington Smith. The Problem of m Rankings. *The Annals of Mathematical Statistics*, 10(3):275–287, 1939.
- A. Kenner, C. Kästner, S. Haase, and T. Leich. TypeChef: Toward Type Checking #ifdef Variability in C. In *Proceedings of the International Workshop Feature-Oriented Software Development (FOSD)*, pages 25–32. ACM, 2010.
- B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, 1988.
- U. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009.

- G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242. Springer, 1997.
- C. Kim, C. Kästner, and D. Batory. On the Modularity of Feature Interactions. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 23–34. ACM, 2008.
- C. Kim, D. Marinov, D. Batory, S. Souto, P. Barros, and M. d’Amorim. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 257–267. ACM, 2013.
- A. Ko and B. Myers. Development and Evaluation of a Model of Programming Errors. In *Proceedings of the Symposium on Human Centric Computing Languages and Environments (HCC)*, pages 7–14. IEEE, 2003.
- S. Kolesnikov, A. von Rhein, C. Hunsen, and S. Apel. A Comparison of Product-based, Feature-based, and Family-based Type Checking. In *Proceedings of the the International Conference on Generative Programming: Concepts & Experiences (GPCE)*, pages 115–124. ACM, 2013.
- M. Kowal, S. Schulze, and I. Schaefer. Towards Efficient SPL Testing by Variant Reduction. In *Proceedings of the International Workshop on Variability & Composition (VariComp)*, pages 1–6. ACM, 2013.
- M. Krone and G. Snelling. On the Inference of Configuration Structures from Source Code. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 49–57. IEEE, 1994.
- D. Kuhn, D. Wallace, and A. Gallo. Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, 2004.
- D. Kuhn, R. Kacker, and Y. Lei. *Introduction to Combinatorial Testing*. CRC Press, 2013.
- W. Landi. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.
- M. Latendresse. Fast Symbolic Evaluation of C/C++ Preprocessing using Conditional Values. In *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR)*, pages 170–179. IEEE, 2003.
- K. Lauenroth, S. Toehning, and K. Pohl. Model Checking of Domain Artifacts in Product Line Engineering. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 269–280. IEEE, 2009.

- B. Leavenworth. Syntax Macros and Extended Translation. *Communications of ACM*, 9(11): 790–793, 1966.
- T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting Static Analysis to Work for Verification: A Case Study. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 26–38. ACM, 2000.
- J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 105–114. ACM, 2010.
- J. Liebig, C. Kästner, and S. Apel. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 191–202. ACM, 2011.
- J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable Analysis of Variable Software. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 81–91. ACM, 2013.
- J. Liebig, A. Janker, F. Garbe, S. Apel, and C. Lengauer. Morpheus: Variability-Aware Refactoring in the Wild. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 380–391. ACM, 2015.
- B. Lientz, B. Swanson, and G. Tompkins. Characteristics of Applications Software Maintenance. *Communications of ACM*, 21(6):466–471, 1978.
- J. Liu, D. Batory, and C. Lengauer. Feature-Oriented Refactoring of Legacy Applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 112–121. ACM, 2006.
- M. Lochau, S. Oster, U. Goltz, and A. Schürr. Model-based Pairwise Testing for Feature Interaction Coverage in Software Product Line Engineering. *Software Quality Journal*, 20(3-4):567–604, 2012.
- D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A Quantitative Analysis of Aspects in the eCos Kernel. In *Proceedings of the EuroSys Conference*, pages 191–204. ACM, 2006.
- D. Lohmann, W. Hofer, W. Schröder-Preikschat, J. Streicher, and O. Spinczyk. CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems. In *Proceedings of the USENIX Technical Conference*, pages 215–228. USENIX Association, 2009.
- R. Madachy. *A Software Project Dynamics Model for Process Cost, Schedule and Risk Assessment*. PhD thesis, University of Southern California, 1994.

- J. Maletic, M. Collard, and H. Kagdi. Leveraging XML Technologies in Developing Program Analysis Tools. In *Proceedings of Workshop Adoption-Centric Software Engineering (ACSE)*, pages 80–85, 2004.
- B. McCloskey and E. Brewer. ASTEC: A New Approach to Refactoring C. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 21–30. ACM, 2005.
- F. Medeiros, M. Ribeiro, and R. Gheyi. Investigating Preprocessor-based Syntax Errors. In *Proceedings of the the International Conference on Generative Programming: Concepts & Experiences (GPCE)*, pages 75–84. ACM, 2013.
- F. Medeiros, M. Ribeiro, R. Gheyi, and B. Fonesca. A Catalogue of Refactorings to Remove Incomplete Annotations. *Journal of Universal Computer Science*, 2014.
- M. Mendonça. *Efficient Reasoning Techniques for Large Scale Feature Models*. PhD thesis, University of Waterloo, 2009.
- M. Mendonça, A. Wąsowski, K. Czarnecki, and D. Cowan. Efficient Compilation Techniques for Large Scale Feature Models. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 13–22. ACM, 2008.
- M. Mendonça, A. Wąsowski, and K. Czarnecki. SAT-based Analysis of Feature Models is Easy. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 231–240. ACM, 2009.
- T. Mens. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.
- T. Mens and T. Tourwé. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.
- J. Midtgaard, C. Brabrand, and A. Wąsowski. Systematic Derivation of Static Analyses for Software Product Lines. In *Proceedings of the International Conference on Modularity*, pages 181–192. ACM, 2014.
- J. Morgenthaler. *Static Analysis for a Software Transformation Tool*. PhD thesis, University of California, 1997.
- G. Murphy, A. Lai, R. Walker, and M. Robillard. Separating Features in Source Code: An Exploratory Study. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 275–284. IEEE, 2001.
- E. Murphy-Hill, C. Parnin, and A. Black. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.

- D. Muthig and T. Patzke. Generic Implementation of Product Line Components. In *Proceedings of the International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World (Net. ObjectDays)*, pages 313–329. Springer, 2002.
- S. Nadi. A Study of Variability Spaces in Open Source Software. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1353–1356. IEEE / ACM, 2013.
- S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining Configuration Constraints: Static Analyses and Empirical Results. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 140–151. ACM, 2014.
- N. Nagappan and T. Ball. Static Analysis Tools as Early Indicators of Pre-release Defect Density. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 580–586. ACM, 2005.
- C. Nie and H. Leung. A Survey of Combinatorial Testing. *ACM Computing Surveys*, 43(2): 1–29, 2011.
- F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
- W. Obdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois, 1992.
- S. Oster, F. Markert, and P. Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 196–210. Springer, 2010.
- Y. Padioleau. Parsing C/C++ Code without Pre-processing. In *Proceedings of the International Conference on Compiler Construction (CC)*, pages 109–125. Springer, 2009.
- Y. Padioleau, J. Lawall, R. Hansen, and G. Muller. Documenting and Automating Collateral Evolutions in Linux Device Drivers. In *Proceedings of the EuroSys Conference*, pages 247–260. ACM, 2008.
- D. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, 2(1):1–9, 1976.
- T. Pearse and P. Oman. Experiences Developing and Maintaining Software in a Multi-Platform Environment. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 270–277. IEEE, 1997.
- D. Pech, J. Knodel, R. Carbon, C. Schitter, and D. Hein. Variability Management in Small Development Organizations: Experiences and Lessons Learned from a Case Study. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 285–294. ACM, 2009.

- G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. le Traon. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST)*, pages 459–468. IEEE, 2010.
- G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Traon. Pairwise Testing for Software Product Lines: Comparison of Two Approaches. *Software Quality Journal*, 20(3-4):605–643, 2012.
- B. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- M. Platoff, M. Wagner, and J. Camaratta. An Integrated Program Representation and Toolkit for the Maintenance of C Programs. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 129–137. IEEE, 1991.
- K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.
- H. Post and C. Sinz. Configuration Lifting: Verification meets Software Configuration. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 347–350. IEEE, 2008.
- C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 419–443. Springer, 1997.
- G. Ramalingam. The Undecidability of Aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, 1994.
- J. Refstrup. Adapting to Change: Architecture, Processes, and Tools: A Closer Look at HP’s Experience in Evolving the Owen Software Product Line. In *Proceedings of the International Software Product Line Conference (SPLC)*, 2009. Keynote presentation.
- A. Reynolds, M. Fiuczynski, and R. Grimm. On the Feasibility of an AOSD Approach to Linux Kernel Extensions. In *Proceedings of Workshop Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, pages 1–7. ACM, 2008.
- D. Ritchie. The Development of the C Language. In *Proceedings of the Conference on History of Programming Languages (HOPL)*, pages 201–208. ACM, 1993.
- M. Rosenmüller, M. Kuhlemann, N. Siegmund, and H. Schirmeier. Avoiding Variability of Method Signatures in Software Product Lines: A Case Study. In *Proceedings of Workshop Aspect-Oriented Product Line Engineering (AOPLE)*, pages 20–25, 2007.
- M. Rosenmüller, S. Apel, T. Leich, and G. Saake. Tailor-Made Data Management for Embedded Systems: A Case Study on Berkeley DB. *Data & Knowledge Engineering (DKE)*, 68(12): 1493–1512, 2009.

- C. Roy and J. Cordy. A Survey on Software Clone Detection Research. Technical Report 2007-541, Queen's University at Kingston, 2007.
- N. Salkind, editor. *Encyclopedia of Measurement and Statistics*. SAGE publications, 2007.
- M. Schäfer. *Specification, Implementation and Verification of Refactorings*. PhD thesis, University of Oxford, 2010.
- M. Schäfer and O. de Moor. Specifying and Implementing Refactorings. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 286–301. ACM, 2010.
- M. Schäfer, M. Verbaere, T. Ekman, and O. de Moor. Stepping Stones over the Refactoring Rubicon. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 369–393. Springer, 2009.
- S. Schulze. *Analysis and Removal of Code Clones in Software Product Lines*. PhD thesis, University of Magdeburg, 2013.
- S. Schulze, E. Jürgens, and J. Feigenspan. Analyzing the Effect of Preprocessor Annotations on Code Clones. In *Proceedings of the Working Conference on Source Code Management and Manipulation (SCAM)*, pages 115–124. IEEE Software, 2011.
- S. Schulze, J. Liebig, J. Siegmund, and S. Apel. Does the Discipline of Preprocessor Annotations Matter? A Controlled Experiment. In *Proceedings of the the International Conference on Generative Programming: Concepts & Experiences (GPCE)*, pages 65–74. ACM, 2013a.
- S. Schulze, M. Lochau, and S. Brunswig. Implementing Refactorings for FOP: Lessons Learned and Challenges Ahead. In *Proceedings of the International Workshop Feature-Oriented Software Development (FOSD)*, pages 33–40. ACM, 2013b.
- S. Schulze, O. Richers, and I. Schaefer. Refactoring Delta-Oriented Software Product Lines. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 73–84. ACM, 2013c.
- S. Schulze, T. Thüm, M. Kuhleemann, and G. Saake. Variant-Preserving Refactoring in Feature-Oriented Software Product Lines. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 73–81. ACM, 2013d.
- S. She, R. Lotufo, T. Berger, A. Wařowski, and K. Czarnecki. The Variability Model of The Linux Kernel. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 45–51. Universität Duisburg-Essen, 2010.
- N. Siegmund, S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting Performance via Automated Feature-Interaction Detection. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 167–177. IEEE, 2012.

- J. Sincero. *Variability Bugs in System Software*. PhD thesis, University of Erlangen-Nuremberg, 2013.
- J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. Is the Linux Kernel a Software Product Line? In *Proceedings of the International Workshop on Opens Source Software and Product Lines (OSSPL)*, 2007.
- J. Sincero, R. Tartler, D. Lohmann, and W. Schröder-Preikschat. Efficient Extraction and Analysis of Preprocessor-based Variability. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 33–42. ACM, 2010.
- Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 550–570. Springer, 1998.
- G. Soares, R. Gheyi, and T. Massoni. Automated Behavioral Testing of Refactoring Engines. *IEEE Transactions on Software Engineering*, 39(2):147–162, 2013.
- H. Spencer and G. Collyer. #ifdef Considered Harmful, or Portability Experience with C News. In *Proceedings of the USENIX Technical Conference*, pages 185–197. USENIX Association, 1992.
- D. Spinellis. Global Analysis and Transformations in Preprocessed Languages. *IEEE Transactions on Software Engineering*, 29(11):1019–1030, 2003.
- D. Spinellis. CScout: A Refactoring Browser for C. *Science of Computer Programming (SCP)*, 75(4):216–231, 2010.
- M. Steffens, S. Oster, M. Lochau, and T. Fogdal. Industrial Evaluation of Pairwise SPL Testing with MoSo-PoLiTe. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 55–62. ACM, 2012.
- F. Steimann and A. Thies. From Public to Privat to Absent: Refactoring Java Programs under Constrained Accessibility. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 419–443. Springer, 2009.
- K. Sullivan, W. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information Hiding Interfaces for Aspect-Oriented Design. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 166–175. ACM, 2005.
- N. Sultana and S. Thompson. Mechanical Verification of Refactorings. In *Proceedings of the International Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 51–60. ACM, 2008.

-
- A. Sutton and J. Maletic. How We Manage Portability and Configuration with the C Preprocessor. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 275–284. IEEE, 2007.
- C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 2002.
- R. Tartler. *Mastering Variability Challenges in Linux and Related Highly-Configurable System Software*. PhD thesis, University of Erlangen-Nuremberg, 2013.
- R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. Feature Consistency in Compile-Time Configurable System Software. In *Proceedings of the EuroSys Conference*, pages 47–60. ACM, 2011.
- R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. Configuration Coverage in the Analysis of Large-scale System Software. *SIGOPS Operating Systems Review*, 45(3):10–14, 2012.
- R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. Static Analysis of Variability in System Software: The 90,000 #ifdefs Issue. In *Proceedings of the USENIX Technical Conference*, pages 421–432. USENIX Association, 2014.
- S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 95–104. ACM, 2007.
- T. Thüm. *Reasoning about Feature Model Edits*. Bachelor’s thesis, University of Magdeburg, 2008.
- T. Thüm, D. Batory, and C. Kästner. Reasoning about Edits to Feature Models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 254–264. IEEE, 2009.
- T. Thüm, I. Schaefer, S. Apel, and M. Hentschel. Family-Based Theorem Proving for Deductive Verification of Software Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 11–20. ACM, 2012.
- T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 47(1):6:1–6:45, 2014.
- F. Tip. Refactoring Using Type Constraints. In *Proceedings of the International Symposium on Static Analysis (SAS)*, pages 1–17. Springer, 2007.
- M. Tomita. LR Parsers for Natural Languages. In *Proceedings of the International Conference on Computational Linguistics (ACL)*, pages 354–357. ACL, 1984.

- M. Vittek. Refactoring Browser with Preprocessor. In *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR)*, pages 101–110. IEEE, 2003.
- D. Waddington and B. Yao. High-Fidelity C/C++ Code Transformation. *Science of Computer Programming (SCP)*, 68(2):64–78, 2007.
- E. Walkingshaw. *The Choice Calculus: A Formal Language of Variation*. PhD thesis, Oregon State University, 2013.
- E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden. Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In *Proceedings of the International Symposium on New Ideas in Programming and Reflections on Software (Onward!)*, pages 213–226. ACM, 2014.
- D. Weise and R. Crew. Programmable Syntax Macros. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 156–165. ACM, 1993.
- N. Wirth. Program Development by Stepwise Refinement. *Communications of ACM*, 14(4): 221–227, 1971.
- C. Zengler and W. Küchlin. Encoding the Linux Kernel Configuration in Propositional Logic. In *Proceedings of Workshop on Configuration*, pages 51–56, 2010. <http://www.hitec-hh.de/confws10/>.
- B. Zhang, M. Becker, T. Patzke, K. Sierszecki, and J. Savolainen. Variability Evolution and Erosion in Industrial Product Lines: A Case Study. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 168–177. ACM, 2013.
- J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk. On the Value of Static Analysis for Fault Detection in Software. *IEEE Transactions on Software Engineering*, 32(4):240–253, 2006.
- H. Zhu, P. Hall, and J. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.