GSDLAB TECHNICAL REPORT

Where do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study

Sarah Nadi, Thorsten Berger, Christian Kästner,
Krzysztof Czarnecki

GSDLAB–TR 2015–01–27 Version 1, January 2015

Generative Software Development Laboratory
University of Waterloo
200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1

WWW page: http://gsd.uwaterloo.ca/

Where do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study

Sarah Nadi, Thorsten Berger, Christian Kästner and Krzysztof Czarnecki

Abstract—Highly configurable systems allow users to tailor software to specific needs. Valid combinations of configuration options are often restricted by intricate constraints. Describing options and constraints in a variability model allows reasoning about the supported configurations. To automate creating and verifying such models, we need to identify the origin of such constraints. We propose a static analysis approach, based on two rules, to extract configuration constraints from code. We apply it on four highly configurable systems to evaluate the accuracy of our approach and to determine which constraints are recoverable from the code. We find that our approach is highly accurate (93 % and 77 % respectively) and that we can recover 28 % of existing constraints. We complement our approach with a qualitative study to identify constraint sources, triangulating results from our automatic extraction, manual inspections, and interviews with 27 developers. We find that, apart from low-level implementation dependencies, configuration constraints enforce correct runtime behavior, improve users' configuration experience, and prevent corner cases. While the majority of constraints is extractable from code, our results indicate that creating a complete model requires further substantial domain knowledge and testing. Our results aim at supporting researchers and practitioners working on variability model engineering, evolution, and verification techniques.

Index Terms—Variability models, reverse-engineering, qualitative studies, static analyses, configuration constraints

1 Introduction

Many software systems need to be customized to specific user needs. Customization is commonly required in embedded systems, for instance, to support a wide range of hardware, to improve performance, or to reduce memory footprints. Consequently, many such systems are designed to be *configurable* by presenting the user with configuration options, or *features*. By selecting a specific set of features, customized variants of the system can be generated. Features can range from options that tweak small functional- and non-functional aspects, to those that enable whole subsystems of the software. Such highly configurable systems range from industrial software product lines to prominent open-source systems software, such as the Linux kernel with currently more than 11,000 features [13], [47], [50].

Configurable systems are usually divided into a *problem space* and a *solution space* [18] as shown in Figure 1. The problem space describes the supported features and their dependencies as constraints, while the solution space is the technical realization of the system and of the functionalities specified by the features (code and build files). Thus, features cross both spaces. They are described in the problem space and mapped to code artifacts in the solution space.

Ideally, configurable systems have a formal, documented *variability model* describing the features and constraints of the problem space. Automated and interactive configurators use such models to support users in navigating a complex configuration space [8], [22], [61], [62]. However, many systems have no documented variability model or rely on informal textual descriptions of constraints (e.g., the FreeBSD kernel [48]). As the number of features and their dependencies increases, configuration becomes more challenging [25], [48], and introducing an explicit variability model is often the way out to conquer complexity and have one central—human- and machine-readable—place for documentation. Manual extraction of constraints and construction of such models for existing systems is a daunting task though, which calls for automation.

Constraints prevent invalid configurations for technical and non-technical reasons. For instance, in an operating system kernel, a technical constraint could prevent having multi-threaded I/O locking without the corresponding threading libraries. Non-technical constraints reflect domain-specific knowledge, such as marketing requirements placed by a project manager or a sales department. For instance, a low-cost model of a mobile phone should not have a high-definition camera.

Despite common use in practice, configuration constraints in variability models are not well understood. Knowing their source, quantity, and quality is important for adopting, evolving, and refactoring highly configurable systems. For instance, understanding sources of constraints provides the basis for their automatic extraction, to support the creation of variability models. It also helps to ensure that no conflicts exist between constraints in the model and in the code. Furthermore,

• *Sarah Nadi is with the College of Computer Science at the Technische Universität Darmstadt, Germany; Thorsten Berger is with the Department of Electrical and Computer Engineering department at the University of Waterloo, Canada; Christian Kästner is with the School of Computer Science School at Carnegie Mellon University, USA; Krzysztof Czarnecki is with the Department of Electrical and Computing Engineering at the University of Waterloo, Canada.*

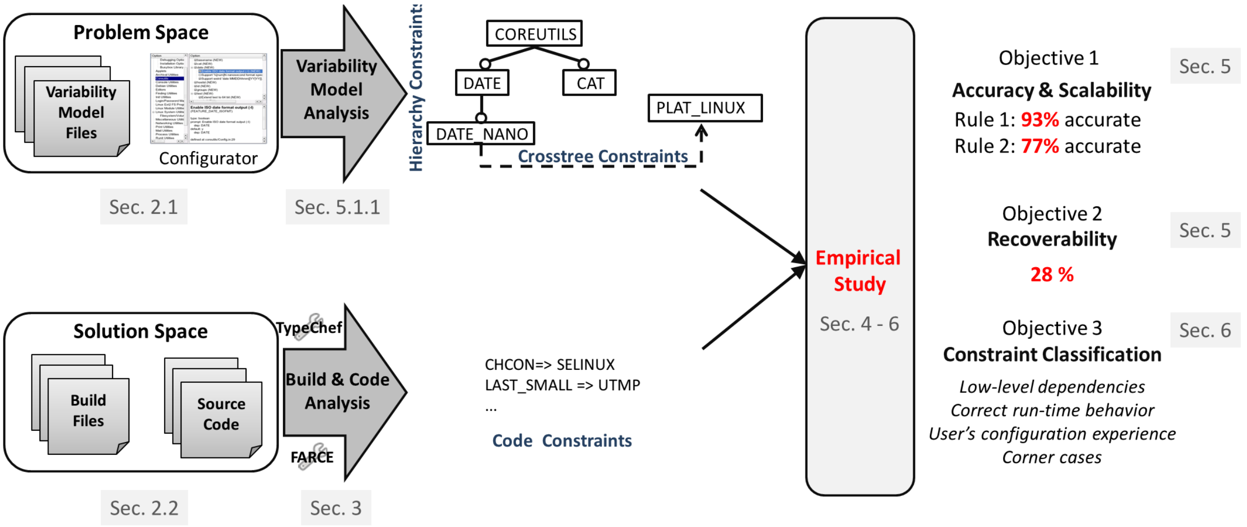


Fig. 1: Overview of our approach and the empirical study

identifying unnecessary constraints in the model can improve software quality and support co-evolution of model and code.

We address this gap by studying configuration constraints in practice. Our goals are (i) to conceive and implement techniques to identify configuration constraints in code, and (ii) to improve the empirical understanding of constraints in variability models. The latter goal strives to determine the reasoning and rationale behind constraints, to assess the limit of our techniques, and to identify complementary sources (additional analyses or expert opinions) of constraints.

We develop *scalable* static analysis techniques to extract configuration constraints from code. We focus on C-based systems with build-time variability using the build system and C preprocessor. Our analysis technique relies on two rules: (1) all valid configurations should build correctly, and (2) they should all yield syntactically different variants. For both rules, we propose novel scalable extraction strategies based on the structural use of #IFDEF directives, and on parser, type, and linker errors. Most importantly, we statically analyze build-time variability effectively, without examining an exponential number of all possible configurations. We empirically study four large open-source systems—uClibc, BusyBox, eCos, and the Linux kernel—with three research objectives: (1) evaluating accuracy and scalability, (2) evaluating recoverability, and (3) classifying constraints.

We show an overview of our proposed approach and the empirical study in Figure 1, leaving details for later. Our results show that our extraction is 93 % and 77 % accurate respectively for the two rules we use, and that it can scale to the size of the Linux kernel, in which we extract over 250,000 unique constraints. We also find that our automated analysis can recover 28 percent of the existing configuration constraints across the four systems.

Our work comprises both an engineering contribution (*extracting constraints from C code*) and an empirical contribution (assessing *accuracy* and *recoverability*, and *classifying existing constraints*). This paper is an extended version of a prior conference paper [35]. Compared to the conference version, we improve our static analysis and, more importantly, we qualitatively study constraints using questionnaires and interviews with 27 developers of the studied subsystems. In summary, we contribute (novel contributions highlighted in bold):

- Adaptations and extensions of existing static analyses to extract configuration constraints from code.
- A novel constraint extraction technique based on feature use and code structure.
- **A combination of the individual analyses to account for interactions among different sources of constraints.**
- An evaluation of our analysis infrastructure with respect to accuracy and recoverability of constraints;
- **A classification of constraint sources based on developer input (interviews and questionnaires), manual analysis, and additional automated analyses.**
- **A discussion of the implications of our empirical results on extraction tools.**

2 Configuration Constraints

Variability support in configurable systems is usually divided into a *problem space* and a *solution space* [18], as shown in Figure 1. This separation allows users to make configuration decisions without knowledge about low-level implementation details. Therefore, both spaces need to be consistent, such that any feature dependencies in the solution space are enforced in the problem space, and no conflicts occur. We are interested in understanding the different types of configuration constraints defined in the problem space, and how many of these are technically reflected in the solution space. This can be done by extracting configuration constraints from both the problem and solution spaces and then comparing and classifying them as shown in Figure 1.

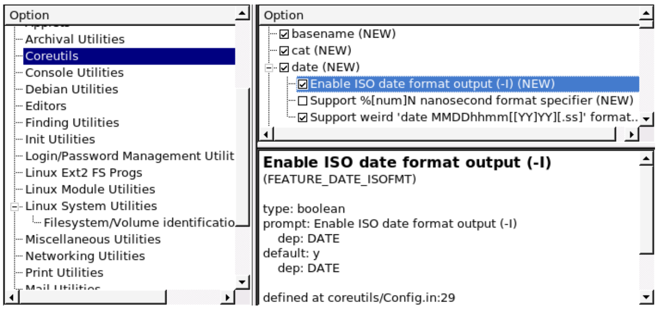
2.1 Problem Space

Features and constraints are described in the problem space, with varying degrees of formality—either informally in plain text, such as in the FreeBSD kernel [48], or using a formal *variability model* expressed in a dedicated language (e.g., Kconfig), as in our subject systems. Given such a model, configurator tools can support users in selecting valid configurations and avoiding invalid ones. Figure 2 shows the configurator of BusyBox, one of our subject systems. The configurator displays features in a hierarchy, which can then be selected by users, while enforcing configuration constraints, such as propagating choices or graying out features that would lead to invalid configurations. Constraints reside in the *feature hierarchy* (a child implies its parent) and in additional rules of *cross-tree constraints* [12]. Specifically, the feature hierarchy is one of the major benefits of a variability model [48], as it helps users to configure a system and developers to organize features.

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

We illustrate these kinds of constraints with examples from two of our subject systems. In the Linux kernel, a *technical constraint* which is reflected in the code is that "multi-threaded I/O locking" depends on "threading support" due to low-level code dependencies. A *technical constraint* which cannot be detected from the code is that "64GB memory support" excludes "386" and "486" CPUs, which stems from the domain knowledge that these processors cannot handle more than 4GB of physical memory. In BusyBox (see Figure 2), a *technical constraint* is that "Enable ISO date format" requires "date", since the code of the former feature could not be compiled without the latter. A *non-technical*, configurator-related, constraint is that feature "date" itself appears under the menu feature "Coreutils" in the configurator hierarchy. Such groupings are used to allow users (and developers) to find features faster.

There has been much research to extract constraints from existing variability models within the problem space [10], [46], [54]. Such extractors can interpret the semantics of different variability modeling languages to extract both hierarchy and cross-tree constraints, as shown in Figure 1.



☑ basename (NEW)
☑ cat (NEW)
☑ date (NEW)
☑ Enable ISO date format output (-I) (NEW)
☐ Support %[num]N nanosecond format specifier (NEW)
☑ Support weird 'date MMDDhhmm[[YY]YY][.ss]' format..

Enable ISO date format output (-I)
(FEATURE_DATE_ISOFMT)

```
type: boolean
prompt: Enable ISO date format output (-I)
   dep: DATE
default: y
   dep: DATE

defined at coreutils/Config.in:29
```

Fig. 2: Configurator of the BusyBox system

2.2 Solution Space

The solution space consists of build and code files. Our focus is on C-based systems that realize configurability with their build system and the C preprocessor. The build system decides the source files and the preprocessor the code fragments to be compiled. The latter is realized using conditional-compilation preprocessor directives such as #IFDEFs.

To compare constraints in the variability model to those in the code, we must find ways to extract global configuration constraints from the code (as opposed to localized code block constraints [54]). We assume that there is a solution-space (code-level) constraint if any configuration violating this constraint is ill-defined by some rule. There may be several sources of constraints that fit such a description. However, in this work, we identify two tractable sources of constraints: (i) those resulting from build-time errors and (ii) those resulting from the effect of features in build files and in the structure of the code (e.g., #IFDEF usage). We now explain the two rules and their justification.

2.2.1 Build-Time Errors

Every valid configuration needs to build correctly. In C projects, various types of errors can occur during the build: preprocessor errors, parsing errors, type errors, and linker errors. Our goal is to detect configuration constraints that prevent such build errors. We derive configuration constraints from the following rule:

> ***Rule 1.*** *Every valid configuration of the system must not contain build-time errors, such that it can be successfully preprocessed, parsed, type checked, and linked.*

A naive, but not scalable, approach to extract these constraints would be to build and analyze every single configuration in isolation. If every configuration with feature X compiles except when feature Y is selected, we could infer a constraint $X \rightarrow \neg Y$. For instance, in Listing 1a, the code will not compile in some configurations, due to a type error in Line 6: The function `foo()` is called under condition X, while it is only defined under condition $\neg Y$; thus, the constraint $X \rightarrow \neg Y$ must always hold. The problem space needs to enforce this constraint to prevent invalid configurations that break the compilation. However, already in a medium-sized system such as

```
#ifndef Y
void foo() { ... }
#endif

#ifdef X
void bar() { foo(); }
#endif
```

(a) type error

```
#if defined(Z)&&defined(X)
...
#ifdef W
...
#endif
...
#endif
```

(b) feature effect

Listing 1: Examples of constraint sources

BusyBox with 881 Boolean features, this results in more than 2^{881} configurations to analyze, which is more than the number of atoms in the universe. We show how this can be avoided in Section 3.

2.2.2 Feature Effect

Ideally, variability models should also prevent meaningless configurations, such as redundant feature selections that do not change the solution space. That is, if a feature A is selected in a configuration, then we expect that A adds, changes, or removes some behavior (that was not previously present). If a feature has no *effect* unless other features are selected (or deselected), a configurator may hide or disable it, simplifying the configuration process for users.

Determining if two variants of a program are equivalent is difficult (even undecidable). We approximate this by comparing whether the programs differ in their source code at all. If two different configurations yield the same code, this suggests some *anomaly* (as opposed to errors described in Section 2.2.1) in the model.

We extract constraints that prevent such anomalies. We use the following rule as a simplified, conservative approximation of our second source of constraints:

> ***Rule 2.*** *Every valid configuration of the system should yield a lexically different program.*

The use of features within the build system and the preprocessor directives for conditional compilation provides information about the context under which selecting a feature makes a difference in the final product. In the code fragment in Listing 1b, selecting W without selecting Z and X will not break the system. However, only selecting W will not affect the compiled code, since the surrounding block will not be compiled without Z and X also being selected. Thus, W → Z∧X is a feature-effect constraint that should likely be in the model, even though violating it will not break the compilation.

2.3 Problem Statement

We can summarize that variability-model constraints arise from different sources. We discussed two such sources above where the constraints exist for technical reasons discoverable from the code. Our work strives to automatically extract such constraints. However, it is not clear if other sources of constraints exist beyond implementation artifacts and how prevalent they are. We, therefore, also aim to identify any additional sources of configuration constraints and analyses used to extract them.

Improving empirical understanding of constraints in real systems is crucial, especially since several studies emphasize configuration and implementation challenges for developers and users due to complex constraints [9], [13], [25], [34]. Such knowledge not only allows us to understand which parts of a variability model can be reverse engineered and checked for consistency from code, and to what extent; but also how much manual effort, such as interviewing developers or domain experts, would be necessary to achieve a full model. For example, a main challenge when reverse-engineering a variability model from constraints is to disambiguate the hierarchy [48]. Thus, this process could be supplemented by knowing which sources of constraints relate to hierarchy information in the model.

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

3 Automatic Extraction Methodology

We used the following methodology to extract configuration constraints from code, as illustrated in Figure 3.

3.1 Extracting File Presence Conditions

To accurately analyze files and to derive constraints, we first need to know under which condition the build system includes each file. We use the term *presence condition (PC)* to refer to a propositional expression over features that determines when a certain code artifact is compiled. For example, a file with presence condition HUSH ∨ ASH is compiled and linked if and only if the features HUSH or ASH are selected.

Such file presence conditions are encoded in the build system, which typically consists of several imperative scripts, descriptions, or Makefiles. We need to manually or automatically extract a presence condition for each file. These file presence conditions allow us to derive global constraints from the low-level sources within each file. For example, if a type error occurs under condition X in a file guarded by a presence condition Y, then the error actually occurs under X ∧ Y. In other words, local presence conditions induced by conditional compilation directives are conjoined with the file presence condition before deriving (global) constraints from our low-level sources.

In Section 5.1, we mention the tools we use to mechanically extract file presence conditions. In the remainder of this section, without restriction of generality, we assume

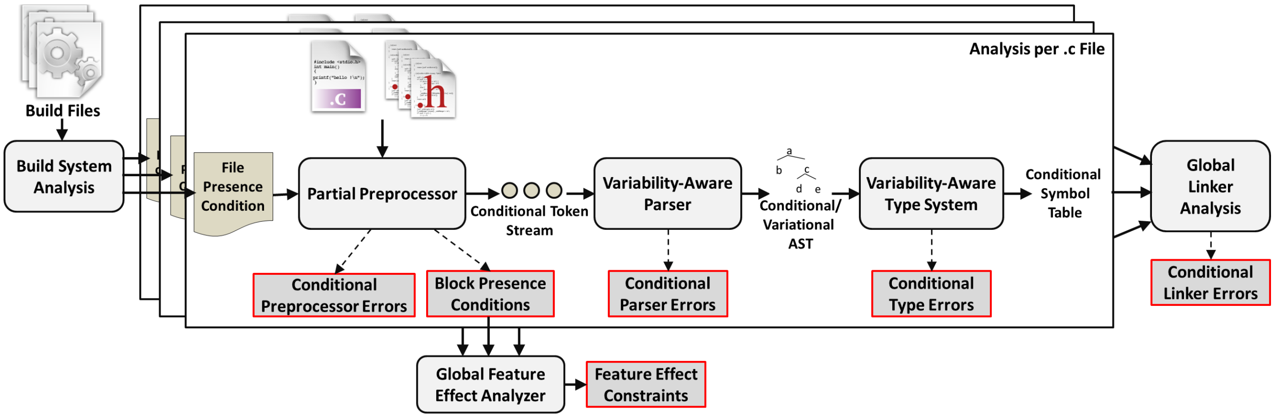


Fig. 3: Variability-aware approach to extract configuration constraints from code

that we have already identified a presence condition ϕ for each file and encoded it inside the file as an #IFDEF ϕ condition that wraps the whole file (e.g., Line 0 in Listing 2, effectively pushing the file presence condition into each file.

3.2 Extracting Code Constraints

We use the two rules described in Section 2 to extract code constraints from *preprocessor errors*, *parser errors*, *type errors*, *linker errors*, and *feature effect*. A simple approach to do this is to analyze every single possible configuration to find which ones contain errors. To avoid such an intractable brute-force approach and to avoid incompleteness from sampling strategies, we build on our recent research infrastructure, *TypeChef*, to analyze the entire configuration space of C code with build-time variability at once [27]–[29].

Our overall strategy for extracting code constraints is based on parsing C code without evaluating conditional compilation directives. We extend and instrument TypeChef to accomplish this. TypeChef only partially preprocesses a source file—it resolves all #INCLUDEs and expands all macros, but preserves conditional compilation directives. On alternative macro definitions or #INCLUDEs, it explores all possibilities, similar to symbolic execution. Partial preprocessing produces a token stream in which each token is guarded by a corresponding accurate presence condition (including the file presence condition, see Section 3.1), which is subsequently parsed into a conditional abstract syntax tree, which again can be subsequently type checked. This variability-aware analysis is conceptually sound and complete with regard to a brute-force approach of preprocessing, parsing, and type checking all configurations separately. However, it is much faster since it does the analysis in a single step and exploits similarities among the implementations of different configurations; see [27]–[29] for further details.

Typically, TypeChef is called with a given variability model and it only emits error messages for preprocessor, parser, or type problems that can occur in valid configurations—discarding all implementation problems that are already excluded by the variability model. This is the classic approach to find implementation errors, which a user can subsequently fix in the implementation or in the variability model [19], [55], [56]. Since we need to extract all constraints without knowledge of valid configurations, we run TypeChef *without* a variability model to process all reported problems in all configurations.

We extend and instrument TypeChef, and implement a new framework FARCE (FeAtuRe Constraint Extractor)[1], which analyzes the output of TypeChef and the structure of the codebase with respect to preprocessor directive nesting, derives constraints according to our two rules described in Section 2.2, and provides an infrastructure to compare extracted constraints between a variability model and code.

We now explain how we extract code constraints using our two rules in detail. We use the C code in Listing 2 as a running example to illustrate the various constraints we can extract.

3.2.1 *Preprocessor, Parser, and Type Constraints*

Preprocessor errors, parser errors, and type errors are detected at different stages of analyzing a translation unit. However, the post-processing used to extract constraints from them is similar; thus, we discuss them together. In contrast, linker errors require a global analysis over multiple translation units, which we discuss separately.

Preprocessor Errors: A normal C preprocessor stops on #ERROR directives, which are usually intentionally introduced by developers to avoid invalid feature combinations. We extend our partial preprocessor to log #ERROR directives with their corresponding condition, and to continue with the rest of the translation unit instead of stopping on the #ERROR message. In our example (Listing 2), Line 3 shows a #ERROR directive that occurs under the condition ASH ∧ NOMMU.

Parser Errors: Similarly, a normal C parser stops on syntax errors, such as unclosed parentheses. Our

[1] https://bitbucket.org/tberger/farce

```
 0 #ifdef ASH //represents the file presence condition
 1
 2 #ifdef NOMMU
 3 #error "... ash will not run on NOMMU machine"
 4 #endif
 5
 6 #ifdef EDITING
 7 static line_input_t *line_input_state;
 8
 9 void init() {
10   initEditing()
11
12   int maxlength = 1 *
13
14   #ifdef MAX_LEN
15     100;
16   #endif
17 }
18 #endif //EDITING
19
20 int main() {
21   #ifdef EDITING_VI
22     #ifdef MAX_LEN
23       line_input_state−>flags |= 100
24     #endif
25   #endif
26 }
27 #endif //ASH
```

Listing 2: Running example of C code with compile-time errors (adapted from `ash.c` in Busybox)

TypeChef parser reports an error message together with a corresponding condition, but continues parsing for other configurations. In Listing 2, a parser error occurs on Line 12 because of a missing semicolon if $\textsc{max\_len}$ is not selected. In this case, our analysis reports a parser error under condition $\textsc{ash} \land \textsc{editing} \land \neg\textsc{max\_len}$.

Type Errors: Where a normal type checker reports type errors in a single configuration, TypeChef's variability-aware type checker [27], [29] reports each type error together with a corresponding condition. In Listing 2, we detect a type error in Line 23 if $\textsc{editing}$ is not selected since `line_input_state` is only defined under condition $\textsc{ash} \land \textsc{editing}$ on Line 7. TypeChef would, thus, report a type error (undefined symbol) under condition $\textsc{ash} \land \textsc{editing\_vi} \land \textsc{max\_len} \land \neg\textsc{editing}$.

Constraints: Following *Rule 1*, we expect that each file should compile without errors. Every error message with a corresponding condition indicates a part of the configuration space that does not compile and should hence be excluded in the variability model. For each condition e of an error, we add a constraint $\neg e$ to the set of automatically extracted configuration constraints.

In our running example, we extract the following constraints (rewritten to equivalent implications): $\textsc{ash} \rightarrow \neg\textsc{nommu}$ from the preprocessor, $\textsc{ash} \rightarrow (\textsc{editing} \rightarrow \textsc{max\_len})$ from the parser, and $\textsc{ash} \rightarrow ((\textsc{editing\_vi} \land \textsc{max\_len}) \rightarrow \textsc{editing})$ from the type system.

More formally, we create a single formula to represent each of these categories of error constraints as follows:

$$\phi_{\text{parser/preprocessor/type}} = \bigwedge_i (\neg e_i) \tag{1}$$

where e_i is the presence condition of a preprocessor/parser/type error.

TABLE 1: Example of two conditional symbol tables

translation unit	symbol	kind	presence condition

Listing 2	`init`
	`main`
	`initEditing`
		export
		export
		import
			ASH ∧ EDITING
			ASH
			ASH ∧ EDITING

other file	`initEditing`	export	INIT

3.2.2 Linker Constraints

To detect linker errors in configurable systems, we build a conditional symbol table for each translation unit during type checking. The symbol table describes all non-static symbols as exported symbols and all called but not defined symbols as imports. All imports and exports are again guarded by corresponding presence conditions. We show the conditional symbol table (without type information) of our running example in Table 1, assuming that symbol `initEditing` is defined under presence condition $\textsc{init}$ in some other translation unit (not shown). More details on conditional symbol tables can be found in related publications on variability-aware module systems [5], [29].

In contrast to the file-local preprocessor, parser, and type constraint analyses, linker analysis is global across all translation units. From all conditional symbol tables, we now detect linker errors and derive corresponding constraints. Again, we follow Rule 1: a linker error arises when a module imports a symbol which is not exported (*def/use*) or when two modules export the same symbol (*conflict*). We derive constraints for each symbol s as follows:

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

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

where $\mathit{imp}(s)$ and $\mathit{exp}(s)$ look up all imports and exports of symbol s in all conditional symbol tables and return a set of tuples (f, ψ), each determining the translation unit f in which s is imported/exported and the presence condition ψ under which this happens. The *def/use* constraints ensure that the presence condition of an import implies at least one presence condition of a corresponding export, while the *conflict* constraints ensure mutual exclusion of the presence conditions of exports with the same symbol name.

An overall linker formula can be derived by conjoining all *def/use* and *conflict* constraints for each symbol in the set of all symbols S:

$$\phi_{linker} = \bigwedge_{s\in S} \mathit{def/use}(s) \land \mathit{conflict}(s) \tag{2}$$

If the two files shown in Table 1 were the only files of our running example, we would extract constraint $\textsc{ash} \land \textsc{editing} \rightarrow \textsc{init}$ for symbol `initEditing`.

3.2.3 Feature Effect

To ensure Rule 2 of lexically different programs in all valid configurations, we detect the configurations under which a feature has no effect on the compiled code and create a constraint to disable the feature in those configurations. The general idea is to detect nesting among #IFDEFs: When a feature only occurs nested inside an #IFDEF of another feature, such as EDITING that occurs only nested inside '#IFDEF ASH' in our running example, the nested feature does not have any effect when the outer feature is not selected. Hence, we would create a constraint that the nested feature should not be selected without the outer feature, because it would not have any effect: EDITING → ASH in our example.

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

First, we collect all unique presence conditions of all code fragments occurring in the entire system (in all translation units, including the corresponding file presence condition as usual). Technically, we inspect the conditional token stream produced by TypeChef's partial preprocessor and collect all *unique* token presence conditions (note that this covers all conditional compilation directives, #IF, #IFDEF, #ELSE, #ELIF, etc. including dynamic reconfigurations with #DEFINE and #UNDEF).

To compute a feature's effect, we use the following insights: given a set of presence conditions P found for code blocks anywhere in the project and the set of features of interest F, then we say a feature $f \in F$ has no effect in a presence condition $\rho \in P$ if $\rho[f \leftarrow True]$ is equivalent to $\rho[f \leftarrow False]$, where $X[f \leftarrow y]$ means substituting every occurrence of f in X by y. In other words, if enabling or disabling a feature does not affect the value of the presence condition, then the feature does not have an effect on selecting the corresponding code fragments.

Furthermore, we can identify the exact condition when a feature f has an effect on a presence condition ρ by finding all configurations in which the result of substituting f is different (using xor: $\rho[f \leftarrow True] \oplus \rho[f \leftarrow False]$). This method is also known as *unique existential quantification* [24].

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

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

We then create a conjunction of all such nesting constraints, and call the final result ϕ_{feffect}. More formally, ϕ_{feffect} would be calculated as follows, where F is the set of all features:

$$\phi_{\text{feffect}} = \bigwedge_{f \in F} (f \rightarrow \bigvee_{\rho \in P} \rho[f \leftarrow True] \oplus \rho[f \leftarrow False]) \quad (3)$$

We could also enable a feature by default and forbid disabling it when disabling has no effect (or use some different default per feature): we just need to negate f on the right-hand side of the above formula. However, we assume the more natural setting where most features are disabled by default, and so we look for the effect of *enabling* a feature.

In our running example in Listing 2, we can identify five unique presence conditions (excluding tokens for spaces and line breaks): ASH, ASH ∧ NOMMU, ASH ∧ EDITING, ASH∧EDITING∧MAX_LEN, and ASH∧EDITING_VI∧MAX_LEN. To determine the effect of MAX_LEN, we would substitute it with `True` and `False` in each of these conditions, and create the the following constraint (assuming that MAX_LEN does not occur anywhere else in the code):

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

This confirms that MAX_LEN only has an effect if ASH and either EDITING or EDITING_VI are selected. In all other cases, the constraint enforces that MAX_LEN remains deselected.

Additionally, to determine how many configuration constraints the build system alone provides, we do the same analysis for file presence conditions only instead of presence conditions of code blocks (which include both file and local presence conditions). Note that this analysis is incomplete and provides only a rough approximation of configuration constraints. On the other hand, it provides insight into the role of the build system in enforcing configuration constraints.

3.2.4 Full Code Formula

Besides having individual formulas that represent each constraint source, we also conjoin them into a single formula representing all code constraints. This ensures that any interaction among the individual constraints is accounted for in an overall, global formula. However, recall that the reasoning behind Rule 1 and Rule 2 is different; the former represents errors, whereas the latter is a heuristic whose violation does not break the system. Thus, we still distinguish between both and yield the following three global formulas:

$$\phi_{\text{rule1}} = \phi_{\text{preprocessor}} \wedge \phi_{\text{parser}} \wedge \phi_{\text{type}} \wedge \phi_{\text{linker}} \quad (4)$$

$$\phi_{\text{rule2}} = \phi_{\text{feffect}} \wedge \phi_{\text{feffect\_build}} \quad (5)$$

$$\phi_{\text{code}} = \phi_{\text{rule1}} \wedge \phi_{\text{rule2}} \quad (6)$$

In addition to the individual formulas 1–3, we will also use these global formulas 4–6 when assessing recoverability of existing variability-model constraints as will be shown in Section 5.

4 Empirical Study Overview

To understand what configuration constraints are enforced in practice and to what extent they can be extracted, we study four real-world systems with existing variability models. Existing models are required to have a basis for comparison. In this section, we describe the four subject systems as well as our three objectives.

4.1 Subject Systems

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

All subjects have a variability model, which we use in the comparison. The first three use the Kconfig language [63], and the last one uses the CDL language [60], each with the respective configurator infrastructure in the problem space.

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

In all systems, the variability models have been created, maintained, and evolved by the original developers of the systems over periods of up to 13 years. Using them reduces experimenter bias in our study. Prior studies of the Linux kernel and BusyBox have also shown that their variability models, while not perfect, are reasonably well maintained [12], [28], [29], [34], [41], [54]. In particular, eCos and Linux have two of the largest publicly available variability models today.

4.2 Objectives

Our empirical study aims at three objectives:

Objective 1 to evaluate *accuracy* and *scalability* of our extraction approach. This is done by checking if the configuration constraints that we extract from implementation are enforced in existing variability models.

Objective 2 to study the *recoverability* of variability-model constraints using our approach. Specifically, we are interested in how many of the existing model constraints reflect implementation specifics that can be automatically extracted from the solution space.

Objective 3 to *classify* variability-model constraints. We want to understand which constraints are technically enforced and which constraints go beyond the code artifacts. This allows us to understand what reverse-engineering approaches to choose in practice.

Since Objectives 1 and 2 primarily evaluate our infrastructure, we discuss them together in Section 5. Objective 3 is more exploratory, aiming at understanding the types of constraints enforced by developers and requires a different research method. Thus, we discuss it separately in Section 6. In both sections, we explain the experiment setup and present the respective results.

5 Accuracy, Scalability, and Recoverability

In this section, we report on the accuracy and scalability of our infrastructure (**Objective 1**), and identify the recoverability of existing variability-model constraints (**Objective 2**) from code.

We first describe the study setup in Section 5.1 and then present the results of objectives 1 and 2 in Sections 5.2 and 5.3, respectively.

5.1 Study Setup

For the setup, we first describe how we extract constraints from the variability model and then how we evaluate our code analysis against these model constraints.

5.1.1 Methodology and Tool Infrastructure

We follow the methodology shown in Figure 1. We first extract hierarchy and cross-tree constraints from the variability models (problem space) of our subject systems. We rely on our previous analysis infrastructures LVAT [3] and CDLTools [1], which can interpret the semantics of Kconfig and CDL respectively to extract such constraints and additionally produce a single propositional formula representing all enforced constraints (see the work on analyzing Kconfig and CDL [10], [46] for details).

We then run TypeChef on each system and use our developed infrastructure FARCE to derive solution-space constraints from its error output (*Rule 1*, cf., Section 2.2.1) and the conditional token stream (*Rule 2*, cf., Section 2.2.2). As a prerequisite, we extract file presence conditions from build systems by using the build-system analysis tool *KBuildMiner* [2] for systems using KBUILD (BusyBox and Linux), and a semi-manual approach for the others.

5.1.2 Evaluation Technique and Measurement Strategy

After problem and solution-space constraints are extracted, we compare them according to the first two objectives.

To address **Objective 1** (evaluate accuracy and scalability), we verify whether extracted solution-space constraints hold in the propositional formula representing the variability model (problem space formula) of each system. We also measure the execution time of the involved analysis steps. For this objective, we assume the existing variability model as the ground truth, since it reflects the system's configuration knowledge specified by developers, and measure accuracy as follows. We keep constraints extracted in the individual steps of our analysis separate. That is, for each build error (*Rule 1*) and each feature effect (*Rule 2*), we create a separate constraint α_i. For each extracted constraint α_i, we check whether it holds in the problem space formula ν (representing variability model constraints) with a SAT solver, by determining whether $\nu \Rightarrow \alpha_i$ is a tautology (i.e., whether its negation is not satisfiable).

For scalability, we record execution time of each analysis step separately to measure the scalability of our approach. All our experiments are executed on a server with two AMD Opteron processors (16 cores each) and 128GB RAM. For all analysis steps performed by TypeChef and KBuildMiner, which can be parallelized, we report the average and the standard deviation of processing each file. In addition, we provide the total processing time for the whole systems, assuming sequential execution of file analyses. For the derivation of constraints, which cannot be easily parallelized, we report the total computation time per system.

To address **Objective 2** (recoverability of model constraints), we determine whether each existing variability model constraint holds in the solution-space constraint formulas we extract. We use the term *recoverability* instead of *recall*, because we do not have a ground truth in terms of which constraints can be extracted from the code. Since no previous study has classified the kinds of constraints in variability models, we cannot expect that 100% of them represent low-level code dependencies which can be statically extracted.

Measuring recoverability is a bit more challenging than measuring accuracy since for the latter, we have the individual, extracted constraints to compare against. However, variability models in practice are described in different modeling languages. Semantics of a variability model are typically expressed uniformly as a single large Boolean function expressed as a propositional formula describing the valid configurations. After experimenting with several slicing techniques for comparing these propositional formulas, we decided to exploit structural characteristics that are commonly found in variability models. In all analyzed models, we can identify child-parent relationships (*hierarchy constraints*) as well as inter-feature constraints (*cross-tree constraints*). This way, we count individual constraints as the developer modeled them, which is intuitive to interpret and allows us to investigate the different types of model constraints. We only account for binary constraints as they are most frequent, whereas accounting for n-ary constraints is an inherently hard combinatorial problem. Technically, we perform the inverse comparison to that described above for accuracy: we compare whether each individual problem-space constraint ν_j holds in the conjunction of all extracted solution-space constraints $\phi_{analysis}$ in each code analysis category (Formulas 1–3) as well as the overall code formulas (Formulas 4–6), i.e., whether $\phi_{analysis} \Rightarrow \nu_i$ is a tautology.

Note that using propositional logic for comparison comes with its own set of problems. For example, comparing two constraints or formulas which have a different set of features or comparing disjunctions may lead to misleading results. We provide a detailed discussion of these cases in Appendix B. Additionally, checking if a constraint holds in three single formulas separately may provide a different result than checking if the constraint holds in the conjunction of the three formulas (as will be seen in the Linux kernel recoverability results). While finding the proper comparison mechanism is an open problem, our comparison technique allows us to understand configuration constraints better, despite its drawbacks. A key factor in selecting this comparison technique is that we can currently manually verify, track, and understand the logic behind the variability-model constraints, which also allows us to ask developers about these constraints.

5.2 Objective 1: Accuracy and Scalability

We expect that all constraints extracted according to Rule 1 hold in the problem-space (variability model) formula, as these prevent any failure in building a system. Constraints that do not hold either indicate a false positive due to an inaccuracy of our implementation or an error in the variability model or implementation; we investigate these cases separately. Such constraint checks have been the standard approach in previous work on finding bugs in configurable systems [19], [28], [57], where inconsistencies between the model and implementation are identified as errors. In contrast, Rule 2 prevents meaningless configurations that lead to duplicate systems. Thus, we expect a large number of corresponding constraints, but not all, to occur in the variability model.

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

Both results show that we achieve a very high accuracy across all four systems. *Rule 1* is a reliable source of constraints where our tooling produces only few

TABLE 2: Constraints extracted with each rule per system, and percentage holding in the variability model (VM)[1]

Code Analysis	uClibc		BusyBox		eCos		Linux	
	# extracted	% found in VM	# extracted	% found in VM	# extracted	% found in VM	# extracted	% found in VM
Rule 1								
Preprocessor Constr.	158	100 %	3	100 %	162	81 %	12,780	81 %
Parser Constr.	59	100 %	23	100 %	133	91 %	8,443	100 %
Type Checking Constr.	947	97 %	54	100 %	139	82 %	256,510	97 %
Linker Constr.	312	63 %	38	100 %	7	100 %	19,654	90 %
Total	1,330	90 %	118	100 %	441	85 %	284,914	96 %
Rule 2								
Feature effect Constr.	57	74 %	359	93 %	263	62 %	2,961	95 %
Feature effect - Build Constr.	26	81 %	62	0 %	n/a	n/a	2,552	97 %
Total	83	76 %	421	79 %	263	62 %	5,513	96 %

[1] Geometric mean of highlighted percentages is used to compute overall accuracy of Rules 1 and 2 (93 % and 77 % respectively).

false positives (extracted constraints that do not hold in the model). Interestingly, a 77 % accuracy rate for Rule 2 suggests that variability models in fact prevent meaningless configurations to a high degree.

Table 3 shows execution times of our tools. Significant time is taken to parse files, which often explode after expanding all macros and #INCLUDE preprocessor directives. Our results show that our analysis scales reasonably where a system as large as Linux can be analyzed in parallel within twelve hours on our hardware.

TABLE 3: Duration, in seconds unless otherwise noted, of each analysis step. Average time per file and standard deviation shown for analysis using TypeChef. Global analysis time shown for post-processing using FARCE

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

5.2.1 Accuracy Discussion

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

Regarding false positives, we identify the following reasons. First, the variability model and the implementation have bugs. In fact, we earlier found several errors in BusyBox and reported them to the developers [29]. We also found one and reported it in uClibc. Second, all steps involved in our analysis are nontrivial. For example, we reimplemented large parts of a type system for GNU C and reverse-engineered details of the Kconfig and CDL languages, as well as the KBUILD build system. Little inaccuracies or incorrect abstractions are possible. After investigating false positives in uClibc linker constraints, we found that many of these occur due to incorrectly (manually) extracted file presence conditions. In general, intricate details in Makefiles, such as shell calls [11], complicate their analysis [53]. Third, our subjects implement their own mechanisms for providing and generating header files at build-time, according to the configuration. We implemented emulations of these project-specific mechanisms to statically mimic their behavior, but such emulations are likely incomplete. We plan to investigate using symbolic execution of build systems [53] in order to accurately identify which header files need to be included under different configurations.

5.2.2 Scalability Discussion

Our evaluation shows that our approach scales, in particular to systems sharing the size and complexity of the Linux kernel. However, we face many scalability issues when combining complex constraint expressions into one formula, mainly in Linux and eCos. Feature-effect constraints are particularly problematic due to the unique existential quantification (see Section 3.2.3), which causes an explosion in the number of disjunctions in many expressions, thus adding complexity to the SAT solver. To overcome this, we omit expressions including more than ten features when aggregating the feature effect formula. This resulted in using only 17 % and 51 % of the feature-effect constraints in Linux and eCos, respectively. The threshold was chosen due to the intuition that larger constraints are too complex and likely not modeled by developers.

We faced similar problems in deriving other formulas, such as the type formula in Linux, but mainly due to the huge number of constraints and not their individual complexity. This required several workarounds and led to high memory consumption in the conversion of the formula into conjunctive normal form, as required by the SAT solver. Thus, we conclude that extracting constraints according to our rules scales, but can require workarounds or filtering expressions to deal with the explosion of constraint formulas. We refer to our online appendix [4] for more details.

TABLE 4: Number (and percentage) of variability-model hierarchy constraints recovered from each code analysis[1]

	uClibc	BusyBox	eCos	Linux
# of VM Hierarchy Constraints	54	366	588	4,999
	Count (%) Recovered from code			
Rule 1				
Preprocessor Constr. ($\phi_{\text{preprocessor}}$)	0 (0 %)	0 (0 %)	0 (0 %)	1 (0 %)
Parser Constr. (ϕ_{parser})	0 (0 %)	0 (0 %)	3 (1 %)	1 (0 %)
Type Checking Constr. (ϕ_{type})	0 (0 %)	1 (0 %)	0 (0 %)	0 (0 %)
Linker Constr. (ϕ_{linker})	0 (0 %)	1 (0 %)	0 (0 %)	1 (0 %)
Rule 1 (ϕ_{rule1})	1 (2 %)	2 (1 %)	4 (1 %)	306 (6 %)
Rule 2				
Feature effect Constr.	8 (15 %)	251 (69 %)	48 (8 %)	325 (6 %)
Feature effect - Build Constr.	4 (7 %)	0 (0 %)	-	1,337 (27 %)
Rule 2 (ϕ_{rule2})	9 (17 %)	251 (69 %)	48 (8 %)	1,663 (33 %)
Full Code Constraints (ϕ_{code})	14 (26 %)	265 (72 %)	53 (9 %)	2,569 (51 %)

[1] Highlighted numbers are those used in the text for easier referencing.

5.3 O2: Recoverability

We now investigate how many variability-model constraints can be automatically extracted from the code. In Tables 4 and 5, we show how many of the variability models' hierarchy and cross-tree constraints, respectively, can be recovered automatically from code. We show the number (and percentage) of constraints recovered by each source (i.e., parsing errors, type errors, feature effect, etc.) as well as the number recovered overall by each rule. Recall that the formula of each rule is the conjunction of the individual formulas related to that rule (see Section 3.2.4). We also show the number and percentage recovered by the overall code formula (ϕ_{code}). Since Tables 4 and 5 split the hierarchy and cross-tree constraints, we provide a summary of the overall aggregated recoverability results in Table 6. In all three tables, we highlight the values mentioned in the text for easier referencing.

As shown in Tables 4 and 5, combining the extracted constraints from all sources used leads to recovering more variability-model constraints. This suggests that the constraints enforced in the variability model are global in the sense that they stem from an interaction among different parts of the system: for example, a combination of preventing a type error and preventing meaningless selections. Note that the same constraint may be recovered via multiple sources. Therefore, the overall code formulas in each table show the total number (and percentage) of *unique* variability-model constraints recovered from analyzing the code. Overall, across the four systems, we recover 31 % of hierarchy constraints, and 23 % of cross-tree constraints. Our overall recoverability across the four systems for all types of constraints using both rules (as shown in Table 6) is 28 %.

To compare the two rules we use to extract solution-space constraints, we show the overlap between the total number of recovered variability-model constraints (both hierarchy and cross-tree) aggregated across both rules in the Venn diagrams in Figure 4. These illustrate that in all systems, a higher percentage of the variability-model constraints reflects feature-effect constraints in the code (Rule 1) and that minimum overlap occurs between constraints recovered by both rules.

TABLE 5: Number (and percentage) of variability-model cross-tree constraints recovered from each code analysis

	uClibc	BusyBox	eCos	Linux
# of VM Cross-tree Constraints	118	265	315	7,759
	Count (%) Recovered from code			
Rule 1				
Preprocessor Constr. ($\phi_{\text{preprocessor}}$)	2 (2 %)	1 (0 %)	5 (2 %)	6 (0 %)
Parser Constr. (ϕ_{parser})	0 (0 %)	0 (0 %)	9 (3 %)	2 (0 %)
Type Checking Constr. (ϕ_{type})	9 (8 %)	15 (6 %)	1 (0 %)	3 (0 %)
Linker Constr. (ϕ_{linker})	11 (9 %)	21 (8 %)	1 (0 %)	19 (0 %)
Rule 1 (ϕ_{rule1})	17 (14 %)	39 (15 %)	26 (8 %)	1,522 (17 %)
Rule 2				
Feature effect Constr. (ϕ_{feffect})	7 (6 %)	14 (5 %)	2 (1 %)	58 (1 %)
Feature effect - Build Constr. ($\phi_{\text{feffect\_build}}$)	4 (3 %)	0 (0 %)	-	316 (4 %)
Rule 2 (ϕ_{rule2})	8 (7 %)	14 (5 %)	2 (1 %)	374 (6 %)
Full Code Constraints (ϕ_{code})	25 (21%)	57 (22 %)	28 (9 %)	4,461 (62 %)

5.3.1 Recoverability Discussion

We can see a pattern in terms of where variability-model hierarchy and cross-tree constraints are reflected in the code. Table 4 shows that a large percentage of hierarchy constraints can be automatically extracted. Specifically as shown in Table 6, 31 % of the hierarchy constraints can be automatically extracted. This suggests that the structure of the variability model (hierarchy constraints) often mirrors the structure of the code. Rule 2 alone can extract an average 24 % of the hierarchy constraints (see Table 6). An interesting case is Linux where already 27 % of the hierarchy constraints are mirrored in the nested directory structure in the build system (i.e., file presence conditions) as shown in Table 4. We conjecture that this results from the highly nested code structure, where most individual directories and files are controlled by a hierarchy of Makefiles, almost mimicking the variability model hierarchy [11], [37].

On the other hand, although harder to recover, Table 5 suggests that cross-tree constraints seem to be scattered across different places in the code (e.g., linker and type information), and seem more related to preventing build errors than hierarchy constraints are. Interestingly, Figure 4 shows that there is no overlap (with the exception of four constraint in uClibc and Linux) between the two rules we use to recover constraints. This aligns with the different reasoning behind them: *one is based on avoiding build errors while the other ensures that product variants are different*. The fact that our static analysis of the code could only recover 28 % of the variability-model constraints suggests that many of the remaining constraints require different types of analysis or stem from sources other than the implementation. We look at this issue in more details in our third objective in the next section.

6 Constraint Classification

To classify the different types of configuration constraints in order to address **Objective 3**, we aggregate and cross-validate data from four types of analyses that act as different data sources. First, we elicit developer feedback in the form of phone interviews and online questionnaires and use grounded theory [17] to analyze this data. Second, we use the recoverability results from our automated

TABLE 6: Summary of overall recoverability results[1]

	uClibc	BusyBox	eCos	Linux	Overall (Geom. Mean)
Rule 1					
Hierarchy	1 (2 %)	2 (1 %)	4 (1 %)	306 (6 %)	2 %
Cross-tree	17 (14 %)	39 (15 %)	26 (8 %)	1,522 (17 %)	13 %
All constraints	18 (10 %)	41 (6 %)	30 (3 %)	1,828 (14 %)	7 %
Rule 2					
Hierarchy	9 (17 %)	251 (69 %)	48 (8 %)	1,663 (33 %)	24 %
Cross-tree	8 (7 %)	14 (5 %)	2 (1 %)	374 (6 %)	4 %
All constraints	17 (10 %)	265 (42 %)	50 (6 %)	2,037 (16 %)	14 %
Full Code (Rules 1 & 2 conjoined)					
Hierarchy	14 (26 %)	265 (72 %)	53 (9 %)	2,569 (51 %)	31 %
Cross-tree	25 (21 %)	57 (22 %)	28 (9 %)	4,461 (62 %)	23 %
All constraints	39 (23 %)	322 (51 %)	81 (9 %)	7030 (55 %)	28 %

[1] Highlighted numbers are those used in the text for easier referencing.

analysis. Third, we conduct a manual analysis of a sample of the non-recovered results to understand why these constraints are enforced. Fourth, we perform additional automated analysis to count certain types of constraints which were discovered using one or more of the previous three analyses.

6.1 Setup and Preliminary Results

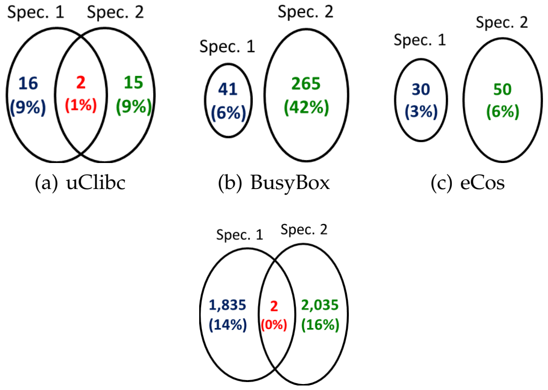
We now describe the setup for the four types of analyses we use to understand and classify constraints. We also show the raw results where applicable. The raw results from the four types of analyses are later aggregated into classification categories in Section 6.1.5.

6.1.1 Developer Interviews

We elicit feedback about configuration constraints from 27 developers across the four systems. We describe how we contact developers, gather the data, and analyze it below.

Developer Recruitment. For each of the four subject systems, we query each system's respective source control repository to identify a list of developers who have made changes to the configuration files encoding the variability models. We then contact the identified developers via email, and give them the choice to participate through a phone interview or a questionnaire. This is done to cater for different developer preferences and availability. A total of 22 developers answered our questionnaires and we conducted phone interviews with 5 developers. Table 7 shows the number of developers per system who participated in our study through both questionnaires and phone interviews.

Questionnaire/Interview Structure. Online questionnaires had a specific set of open-ended questions for the developers to answer without our intervention. For each system, we additionally provided three to four examples of constraints that we could not recover and asked developers to explain why such dependencies are enforced. Interviews, on the other hand, were semi-structured [45] and lasted an average of 34 minutes. While we had the same list of questions and examples from the questionnaire in mind during the phone interviews, we allowed the conversation to steer away from these fixed questions depending on the developer's responses. Thus, each interview was shaped by the developer's responses and their willingness to share information. Questions in both the interviews and questionnaires revolved around the following themes:



(a) uClibc (b) BusyBox (c) eCos

(d) Linux

Fig. 4: Overlap between Specifications 1 and 2 in recovering variability-model constraints. An overlap means that the same model constraint can be recovered by both rules.

TABLE 7: Number of developers interviewed through questionnaires and phone conversations

System	Questionnaires	Phone Interviews	Total
uClibc	2	1	3
BusyBox	1	2	3
eCos	1	0	1
Linux	18	2	20
Total	22	5	27

- When is a feature added to the variability model?
- When is a dependency enforced?
- How can we extract configuration constraints?

Answers from all three themes help us understand when dependencies are enforced and how they can be extracted. For the third question, we ask developers about our two extraction rules. This included their view on valid versus invalid configurations as well as nesting of #IFDEFs in code.

Data Analysis. To analyze the data we collected from the 27 developer responses, we first transcribe all phone interviews. Since our study is of exploratory nature, we use grounded theory [17] to analyze the data from both interviews and questionnaires where we use open-coding to identify key sources of dependencies. For the interview data, participants are coded for anonymity. We give each developer a code showing the system (*UC* for uClibc, *BB* for BusyBox, *EC* for eCos, and *LI* for Linux) followed by a number.

TABLE 8: Manual analysis of non-recovered constraints. We could explain 58 % of the non-recovered constraints through four cases, but could not determine the rationale for the remaining constraints[1]

Case	Description	Number of Constraints	% Adjusted to Constraint Population
1. Additional analysis required	Can be recovered using more expensive analysis	30	16 %
1.1 Data-flow analysis or testing		16	9 %
1.2 More specific analysis		14	8 %
2. More relaxed code constraints	Extraction relates two features but is less strict than the variability model	27	15%
3. Domain knowledge	At least one of the features is not used in the code & relation can only be identified through expert knowledge	40	22 %
3.1 Configurator-related		27	15 %
3.2 Platform or hardware knowledge		13	7 %
4. Limitation in extraction	We do not support non-boolean comparisons and C++ code	5	2 %
5. Unknown	We could not determine the rationale behind the enforced constraint	42	23 %

[1] Highlighted numbers are those used in the text for easier referencing.

6.1.2 Automated Classification

To facilitate parts of the investigation, we use the recoverability results from Section 5.3 to automatically classify a large number of constraints as technical and statically discoverable. Specifically, our analysis shows that 28 % of the constraints are code dependencies which can be automatically extracted (see Table 6).

6.1.3 Manual Analysis

To understand what other categories of constraints exist, we randomly sample 144 non-recovered constraints (18 hierarchy and 18 cross-tree constraints from each subject system). We then divide these constraints among the authors of the paper for manual investigation. The goal is for each author to try to identify the reason behind enforcing these non-recovered constraints by manually looking at the implementation as well as reading the documentation of the features involved. In the process, we also try to understand why these constraints could not be recovered using our automated analysis. Each author records the findings for each of their assigned set of constraints. At the end of the process, one author went through all the recorded reasons in order to categorize them. Thus, 75 % of the studied constraints are cross-validated by two authors. In Table 8, we show the raw data from this analysis, where we summarize the five cases that explain the analyzed sample of non-recovered constraints. In the last column, we show the percentages generalized to the general constraint population [2], which we use in the results below.

6.1.4 Additional Automated Analysis

Our interviews and manual analysis showed that there are certain patterns of constraints such as those containing feature(s) not used in the code or those containing features related to hardware or platform restrictions. We automatically count the first case by checking all variability-model constraints to see how many constraints have such features. We find that 21 % of constraints across the four systems have at least one feature not used in the code. For the second case, we count the constraints in BusyBox which contain the hardware feature PLATFORM_LINUX which we came across in the manual analysis and which was also discussed during the interviews. We find that 110 out of the 366 cross-tree constraints in BusyBox contain this feature.

6.1.5 Integrating Our Results

Our results from the four types of analyses above suggest that there are four cases for enforcing configuration constraints: *(1) enforcing low-level code dependencies, (2) ensuring correct run-time behavior, (3) improving the user's configuration experience, and (4) avoiding corner cases*. In the next four subsections, we describe each of the four cases along with relevant examples and supporting findings. Our aim is to understand when dependencies are enforced and if such dependencies can be automatically extracted. Thus, for each case, we provide examples, describe how developers think such dependencies can be identified, and deduce what implications this might have on automatic extraction tools. We summarize how the classification categories are supported by the different data sources in Table 9.

6.2 Enforcing low-level code dependencies

Large configurable systems are designed to be modular such that there is a procedure, file, or component that is responsible for each functionality. Due to such modular design, it is common that one feature may need to use

[2] To determine the generalized percentages, we have to also consider that we have already automatically recovered 28 % of the existing configuration constraints. Thus, there is a remainder of 72 % of non-recovered constraints from which we obtain our sample. Thus, getting the generalized percentage of constraints representing domain knowledge in Table 8 can be found as follows: $(40/144) * 0.72 = 20\%$. The same can be applied for the other cases.

TABLE 9: Summary of the supporting data sources for each category of configuration constraints.

	Developer Interviews	Automated Recovery Results	Manual Analysis	Additional Automated Analysis
Enforcing Low-level Code Dependencies	✓	✓	✓	
Ensuring Correct Run-time Behavior	✓		✓	✓
Improving the User's Configuration Experience	✓		✓	✓
Avoiding Corner Cases	✓			

certain functionalities offered by another feature. This can be in the form of low-level code dependencies such as using symbols defined in a different feature. When such relationships exist, dependencies between the related features have to be enforced in the variability model to allow the system to build successfully, as pointed out by many of the interviewed developers. For example, one of the uClibc developers explains it as follows:

> *"Kconfig dependencies usually express requirements [related to] internal libraries inside the project [as well as] requirements to avoid build failures because of functionality [needed from other modules]."* (UC_1)

6.2.1 Examples

When a module in the system relies on the definition of certain symbols from a different module, such dependency is marked in the variability model. We provided an example of such a low-level dependency between features X and Y in Listing 1a in Section 2.2.1. A real example from BusyBox is feature WHO which depends on feature FEATURE_UTMP. The WHO utility displays the current logged-in user. In order to display that user, the WHO applet needs access to the /var/run/utmp file which keeps track of the logged in users. This file is controlled by FEATURE_UTMP. On a low level, WHO uses function getutent to identify the current logged in user which is only defined if FEATURE_UTMP is selected.

6.2.2 Identification

We find that low-level dependencies represent at least 45% of configuration constraints as shown by our recoverability analysis and manual analysis results. This is based on the 28 % recovered by our analysis, 9 % related to data-flow (see Table 8), and 8 % related to additional code analysis (see Table 8). We believe that 45 % represents a lower bound since our manual analysis may have missed specific dependencies. Our interview data also suggests that low-level dependencies are the most common reason behind enforcing configuration constraints.

We now discuss specific code analysis techniques which can identify the various types of low-level dependencies we found in our data.

Build and linker analysis: Our recoverability results show that 28 % of the constraints can be extracted using our TypeChef and FARCE infrastructures. Rules 1 and 2 rely on ensuring that the system builds and links correctly (i.e., all low-level dependencies are respected) and that the selection of a feature changes something in the code. This is confirmed by the developers we interviewed, who explained that low-level code dependencies can be found by analyzing the source code files as well as the build files. They mainly suggest studying def/use chains of symbols and looking into linker failures to determine missing symbols signifying dependencies. Most of them suggested that imitating the linker and the C preprocessor would be the best way to determine such dependencies. A Linux developer summarizes the best way to identify low-level code dependencies as follows, confirming our extraction methodology:

> *"[I would] identify interfaces (functions, variables, macros) protected by #ifdefs, and identify translation units that use the protected interfaces and whether the use is also protected by other #ifdefs. [I would also] look into the build files. [However,] this would only identify build-time dependencies."* (LI_7)

Data-flow analysis: During the interviews, one developer pointed out that dependencies may result from code that implicitly depends on initialization or value updates that are done somewhere else. Detecting such cases requires data-flow analysis. This is also confirmed through our manual analysis where we found that 9% of constraints might be recovered through data-flow analysis. While our extraction infrastructure does not yet support data-flow analysis, there is existing research that can be used as a basis to create a variability-aware data-flow analysis which can scale to large systems [15], [16], [32].

Additional analysis: Our manual analysis shows that 8% of configuration constraints can be recovered through more specific analysis. This includes more advanced build system analysis than what we currently support or system-specific analysis such as the use of applets in BusyBox or the kernel module system in the Linux kernel.

6.2.3 Implications for Extraction Tools

The above discussion shows that dependencies extractable from the code and build files (with varying degree of cost) can account for a large portion of configuration constraints. This is promising for automatic extraction tools. However, there are still some challenges preventing a complete extraction which we show with the following two cases we learn from developer interviews.

Deferring problems to runtime: Developers explain that they often use *static inline function stubs* to *prevent* build-time errors from occurring as shown in Figure 5. In this case, function pci_register_driver in pci.h (Figure 5a) is defined if feature PCI is selected and is defined as a static inline function returning zero if PCI is not selected.

```
#ifdef CONFIG_PCI
  int pci_register_driver(struct pci_driver *drv);
#else
/*
 * If the system does not have PCI, clearly these return errors.
 * Define these as simple inline functions to avoid hair in drivers.
 */
static inline int pci_register_driver(struct pci_driver *drv)
{
  return 0;
}
#endif
```

(a) pci.h

```
//File PC = SCSI_QLA_ISCSI && SCSI

#include <pci.h>

int qla4xxx_module_init(void){
  int ret;
  ...
  ret = pci_register_driver(&qla4xxx_pci_driver);
  if (ret)
    goto unregister_transport;
  ...
  return 0;

  unregister_transport:
    iscsi_unregister_transport(&qla4xxx_iscsi_transport);
}
```

(b) ql4_os.c

Fig. 5: Static inline function definitions may prevent tools from detecting low-level code dependencies.

The function is then used in ql4_os.c (Figure 5b), which is only compiled if features SCSI_QLA_ISCSI and SCSI are selected. A developer looking at this code can understand that there is a relationship between SCSI_QLA_ISCSI and PCI, since function pci_register_driver registers this SCSI driver only if PCI is selected. In the case when PCI is not selected, the function will be defined as the empty stub returning zero, which would result in the driver not being registered. However, a regular compiler, or a static analysis infrastructure such as ours, would not detect this relationship, since the static inline function prevents any type errors from occurring. As one uClibc developer points out:

> *"A lot of projects like the Linux kernel will provide static inline stubs which replace the actual implementation when a specific Kconfig symbol is turned on/off, specifically to avoid build failures."* (UC_1)

This suggests that developers may intentionally push handling meaningless or incorrect behavior to runtime rather than handling such problems at build-time. Using static inline functions to accomplish such behavior is actually part of the recommended practices in the Linux kernel guidelines.[3] Detecting such situations would require project-specific heuristics which are hard to generalize (and automate).

Mixing run-time and build-time behavior: Several developers also mention the tendency to move to using C-based if checks rather than preprocessor-based #IFDEF checks, which represents a second challenge for automatic extraction tools. With if-based checks, code elimination is left to the compiler where a new macro that is defined to 1 is created if the feature is selected. If the feature is not selected, this new macro is defined to 0 (triggering dead-code elimination). We show such an example in Figure 6, where the same conditional compilation is achieved with the C preprocessor or through a regular C if check, assuming dead-code elimination in the compiler. This achieves a similar effect to the C preprocessor, but while avoiding syntax error problems (often appearing in certain configurations only) which may be caused by #IFDEF checks. On the contrary, if there is a syntax error in the if check, the build will break on all configurations and not just a specific one, making it easier to debug (BB_3). This involvement of run-time behavior again complicates dependency extraction for automated tools, since other runtime checks which cannot be evaluated at build-time can be mixed with the feature check in the if condition [33], [59].

6.3 Ensuring Correct Run-time Behavior

Apart from enforcing low-level dependencies that are found in the system's implementation, we find that dependencies are also enforced to ensure correct run-time behavior as indicated by all four data sources. That is, the system may use external libraries or platform functionalities that are only known at runtime. Additionally, this category of constraints prevents functionalities which will not be beneficial on certain platforms from being selected.

6.3.1 Examples

As an example of a runtime dependency enforced by configuration constraints, any feature in BusyBox which relies on the proc file system would only work on Linux environments. Therefore, such a feature would always depend on the PLATFORM_LINUX feature. As a developer explains, such features might compile normally without the PLATFORM_LINUX selected, but then if they try to open files in proc, nothing will happen since the proc file system is not available on non-Linux platforms.

A similar example from the Linux kernel found during our manual analysis is SERIO_CT82C710 → X86_64. The first feature controls the port connection on that particular chip, but only works with an X86_64 architecture. We can also tell that such platform restrictions are common from our manual analysis and our additional automated analysis. From our manual analysis, we find that 7% of constraints have at least one feature that is not used in the

[3] http://www.kernel.org/doc/Documentation/SubmittingPatches

```c
static void putprompt (const char *s)
{
  #ifdef CONFIG_ASH_EXPAND_PRMT
    free((char*)cmdedit_prompt);
    cmdedit_prompt = ckstrdup(s);
    return;
  #endif
  cmdedit_prompt = s;
}
```

(a) Preprocessor-based checks

```c
static void putprompt (const char *s)
{
  if (ENABLE_ASH_EXPAND_PRMT) {
    free((char*)cmdedit_prompt);
    cmdedit_prompt = ckstrdup(s);
    return;
  }
  cmdedit_prompt = s;
}
```

(b) C-based checks

Fig. 6: Relying on the C compiler (b) versus relying on the C preprocessor (a) for conditional compilation.

code and is related to some platform or hardware restriction. In the specific example of BusyBox, our additional automated analysis shows that 110 out of 366 cross-tree constraints involve the feature PLATFORM_LINUX indicating that such hardware restrictions are very common.

A different yet similar example provided by developers is including code that uses a PIE executable (Position Independent Executable) on a platform that cannot benefit from it, increasing the binary size by 20% for no purpose. Such an example shows that the objective behind runtime dependencies is not only to prevent a run-time error from occurring, but also to prevent functionalities which will not be beneficial on a specific platform.

6.3.2 Identification

Run-time dependencies are very common. We find that run-time dependencies are usually identified through domain knowledge or testing.

Domain Knowledge: Our interviews show that often developers simply rely on their domain knowledge to identify run-time dependencies. Different developers across the subject systems provide similar comments about how identifying many of these dependencies basically ends up coming down to experience. This experience is gained from working with several hardware boards and knowledge of previous, similar problems which might have occurred. This aligns with our manual analysis findings where we could not explain 29 % of the constraints we manually analyzed in our sample[4] or found constraints where the relationship can only be determined through domain knowledge.

Testing: During our interviews, we find that developers do not always know of all such dependencies. Some of these run-time dependencies are only found through testing. As explained to us by several developers, what happens in practice is a trial and error process. When the system is being configured for a new board, for example, developers select the configuration they believe should work (based on their expertise). They then test this configuration and correct any problems which may arise. Simply put, “you configure it until it works” (LI_19). However, since there are many different hardware devices to test for, some of the dependencies are not known until a user reports some problem on a specific board, for example. Thus, additional configuration constraints may be identified at a later stage through user testing. The following two quotes illustrate this:

> *“In my experience, there is also a great deal of empirical build testing which implied adding a dependency. Since [the build system] only takes care of [the] build time aspect of a specific software, runtime dependencies are sorted out differently.”* (UC_1)
>
> *“You catch everything with your knowledge, and the remainder comes from user testing (which then expands your knowledge, obviously).”* (BB_2)

This aligns with findings from configuration testing in other domains (e.g., [21]).

6.3.3 Implications for Extraction Tools

Since identifying run-time dependencies stems from either testing or domain knowledge, this seems like a limitation for automatic extraction tools. Static analysis of the code would not reveal these dependencies. An option would be to perform some testing on each, or a representative sample, of the supported platforms to see which configurations work or fail [39]. However, this is a very costly and time-consuming process due to its reliance on the availability of hardware components. That said, there may also be ways to find such dependencies from the code as we show below.

Relying on Feature Effect: While many of the features representing hardware components or platform support may not be used in the code, some of them are. For example, checks for PCI or 64 bit support are sometimes used in the code in the form of #IFDEF checks or in the form of file presence conditions. In this case, our feature effect heuristic reflected in Rule 2 may be used to recover such dependencies. As shown in our empirical, recoverability results, Rule 2 alone can already extract 24 % of the hierarchy constraints. A quick look at the hierarchy constraints suggests that several of them are related to hardware features, such as PCI or NET_ETHERNET. However, this can only be used as a heuristic and will not be able to identify all such constraints, as some developers may not add such checks in the code and will rely on the variability model enforcing the right feature combinations. One BusyBox

[4] From Table 8: 42/144 = 29 %

developer elaborates about this when explaining why a dependency in the variability model might be more strict than its corresponding check in the code:

> *"You do not need to test for things that are already covered implicitly by something else elsewhere. You basically only need tests that prevent something you do not want. If the thing you do not want cannot happen anyways, you do not need to test for [it]."* (BB_3)

Apart from the above problem, the feature effect heuristic also cannot differentiate between feature dependencies and feature interactions. We provide more details about this problem in Appendix A.

To overcome such challenges, relying on experts' domain knowledge might be the best alternative depending on the availability of such experts. After automatically detecting build-time dependencies, and feature effect dependencies, the information can be presented to such experts where they can add the dependencies related to run-time behavior. This is of course a manual process, but it might be the most effective method for identifying such dependencies when starting with partial knowledge. This is also supported by the fact that our additional automated analysis found that 21% of the existing model constraints have at least one feature not used in the implementation, which means that such constraints cannot be automatically extracted.

6.4 Improving the User's Configuration Experience

In all the systems we analyzed, the variability model is used as a backend to a configurator that guides the user during the configuration process. Such a configurator contains menus and sub-menus, as well as other groupings, which facilitate the configuration process such that the user is not overwhelmed with all features at once. According to developers, as well as our manual analysis, some dependencies exist in the variability model only to support such an organization.

We believe that such dependencies are common. Our manual analysis shows that 15 % of constraints are related to organizing things in the configurator. Our additional automated analysis shows that 21 % of configuration constraints contain at least one feature not used in the code. However, we cannot automatically determine if such feature(s) are related to the configurator or not. They might instead represent hardware features as discussed in the previous section.

6.4.1 Examples

In Kconfig-based systems, menu and menuconfig items are mainly used to create menus and groupings, even though they could still be used in the code. BusyBox developer BB_3 tells us that menu symbols or grouping symbols are used to avoid overwhelming the user with complexity. For example, you cannot select specific network cards in the configurator unless the NETWORKING feature is switched on. Thus, in some cases, a hierarchy constraint may exist for presentation purposes in the configurator rather than because of any technical dependencies.

A similar example is that of the Linux kernel hierarchy constraint IPC_NS → NAMESPACES, which we could not recover. In this example, NAMESPACES is not used in the code at all even though it is a regular feature (not a menu or menuconfig). From the developers, we learn that NAMESPACES is just used in Kconfig for organization purposes, such that all namespace-related features can be disabled/enabled with one click rather than individually choosing the features. It is interesting to note that in subsequent releases of the Linux kernel, NAMESPACES has been changed into a menuconfig item instead of a config item, which makes its organization role in Kconfig more obvious.

A similar example is the BusyBox feature DESKTOP. A dependency on DESKTOP is added to features which only work on desktop environments. Thus, users configuring a non-desktop environment would not even see these features, preventing clutter and confusion with non-relevant features during configuration.

6.4.2 Identification

It seems that identifying which features need to be grouped together and how features should appear in the configurator is mainly related to common sense and domain knowledge. Developers know which features are related and, thus, group them together in the same menu. We come up with the same conclusion from our manual analysis as well.

6.4.3 Implications for Extraction Tools

Since configurator-related dependencies are not necessarily reflected in the code, there is no way for automatic detection tools to extract them. However, using certain heuristics may work depending on the project. For example, if the list of all features is available, a name-based heuristic similar to that used by She et al. [48] can be used to identify related features which may appear in the same menu. For example, CONFIG_NET_KEY may be nested under a CONFIG_NET menu because their names are similar.

6.5 Avoiding Corner Cases

Our interview data reveals that developers may enforce dependencies to prevent reasonable corner cases that are not supported yet. Only a couple of developers mentioned this so we believe it is not as common as the previous three cases. As a Linux developer puts it:

> *"Programmers are not interested in all corner cases, so support for these configurations may be papered over using dependencies."* (LI_11)

6.5.1 Examples

One BusyBox developer (BB_3), who is also familiar with the Linux kernel, points out that there was a configuration option BROKEN where unsupported functionalities would depend on it to indicate that they are not fully supported.

Another developer provides a more specific example from the Linux kernel where a crash occurred because of the futex[5] code on the m68k architecture. Rather than generalizing the futex code to work properly on all architectures, the code was changed such that the check causing the crash can be skipped depending on the value of feature HAVE_FUTEX_CMPXCHG. On any architecture that suffers from this runtime crash, a dependency from the feature representing that architecture to the FUTEX_CMPXCHG feature is added to prevent the crash. As the developer explains,

> *"If the various architectures represent corner-cases, the Kconfig solution avoids the need to generalize the futex code to cope with them all."* (LI_11)

6.5.2 Identification

Developers are familiar with the system and its domain and can determine which cases should be supported and which cases may be very rare such that they can be temporarily or permanently ignored.

6.5.3 Implications for Extraction Tools

Knowledge of which configurations are prevented due to avoiding corner cases is hard to automatically detect, unless developers explicitly mark such corner cases with #ERROR directives or explicitly document them otherwise. In this case, these dependencies can be identified with extraction tools such as ours. Otherwise, automatic extraction tools (even if performing runtime analysis) cannot identify such cases. We believe documenting dependencies (when applicable) with #ERROR directives is a good practice since it makes it easier to ensure that they are enforced in the variability model. Using #ERROR directives also makes it easier to understand the rationale behind an enforced constraint as well as clearer mapping from the variability model to where the constraint is enforced in the code.

6.6 Constraint Classification Discussion

Our recoverability results from Section 5.3 show that 28 % of existing configuration constraints are low-level code dependencies, statically discoverable from the code. Based on data from our other data sources, we believe that low-level implementation dependencies represent at least 45 % of configuration constraints, which is very promising to automated tools.

On the other hand, the three other categories of configuration constraints we found show that automated analysis is not sufficient to extract a complete variability model. We presented heuristics that may be used in conjunction with developer input. Such heuristics suggest that some of developers' domain knowledge may still be found in the implementation. However, since identifying many of the problem-space constraints relies on domain knowledge from developers, this emphasizes the need for explicit variability models to document such knowledge.

7 Threats to Validity

7.1 Internal validity

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

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

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

Classification. Our classification categories are based on our interpretation and grouping of the data, but since

[5] Short for "fast user-space mutex", which is a Linux kernel system call that programmers can use to implement basic locking

we rely on several data sources, we benefit from cross-validation of findings. However, there may be additional categories which have not been revealed from our data sources.

7.2 Construct validity

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

7.3 External validity.

Developer feedback. In our qualitative study, we managed to get the feedback of only one eCos developer due to the poor response rate. However, due to the similar nature of eCos to the other systems (apart from using a different configuration language), we believe that comments from the developers of the other systems would still apply to it. This is especially true since the comments from the eCos developer we interviewed aligned well with many of our findings from the other developers.

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

8 Related Work

This work builds upon, but significantly extends our prior work. We reuse the existing TypeChef analysis infrastructure for analyzing #ifdef-based variability in C code with build-time variability [28], [29], [32]. However, we use it for a different purpose and extract constraints from various intermediate results in a novel way, including an entirely novel approach to extract constraints from a feature-effect heuristic. Furthermore, we double the number of subject systems in contrast to prior work (before the conference version of this paper). The work is complementary to our prior reverse-engineering approach for feature models [48] (an academic variability modeling notation [26]), where we showed how to get from constraints to a feature model suitable for end users and tools. Now, we focus on deriving constraints in the first place which also involves understanding where these constraints come from. This paper is an extended version of a previous conference publication [35]. We extended the comparisons to include a global code formula aggregated from all extracted constraints from different sources to account for interactions between constraints from different analyses, which better represents the recoverability results in Section 5.3. Doing this increases the recoverability of existing variability-model constraints from 19 % in the previous conference publication [35] to 28 % in this paper. We significantly extended our analysis of different constraints in practice, adding a qualitative study (interviews and surveys), several additional automated analyses, and a new analysis triangulating all those results into the categories presented in Section 6. These additional analyses enable our discussion of when configuration constraints are enforced in practice.

Techniques to extract features and their constraints have been developed before, mainly to support the re-engineering, maintenance, and evolution of highly-configurable systems. From a process and business perspective, researchers have developed approaches to re-engineer existing systems into an integrated configurable system [7], [14], [49], [52]. These approaches include strategies to make decisions: when to mine, which assets to mine, and whom to involve. Others have developed re-engineering approaches by analyzing non-code artifacts, such as product comparisons [20], [23]. In contrast to techniques using non-code and domain information, we extract *technical constraints* from code with #IFDEF variability.

From a technical perspective, previous work has also attempted to extract constraints from code with #IFDEF variability [30], [48], [54]. Most attempts focus on the preprocessor code exclusively [30], [54], looking for patterns in preprocessor use, but do not parse or even type check the underlying C code. That is, they are (at most) roughly equivalent to our partial-preprocessor stage. Prior attempts to parse unpreprocessed code typically relied on heuristics (unsound) [40] or could only process specific usage patterns (incomplete) [6]. For instance, our previous work [48] used an inexact parser to approximate parts of our Rules 1 and 2. Our new infrastructure is sound and complete [28], allowing *accurate* subsequent syntax, type, and linker analyses.

Complementary to analyzing build-time #IFDEF variability, some researchers have focused on load-time variations through program parameters. Rabkin and Katz design an approach to identify load-time options from Java code, but not constraints among them [43]. Similarly, Lillack et al. [33] track the use of load-time configuration options through static taint analysis to identify the code parts they influence, but do not track the dependencies between such options. Reisner et al. [44] use symbolic execution to identify interactions and constraints among configuration parameters by symbolically executing a system's test cases. Such dynamic analysis can identify additional constraints as discussed in Section 6.2.2. However, scalability of symbolic execution is limited to medium size systems (up to 14K lines of code with up to 30 options in [44]), whereas our build-time analysis scales to systems as the Linux kernel. We also avoid using techniques such as data-flow analysis [15], [16], [32] due to scalability issues. In future work, although challenging to scale, we plan to investigate additional analysis approaches that track load-time and runtime variability (e.g., from command-line parameters). Data-flow analysis, symbolic execution, and testing tailored to variability [15], [32], [38], [44] are interesting starting points.

Finally, researchers have investigated the maintenance and evolution of highly configurable systems. There has been a lot of research directed at studying and ensuring the consistency of the problem and solution spaces [57]. However, most of this work has analyzed features in isolation, either in the problem space [13], [42], [47], [58] or in the solution space [31], [51] to identify modeling practices and feature usage. Some work has also looked at both sides to study co-evolution [34], [41] or to detect bugs due to inconsistencies between models and code [28], [29], [36], [54], [55]. While our results can enhance these consistency checking mechanisms, our goal is to clarify where constraints arise from and to demonstrate to what extent we can extract model constraints from the code.

9 Conclusion

As large configurable systems become more common, variability models will become more essential to effectively manage and maintain such systems. Identifying configuration constraints is directly related to creating such variability models. However, there has not been enough work about how developers identify configuration constraints in practice and what knowledge do such constraints reflect. Additionally, there are no automated techniques to accurately identify configuration constraints in large-scale systems.

We address both problems by engineering static analyses to extract configuration constraints and by performing a large-scale study of constraints in four real-world systems. The objectives of this study are to *(1) evaluate accuracy and scalability*, *(2) evaluate recoverability*, and *(3) classify constraints*.

Our results show that manually extracting technical constraints is very hard for non-experts of the systems, even when they are experienced developers. We experienced this first-hand, giving a strong motivation for automating the task. With respect to *Objective 1*, we show that automatically extracting accurate configuration constraints from large codebases is feasible to a large degree and that our analyses scale. We can recover constraints that in almost all (93 %) cases assure a correct build process. Additionally, our new feature effect heuristic is surprisingly effective (77 % accurate). With respect to *Objective 2*, we find that variability models contain much more information than we can recover from code. Although our scalable static analysis can recover more than a quarter (28 %) of the model constraints, additional analyses and external information may be needed. With respect to *Objective 3*, our qualitative study involving 27 developers and manual analysis identifies four cases where configuration constraints are enforced in the variability model:

- *Enforcing low-level code dependencies* to ensure that the system builds correctly. Build and linker analysis as well as data-flow analysis can extract such dependencies.
- *Ensuring correct run-time behavior* such that the system runs correctly and only contains functionality that would actually work at run-time. This usually involves platform dependencies where some functionalities only work on certain hardware. Such dependencies are usually identified from domain knowledge as well as testing (including user-testing).
- *Improving the user's configuration experience* through feature groupings and better constraint propagation in the configurator. Identifying which features are related usually depends on domain knowledge.
- *Avoiding corner cases* such that combinations of features leading to known, unsupported behavior are avoided. These can be identified through system expertise and domain knowledge as well as cases where this is explicitly marked by #ERROR directives.

Apart from the first case, we find that identifying the other cases creates obstacles for automated analysis tools since these are often known through expert knowledge or through user testing. We believe that using automated extraction tools such as ours in addition to eliciting domain knowledge and feedback from expert developers may be the best way to create complete variability models.

Acknowledgments

We would like to thank all the developers who participated in our study. This work has been partly supported by NSERC CGS-D2-425005, ARTEMIS JU grant n° 295397 VARIES, Ontario Research Fund (ORF) Project on Software Certification, and NSF grant CCF-1318808.

References

[1] CDLTools. https://bitbucket.org/tberger/cdltools.

[2] KBuildMiner. http://code.google.com/p/variability/wiki/PresenceConditionsExtraction.

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

[4] Online appendix. http://gsd.uwaterloo.ca/farce.

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

[6] 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 Computer Society, 2001.

[7] J. Bayer, J.-F. Girard, M. Würthner, J.-M. DeBaud, and M. Apel. Transitioning legacy assets to a product line architecture. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 446–463. Springer, 1999.

[8] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615 – 636, 2010.

[9] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski. A survey of variability modeling in industrial practice. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 7:1–7:8, 2013.

[10] T. Berger and S. She. Formal semantics of the CDL language. Technical Note. Available at www.informatik.uni-leipzig.de/~berger/cdl_semantics.pdf.

[11] T. Berger, S. She, K. Czarnecki, and A. Wąsowski. Feature-to-Code mapping in two large product lines. Technical report, University of Leipzig, 2010.

[12] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. A study of variability models and languages in the systems software domain. *IEEE Transactions on Software Engineering*, 39(12):1611–1640, 2013.

[13] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. Variability modeling in the real: A perspective from the operating systems domain. In *Proceedings of the International Conference Automated Software Engineering (ASE)*, pages 73–82. ACM Press, 2010.

[14] J. Bergey, L. O'Brian, and D. Smith. Mining existing assets for software product lines. Technical Report CMU/SEI-2000-TN-008, SEI, Pittsburgh, PA, 2000.

[15] E. Bodden, M. Mezini, C. Brabrand, T. Tolêdo, M. Ribeiro, and P. Borba. Spllift - statically analyzing software product lines in minutes instead of years. In *Proceedings of the Conference Programming Language Design and Implementation (PLDI)*, pages 355–364. ACM Press, June 2013.

[16] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba. Intraprocedural dataflow analysis for software product lines. In *Proceedings of the International Conference Aspect-Oriented Software Development (AOSD)*, pages 13–24. ACM Press, 2012.

[17] J. Corbin and A. Strauss. *Basics of Qualitative Research. Techniques and Procedures for Developing Grounded Theory*. 3rd edition, 2008.

[18] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, MA, 2000.

[19] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Proceedings of the International Conference Generative Programming and Component Engineering (GPCE)*, pages 211–220. ACM Press, 2006.

[20] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang, and P. Heymans. Feature model extraction from large collections of informal product descriptions. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 290–300. ACM Press, 2013.

[21] N. Devos, C. Ponsard, J.-C. Deprez, R. Bauvin, B. Moriau, and G. Anckaerts. Efficient reuse of domain-specific test knowledge: An industrial case in the smart card domain. In *Proceedings of the International Conference Software Engineering (ICSE)*, pages 1123–1132, June 2012.

[22] D. Dhungana, P. Grünbacher, and R. Rabiser. The DOPLER metatool for decision-oriented variability modeling: A multiple case study. *Automated Software Engineering*, 18(1):77–114, 2011.

[23] N. Hariri, C. Castro-Herrera, M. Mirakhorli, J. Cleland-Huang, and B. Mobasher. Supporting domain analysis through mining and recommending features from online product listings. *IEEE Transactions on Software Engineering*, 39(12):1736–1752, 2013.

[24] P. G. Hinman. *Fundamental of mathematical logic*. Peters, 2005.

[25] A. Hubaux, Y. Xiong, and K. Czarnecki. A user survey of configuration challenges in Linux and eCos. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 149–155. ACM Press, 2012.

[26] K. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, Pittsburgh, PA, 1990.

[27] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol.*, 21(3):14:1–14:39, July 2012.

[28] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the International Conference Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824. ACM Press, Oct. 2011.

[29] C. Kästner, K. Ostermann, and S. Erdweg. A variability-aware module system. In *Proceedings of the International Conference Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, 2012.

[30] D. Le, H. Lee, K. Kang, and L. Keun. Validating consistency between a feature model and its implementation. In *Safe and Secure Software Reuse*, volume 7925, pages 1–16. Springer, 2013.

[31] 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 Software Engineering (ICSE)*, volume 1, pages 105 –114, 2010.

[32] 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/Foundations of Software Engineering (ESEC/FSE)*, pages 81–91. ACM Press, 2013.

[33] M. Lillack, C. Kästner, and E. Bodden. Tracking load-time configuration options. In *Proceedings of the International Conference Automated Software Engineering (ASE)*, pages 445–456, 2014.

[34] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski. Evolution of the Linux kernel variability model. In *Software Product Lines: Going Beyond*, volume 6287, pages 136–150. Springer, 2010.

[35] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *Proceedings of the International Conference Software Engineering (ICSE)*, pages 140–151. ACM, 2014.

[36] S. Nadi and R. Holt. Mining Kbuild to detect variability anomalies in Linux. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 107–116, 2012.

[37] S. Nadi and R. Holt. The Linux kernel: A case study of build system variability. *Journal of Software: Evolution and Process*, 2013. Early online view. http://dx.doi.org/10.1002/smr.1595.

[38] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *Proceedings of the International Conference Software Engineering (ICSE)*, 2014.

[39] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):11:1–11:29, Feb. 2011.

[40] Y. Padioleau. Parsing C/C++ code without pre-processing. In *Proceedings of the International Conference Compiler Construction (CC)*, pages 109–125. Springer, 2009.

[41] L. Passos, J. Guo, L. Teixeira, K. Czarnecki, A. Wąsowski, and P. Borba. Coevolution of variability models and related artifacts: A case study from the Linux kernel. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 91–100. ACM Press, 2013.

[42] L. Passos, M. Novakovic, Y. Xiong, T. Berger, K. Czarnecki, and A. Wąsowski. A study of non-Boolean constraints in variability models of an embedded operating system. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 2:1–2:8. ACM Press, 2011.

[43] A. Rabkin and R. Katz. Static extraction of program configuration options. In *Proceedings of the International Conference Software Engineering (ICSE)*, pages 131–140. ACM Press, 2011.

[44] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *Proceedings of the International Conference Software Engineering (ICSE)*, pages 445–454. ACM Press, 2010.

[45] P. Runeson and M. HÃűst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[46] S. She and T. Berger. Formal semantics of the Kconfig language. Technical Note. Available at eng.uwaterloo.ca/~shshe/kconfig_semantics.pdf.

[47] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. The variability model of the Linux kernel. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, 2010.

[48] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse engineering feature models. In *Proceedings of the International Conference Software Engineering (ICSE)*, pages 461–470. ACM Press, 2011.

[49] D. Simon and T. Eisenbarth. Evolutionary introduction of software product lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, volume 2379, pages 272–282. Springer, 2002.

[50] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. Is the Linux kernel a software product line? In *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL)*, 2007.

[51] J. Sincero, R. Tartler, D. Lohmann, and W. Schröder-Preikschat. Efficient extraction and analysis of preprocessor-based variability. In *Proceedings of the International Conference Generative Programming and Component Engineering (GPCE)*, pages 33–42. ACM Press, 2010.

[52] C. Stoermer and L. O'Brien. MAP – Mining architectures for product line evaluations. In *Proceedings of the Working Conference Software Architecture (WICSA)*, pages 35–44. IEEE Computer Society, 2001.

[53] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen. Build code analysis with symbolic evaluation. In *Proceedings of the International Conference Software Engineering (ICSE)*, pages 650–660. IEEE Computer Society, 2012.

[54] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 47–60. ACM Press, 2011.

[55] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proceedings of the International Conference Generative Programming and Component Engineering (GPCE)*, pages 95–104. ACM Press, 2007.

[56] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake. Analysis strategies for software product lines. Technical Report FIN-004-2012, School of Computer Science, University of Magdeburg, Apr. 2012.

[57] 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*, 2014. to appear.

[58] T. Thüm, D. Batory, and C. Kästner. Reasoning about edits to feature models. In *Proceedings of the International Conference Software Engineering (ICSE)*, pages 254–264. IEEE Computer Society, 2009.

[59] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.

[60] B. Veer and J. Dallaway. The eCos component writer's guide. ecos.sourceware.org/ecos/docs-latest/cdl-guide/cdl-guide.html.

[61] J. White, D. Schmidt, D. Benavides, P. Trinidad, and A. Cortés. Automated diagnosis of product-line configuration errors in feature models. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 225–234. IEEE Computer Society, 2008.

[62] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki. Generating range fixes for software configuration. In *Proceedings of the International Conference Software Engineering (ICSE)*, pages 58–68. IEEE Computer Society, 2012.

[63] R. Zippel and contributors. kconfig-language.txt. available in the kernel tree at www.kernel.org.

Sarah Nadi is a post-doctoral researcher at the Software Technology Group (STG) at the Technische Universität Darmstadt in Germany. She received her PhD in 2014 from the University of Waterloo, Canada where she worked on detecting variability anomalies in software product lines and reverse-engineering configuration constraints. Her PhD work was supported by an NSERC Alexander Graham Bell Canada Graduate Scholarship. Her research interests include automated support for software development and maintenance, variability support for software product lines, build systems, and mining software repositories.

Thorsten Berger is a post-doctoral fellow in the Generative Software Development lab at the University of Waterloo, Canada. He received his PhD degree from the University of Leipzig, Germany. His dissertation was supported by a PhD scholarship from the German National Academic Foundation, awarded for outstanding academic achievements, and by grants from the German Federal Ministry of Education and Research, and the German Academic Exchange Service. He also participated in national and international research projects funded by the Federal Ministry of Education and Research and the European Union's Seventh Framework Program. His research interests comprise model-driven development, variability modeling for software product lines and software ecosystems, and variability-aware static analyses of source code and build systems.

Christian Kästner is an assistant professor in the School of Computer Science at Carnegie Mellon University. He received his PhD in 2010 from the University of Magdeburg, Germany, for his work on virtual separation of concerns. For his dissertation he received the prestigious GI Dissertation Award. His research interests include correctness and understanding of systems with variability, including work on implementation mechanisms, tools, variability-aware analysis, type systems, feature interactions, empirical evaluations, and refactoring.

Krzysztof Czarnecki is a professor of electrical and computer engineering at the University of Waterloo, Canada. Before coming to Waterloo, he was a researcher at DaimlerChrysler Research (1995-2002), Germany, focusing on improving software development practices and technologies in the enterprise, automotive, space, and aerospace domains. He coauthored the book Generative Programming: Methods, Tools, and Applications [18], which deals with automating software component assembly based on domain-specific languages. While at Waterloo, he has held the NSERC/Bank of Nova Scotia Industrial Research chair in Requirements Engineering of Service-Oriented Software Systems (2008-2013) and has worked on a range of topics in model-driven software engineering, including software-product lines and variability modeling, consistency management, and bidirectional transformations, and example-driven modeling. He received the Premier's Research Excellence Award in 2004 and the British Computing Society in Upper Canada Award for Outstanding Contributions to the IT Industry in 2008.

Appendix A
Feature Dependencies Versus Feature Interactions

When discussing our feature effect heuristic during the interviews, developers raise the problem of feature interactions. Developers indicate that #IFDEF A being nested in #IFDEF B may not necessarily mean that feature A depends on feature B (as implicitly assumed by Rule 2). It may be the case that A and B are independent but some special handling must be done if they are both selected. A specific example would be checking for PCI support in a particular board file. The board can be used without PCI, but if PCI is enabled, then it has to be initialized to be usable (interviewee LI_D1). In this case, #ifdef PCI would be nested within #ifdef BOARD. Assuming PCI does not appear anywhere else in the code, our Rule 2 would extract the constraint PCI → BOARD. However, as the developer explains in her quote, PCI is an independent feature. It can be selected with or without the board, but if PCI is selected with this board, some special handling must occur. Thus, it seems that the nesting of #IFDEFs may reflect both feature dependencies (given the high recoverability we found) and feature interactions. Automatically differentiating between these two cases would be difficult. This might also explain the lower accuracy associated with Rule 2.

Appendix B
The Comparison Problem

Based on our findings from interviewing developers in Section 6, it seems that Rule 1 should recover many of the existing variability-model constraints since many of these dependencies are related to low-level code dependencies. However, our quantitative results show that although Rule 1 is highly accurate, it only recovers a few of the variability-model constraints. For example, the type analysis in Linux extracts over a quarter million constraints which are 97% accurate (Table 2), and yet only recovers 3 cross-tree constraints in Table 5. Case 2 of our manual analysis (see Table 8) suggests that we might be extracting more relaxed constraints using our analysis leading to this discrepancy. To understand this more, we investigate the constraints extracted by Rule 1 and find that this discrepancy happens, to some degree, because of shortcomings of using propositional logic to compare the problem and solution spaces.

In the following subsections, we first present two situations which illustrate how the above discrepancy can happen: (1) having missing information and (2) extracting more relaxed constraints from the code. We then follow this with a more general discussion about the comparison problem.

B.1 Missing Information

For the first situation, we find that in some cases, a code constraint holds in the variability model, but does not contain all the related features found in the model. We now discuss such an example from BusyBox. The following is a constraint we extract from a preprocessor error in the code.

$$\text{LAST\_SMALL} \rightarrow \text{UTMP} \tag{1}$$

The following are the related constraints in the variability model.

$$\begin{align} \text{LAST\_SMALL} &\rightarrow \text{LAST} \tag{2}\\ \text{LAST} &\rightarrow \text{WTMP} \tag{3}\\ \text{WTMP} &\rightarrow \text{UTMP} \tag{4}\\ & \tag{5} \end{align}$$

Based on these three constraints, the following variability-model constraint can be deduced through transitive closure.

$$\text{LAST\_SMALL} \rightarrow \text{UTMP} \tag{6}$$

Given these constraints, the extracted preprocessor constraint LAST_SMALL → UTMP holds in the model by transitive closure as shown in Equation 6. Therefore, although there is no direct dependency between LAST_SMALL and UTMP in the model, an indirect one is created when all the constraints are combined together. This explains how the preprocessor constraint holds in the variability model.

On the other hand, if we come to compare the three direct variability-model constraints in Equations 2–4 to the extracted code constraints (the preprocessor constraint in Equation 1 in this case), we find that none of them can be recovered by the extracted code constraint. This is because features LAST and WTMP do not appear as part of the constraints extracted from the code. Thus, in this case, the extracted code constraint is accurate, but does not help in recovering any of the manually-modeled variability-model constraints because it has missing information about the remaining related features.

B.2 More Relaxed Constraints

The second situation we find where an extracted code constraint holds in the model, but does not recover any model constraints happens when the extracted code constraint is more relaxed than that enforced in the model. We now discuss such an example. The following is a constraint we extract from our analysis of parsing errors in BusyBox:

$$\begin{split} &\text{ADDUSER\_LONG\_OPTIONS} \rightarrow \\ &\qquad \text{GETOPT\_LONG} \vee \text{LONG\_OPTS} \vee \neg\text{ADDUSER} \end{split} \tag{7}$$

The related variability-model constraint is as follows:

$$\text{ADDUSER\_LONG\_OPTIONS} \rightarrow \text{ADDUSER} \land \text{LONG\_OPTS} \tag{8}$$

In this case, the disjunction in the extracted parser constraint makes it more relaxed such that if any of the clauses of the disjunction are true, the whole constraint would hold in the variability model. In this example, we see in Equation 8 that in order for ADDUSER_LONG_OPTIONS to be selected in the variability model, LONG_OPTS must also be selected. This means that one clause from the disjunction in the extracted code constraint will always hold causing the whole implication to always hold in the variability model. This is although other clauses of the disjunction such as depending on ¬ADDUSER do not necessarily make sense. However, the variability model still guarantees that the parser constraint will always hold (i.e., the parser error will never occur).

We can see that the extracted code constraint holds in the model. However, the opposite is not true. Although the parser constraint allows for both ADDUSER and LONG_OPTS to be simultaneously selected with ADDUSER_LONG_OPTIONS, it does not enforce it to *always* hold.

B.3 Discussion

The above examples explain why the extracted code constraints may be accurate, but not help in recovering many existing variability-model constraints from our subject systems. They help us understand the problems and limitations of using propositional logic for comparing two formulas or two sets of Boolean constraints.

The first example in Section B.1 illustrates two of these problems. First, the variability model has full knowledge of all configuration features. On the other hand, the extracted code constraints only involve a subset of these features. This can be because the features are not used in the implementation to begin with (e.g., configurator-related) or because they are simply not involved in preventing any build-time error. One way around this is to remove all variability-model constraints containing features not used in the implementation from the comparison. We tried this and found that our total recoverability would improve from % to 34 %. However, one of our goals is to understand how much of a variability model can be automatically reverse engineered which is why we use all the variability-model constraints. Keeping track of these constraints is still important to understand all types of constraints, but removing them from the comparison may allow more practical results. The second problem portrayed here is which set of constraints to use for comparison. If we had also used the edges of an implication graph for comparison, we would have additionally been checking if the derived variability-model constraint LAST_SMALL → UTMP, shown in Equation 6, holds in the code. In this case, we would have been comparing four variability-model constraints instead of three and would have been able to recover one of them. The second example in Section B.2 shows that disjunctions can skew our interpretation of a comparison.

Unfortunately, though, propositional logic seems to be the best available alternative for comparison. We considered other possibilities such as model counting (counting the number of configurations allowed by the variability model and those allowed by the extracted code formula). However, such numbers tell us little about the extracted dependencies and can be very misleading. Another option would be to construct feature models from the extracted code constraints and the existing variability-model constraints, and then to compare both feature models. However, constructing a variability model from constraints is already a challenging task [**?**], which may also require additional developer input.

While finding the proper comparison mechanism seems to still be an open problem, we believe that the comparison technique we use allows us to understand configuration constraints better, despite its drawbacks. Our main reason for the choice is that we can currently manually verify, track, and understand the logic behind the variability-model constraints, which also allows us to ask developers about these constraints.

References

[1] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse engineering feature models. In *Proceedings of the International Conference Software Engineering (ICSE)*, pages 461–470. ACM Press, 2011.