Mining Configuration Constraints: Static Analyses and Empirical Results

Sarah Nadi
University of Waterloo, Canada

Thorsten Berger
IT University of Copenhagen, Denmark

Christian Kästner
Carnegie Mellon University, USA

Krzysztof Czarnecki
University of Waterloo, Canada

ABSTRACT

Highly-configurable systems allow users to tailor the software to their specific needs. Not all combinations of configuration options are valid though, and constraints arise for technical or non-technical reasons. Explicitly describing these constraints in a variability model allows reasoning about the supported configurations. To automate creating variability models, we need to identify the origin of such configuration constraints. We propose an approach which uses build-time errors and a novel feature-effect heuristic to automatically extract configuration constraints from C code. We conduct an empirical study on four highly-configurable open-source systems with existing variability models having three objectives in mind: evaluate the accuracy of our approach, determine the recoverability of existing variability-model constraints using our analysis, and classify the sources of variability-model constraints. We find that both our extraction heuristics are highly accurate (93 % and 77 % respectively), and that we can recover 19 % of the existing variability-models using our approach. However, we find that many of the remaining constraints require expert knowledge or more expensive analyses. We argue that our approach, tooling, and experimental results support researchers and practitioners working on variability model re-engineering, evolution, and consistency-checking techniques.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, Measurement, Experimentation

Keywords

Variability models, feature models, software product lines, reverse engineering, static analysis, empirical software engineering

1. INTRODUCTION

Developing highly configurable software that can be tailored to specific needs has been receiving increasing attention by practitioners and researchers. Configuration options, or *features*, allow customizing functionality to user needs. For example, providing options to reduce energy consumption and memory footprint when building for embedded systems. Features can range from those tweaking small functional- and non-functional aspects, to those enabling whole subsystems of the software. Large configurable systems can easily have thousands of features with complex *constraints* that restrict valid combinations and values. Examples of such systems range from large industrial software product lines to prominent open-source systems software, such as the Linux kernel with currently more than 11,000 features [14, 41, 44].

Such configurable systems are usually divided into a *problem space* and a *solution space* [18] as shown in Figure 1 (explained shortly). The problem space describes the supported features and their dependencies, while the solution space is the technical realization of the system and the functionalities specified by the features (i.e., 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 problem space. Automated and interactive configurators use such models to support users in navigating a complex configuration space [9, 21, 53, 54]. However, many systems have no documented variability model or rely on informal textual descriptions of constraints (e.g., the FreeBSD kernel [42]). As the

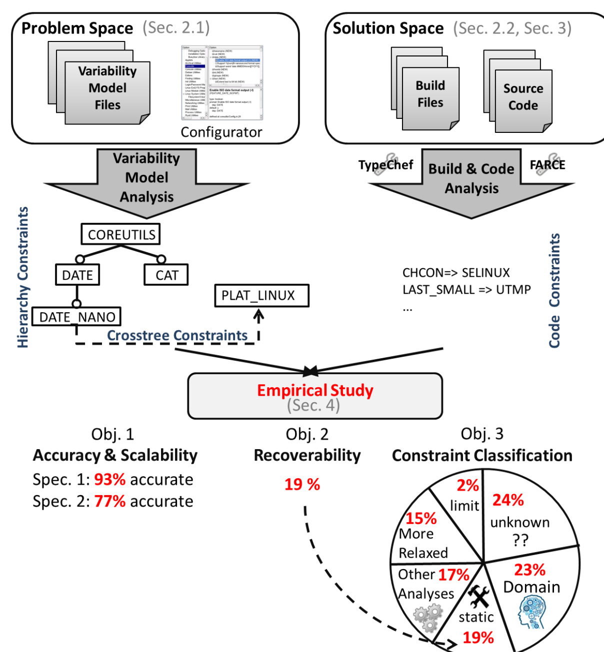

Figure 1: Overview of proposed approach and empirical study

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE '14, May 31 – June 7, 2014, Hyderabad, India
Copyright 14 ACM 978-1-4503-2756-5/14/05 ...$15.00.

number of features and their dependencies increases, configuration becomes more challenging [23,42], 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.

Identifying the sources of configuration constraints is essential to support automatically creating variability models. We expect a broad spectrum of constraints in a variability model ranging from purely low-level technical constraints, which reflect code dependencies (e.g., *multi-threaded I/O locking* depends on *threading* support in an operating system kernel), to purely non-technical constraints, which reflect domain-specific knowledge (e.g., marketing requirements placed by a project manager, or a sales department). The former can be discovered by just analyzing the code, while the latter can be found through talking to experts or looking at requirements documents, for example. However, additional sources of constraints may lie between these two ends. For example, there may be technical constraints not discoverable except through specific tests on particular platforms. We are not aware of any study that empirically investigates how prevalent different sources of constraints are in existing variability models. Such knowledge provides valuable insights into the practicability of automatically constructing a variability model.

In this work, we investigate the different sources of configuration constraints and to what extent we can *automatically* and *accurately* extract constraints from existing implementations using static analysis techniques. Figure 1 shows an overview of the approach we follow as well as our empirical evaluation. Our work has both a significant engineering contribution (*extracting constraints* from C code) and an empirical contribution (assessing *recoverability* and *classifying constraints* in existing variability models).

Extracting Constraints. Our work focuses on C based systems with build-time variability using the build system and C preprocessor. Since many features are directly used in implementation files [33], we assume that many of the configuration constraints are reflected in the code. We design and implement a *scalable* approach to extract constraints statically. We use two specifications: all valid configurations should build correctly, and they should all yield different products. For both specifications, we propose novel scalable extraction strategies based on the structural use of #IFDEF directives, on parser and type errors, and on linker checks. Whereas prior work *approximated* constraints from preprocessor directives [28,42,45,48,55], we design an infrastructure that *accurately* represents C code based on our previous research on variability-aware parsing and type checking [26,27,30]. In a nutshell, we statically analyze build-time variability effectively without examining an exponential number of configurations. We demonstrate scalability by extracting constraints from four large open-source systems (uClibc, BusyBox, eCos, and the Linux kernel) and evaluate accuracy by comparing the constraints to existing developer-created models. Our results show that our extraction is 93% and 77% accurate respectively for the two specifications we use, and can scale to the size of the Linux kernel in which we extract over 250,000 unique constraints.

Assessing Recoverability. We use our described infrastructure to automatically measure how many of the constraints in the existing variability models correspond to technical statically-discoverable code dependencies. Our results show that on average, 19 % of variability-model constraints reflect technical dependencies statically recoverable from code with our techniques. While around 3% prevent build-time errors, 15% of these model constraints correspond to simple nesting relationships in the code.

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

Contributions and Perspectives. To summarize, our contributions are: *(i)* an extension and combination of existing analyses (e.g., linker analysis and type checking) to extract configuration constraints, *(ii)* a novel constraint extraction technique based on feature use and code structure, *(iii)* a quantitative study of the effectiveness of such techniques to recover constraints, and *(iv)* a qualitative study of sources of constraints in existing models.

Our results can be used in various ways. For re-engineering approaches, our analyses extract constraints that can be used to (re-)construct variability models. For the evolution of systems, our techniques provide the basis for detecting inconsistencies and proposing fixes. Our empirical data, in particular identifying which types of code analysis recover most variability-model constraints, can help to design effective and optimized analysis techniques. Finally, information about which constraints appear in the code, and where they stem from (e.g., preventing a type error) may be useful for developers in understanding intricate dependencies when configuring these systems [10,23].

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 much 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 [42], 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 specifications of cross-tree constraints [13]. Specifically, the feature hierarchy is one of the major benefits of a variability model [42], 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

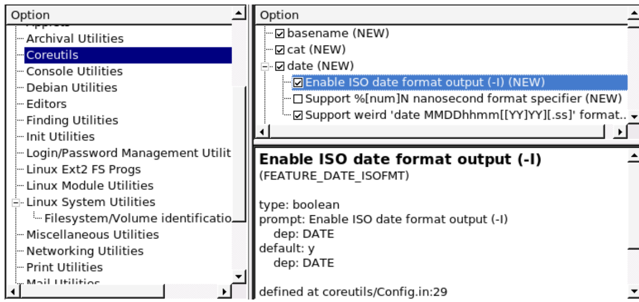

Figure 2: Configurator of the BusyBox system

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

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 would just 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.

There has been much research to extract constraints from existing variability models within the problem space [11, 40, 48]. Such extractors can interpret the semantics of different variability modeling languages to extract both hierarchy and cross-tree constraints, as shown in Figure 1.

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 [48]). We assume that there is a solution-space (code-level) constraint if any configuration violating this constraint is ill-defined by some specification. 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 justification behind these two specifications.

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 specification:

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

(a) type error

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

(b) feature effect

Listing 1: Examples of constraint sources

Specification 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 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 or changes some code functionality 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 specification as a simplified, conservative approximation of our second source of constraints:

Specification 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, $W \rightarrow Z \wedge X$ is a feature-effect constraint that should likely be in the model, even though violating it will not break the compilation.

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,

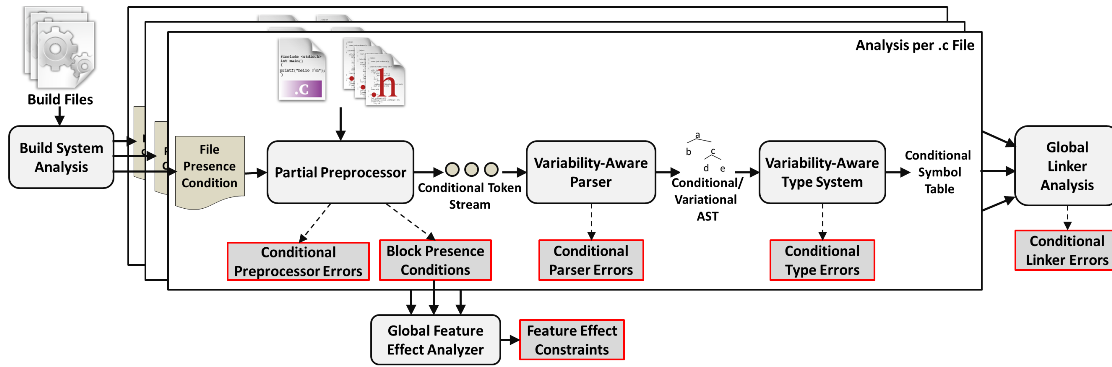

Figure 3: Variability-aware approach to extract configuration constraints from code

it is not clear if other sources of constraints exist beyond implementation artifacts and how prevalent they are. We, therefore, also strive to identify the sources of any non-recovered constraints.

Improving empirical understanding of constraints in real systems is crucial, especially since several studies emphasize configuration and implementation challenges for developers and users due to complex constraints [10,14,23,31]. Such knowledge not only allows us to understand which parts of a variability model can be reverse engineered and consistency-checked from code, and to what extent; but also how much manual effort, such as interviewing developers or domain experts, would be necessary to achieve a full model. For example, a main challenge when reverse-engineering a variability model from constraints is to disambiguate the hierarchy [42]. 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 specifications 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 specifications (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. EXTRACTING CODE CONSTRAINTS

One of our main goals is to extract configuration constraints from the solution space in order to compare them to the variability-model constraints (cf. Figure 1). To do so, we use the two specifications described in Section 2 to extract code constraints from *preprocessor errors*, *parser errors*, *type errors*, *linker errors*, and *feature effect*. Figure 3 shows an overview of the approach we use and which we explain in details in this section.

As shown in Figure 3, before analyzing the code in a specific C file, we first need to know under which condition the build system includes this file to be able to accurately derive constraints. 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 PC $\text{HUSH} \vee \text{ASH}$ is compiled and linked iff the features HUSH or ASH are selected.

To avoid an intractable brute-force approach of analyzing every possible configuration, 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 [25–27]. 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. As shown in Figure 3, partial preprocessing produces a token stream in which each token is guarded by a corresponding PC (including the file PC), which is subsequently parsed into a conditional abstract syntax tree, which can be subsequently type checked. This variability-aware analysis is conceptually sound and complete with regard to a brute-force approach of analyzing all configurations separately. However, it is much faster since it does the analysis in a single step and exploits similarities among implementations of different configurations; see [25–27] for more details.

In previous research with TypeChef, it was typically called with a given variability model such that it only emits error messages for parser or type problems that can occur in valid configurations—discarding all implementation problems that are already excluded by the variability model. This is the classic approach to find consistency errors, which a user can subsequently fix in the implementation or in the variability model [19, 49, 50]. Since we need to extract all constraints without knowledge of valid configurations, we use TypeChef in a different context where we run it *without* a variability model, and process all reported problems in all configurations.

We extend and instrument TypeChef, and implement a new framework FARCE (FeAtuRe Constraint Extractor) [2], which analyzes the output of TypeChef and the structure of the codebase with respect to preprocessor directive nesting, derives constraints according to our low-level specifications, and provides an infrastructure to compare extracted constraints between a variability model and code. We now explain our design decisions and methodology using the C code in Listing 2, adapted from BusyBox, as a running example.

3.1 Preprocessor, Parser, and Type Constraints

Preprocessor errors, parser errors, and type errors are detected at different stages of analyzing a file. However, the post-processing used to extract constraints from them is similar; thus, we discuss them together. In contrast, linker errors require a global analysis over multiple files, 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 the partial preprocessor to log #ERROR directives with their corresponding presence condition and to continue with the rest of the file instead of stopping on the #ERROR message. In Listing 2, Line 3 shows a #ERROR directive that occurs under the condition $\text{ASH} \wedge \text{NOMMU}$.

Parser Errors. Similarly, a normal C parser stops on syntax errors, such as mismatched parentheses. Our TypeChef parser reports

```
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)

an error message together with a corresponding presence condition, but continues parsing for other configurations. In Listing 2, a parser error occurs on Line 12 because of a missing semicolon if MAX_LEN is not selected. In this case, our analysis reports a parser error under condition $\text{ASH} \wedge \text{EDITING} \wedge \neg\text{MAX_LEN}$.

Type Errors. Where a normal type checker reports type errors in a single configuration, TypeChef's variability-aware type checker [25, 27] reports each type error together with a corresponding PC. In Listing 2, we detect a type error in Line 23 if EDITING is not selected since `line_input_state` is only defined under condition $\text{ASH} \wedge \text{EDITING}$ on Line 7. TypeChef would, thus, report a type error under condition $\text{ASH} \wedge \text{EDITING_VI} \wedge \text{MAX_LEN} \wedge \neg\text{EDITING}$.

Constraints. Following *Specification 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 ϕ of an error, we extract a configuration constraint $\neg\phi$. In our running example, we extract the following constraints (rewritten to equivalent implications): $\text{ASH} \rightarrow \neg\text{NOMMU}$ from the preprocessor, $\text{ASH} \rightarrow (\text{EDITING} \rightarrow \text{MAX_LEN})$ from the parser, and $\text{ASH} \rightarrow ((\text{EDITING_VI} \wedge \text{MAX_LEN}) \rightarrow \text{EDITING})$ from the type system.

3.2 Linker Constraints

To detect linker errors in configurable systems, we build a conditional symbol table for each file during type checking. The symbol table describes all non-static functions as exported symbols and all called but not defined functions as imports. All imports and exports are again guarded by corresponding PCs. We check only linkage within the application and discard all symbols defined in libraries (with additional analysis though, we could also model library symbols with 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 PC INIT in some other file (not shown). For more details on conditional symbol tables, see [6, 27].

Table 1: Example of two conditional symbol tables

file	symbol	kind	presence condition
Listing 2	`init`	export	$\text{ASH} \wedge \text{EDITING}$
	`main`	export	ASH
	`initEditing`	import	$\text{ASH} \wedge \text{EDITING}$
other file	`initEditing`	export	INIT

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

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

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

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 file f in which s is imported/exported and PC ψ. The *def/use* constraints ensure that the PC of an import implies at least one PC of a corresponding export, while the *conflict* constraints ensure mutual exclusion of the PCs of exports with the same function name. An overall linker constraint can be derived by conjoining all *def/use* and *conflict* constraints for each symbol in the set of all symbols S: $\bigwedge_{s\in S} \mathit{def/use}(s) \wedge \mathit{conflict}(s)$. If the two files shown in Table 1 were the only files in the system, we would extract the constraint $\text{ASH} \wedge \text{EDITING} \rightarrow \text{INIT}$ for symbol `initEditing`.

3.3 Feature Effect

To ensure Specification 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 occurs only 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: $\text{EDITING} \rightarrow \text{ASH}$ 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 develop the following novel and principled approach, deriving a constraint for each feature's effect from PCs throughout the system.

First, we collect all unique PCs of all code fragments occurring in the entire system (in all files, including the corresponding file PC as usual). Technically, we inspect the conditional token stream produced by TypeChef's partial preprocessor and collect all *unique* PCs (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 PCs 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 an no effect in a PC $\phi \in P$ if $\phi[f \leftarrow \mathit{True}]$ is equivalent to $\phi[f \leftarrow \mathit{False}]$,

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 PC, 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 PC ϕ. In all configurations in which the result of substituting f is different (using xor: $\phi[f \leftarrow True] \oplus \phi[f \leftarrow False]$). This method is also known as unique existential quantification.

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

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

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

In our running example, we can identify five unique PCs (excluding tokens for spaces and line breaks): ASH, ASH $\wedge$ NOMMU, ASH $\wedge$ EDITING, ASH $\wedge$ EDITING $\wedge$ MAX_LEN, and ASH $\wedge$ EDITING_VI $\wedge$ 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 iff 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 constraints the build system alone provides, we do the same analysis for file PCs instead of PCs of code blocks. Note that the feature effect analysis on the build system alone is incomplete and provides only a rough approximation.

4. EMPIRICAL STUDY

We now study four real-world systems with existing variability models. As shown in Figure 1, our objectives are: **O1** to evaluate *accuracy* and *scalability* of our extraction approach. This is done by checking if the configuration constraints we extract from implementation are enforced in existing variability models. **O2** 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. **O3** to *classify* variability-model constraints. In other words, 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. For all three objectives, we report the key results in this paper. Refer to our online appendix for full datasets, additional statistics, and detailed qualitative results [5].

4.1 Study Setup

4.1.1 Subject Systems

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

Our subjects comprise the following systems and variability model sizes. The first three use the Kconfig [56], and the last one uses the CDL [52] language and configurator infrastructure in the problem space. We choose systems with exsiting variability models to have a basis for comparison.

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

In all 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 [26, 27, 31, 36, 48]. In particular, eCos and Linux have two of the largest publicly available variability models today.

4.1.2 Methodology and Tool Infrastructure

We follow the methodology shown in Figure 1. We first extract hierarchy and cross-tree constraints from the variability models of our subject systems (problem space). We rely on our previous analysis infrastructures LVAT [4] 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 [11, 40] for details).

We then run TypeChef on each system, and use our developed infrastructure FARCE to derive solution-space constraints from its error output (*Specification 1*, cf. Section 2.2.1) and the conditional token stream (*Specification 2*, cf. Section 2.2.2). As a prerequisite, we extract file PCs from build systems by reusing our build-system analysis tool *KBuildMiner* [3] for systems using KBUILD (BusyBox and Linux), and a semi-manual approach for the others.

4.1.3 Evaluation Technique

After problem and solution-space constraints are extracted, we compare them according to our three objectives. To address **O1** (evaluate accuracy and scalability), we verify whether extracted solution-space constraints hold in the propositional formula representing the variability model (problem space) of each system. We also measure the execution time of the involved analysis steps. For this objective, we assume the existing variability model as the ground truth, since it reflects the system's configuration knowledge which developers have specified.

To address **O2** (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

Table 2: Constraints extracted with each specification per system, and percentage holding in the variability model (VM)

Code Analysis	uClibc		BusyBox		eCos		Linux	
	# extracted	% found in VM	# extracted	% found in VM	# extracted	% found in VM	# extracted	% found in VM
Specification 1								
Preprocessor Constr.	158	100	3	100	162	81	12,780	81
Parser Constr.	60	100	23	100	133	91	8,443	100
Type Checking Constr.	958	96	54	100	139	82	256,510	97
Linker Constr.	314	63	38	100	7	100	19,654	90
Total	1,340	90	118	100	441	85	284,914	96
Specification 2								
Feature effect Constr.	55	75	359	93	263	62	2,961	95
Feature effect - Build Constr.	25	80	62	0	n/a	n/a	2,552	97
Total	80	76	421	79	263	62	5,513	96

recoverability instead of *recall*, because we do not have a ground truth in terms of which constraints can be extracted from the code. Since no previous study has classified the kinds of constraints in variability models, we cannot expect that 100% of them are enforced in the code and can be automatically extracted. To address this gap and **O3** (classification of variability-model constraints), we show the types of constraints we could automatically recover, and manually investigate 144 randomly sampled non-recovered model constraints to characterize constraints that are not found by our analysis. Note that averages and numbers across subjects are geometric means.

4.2 O1: Accuracy and Scalability

We expect that all constraints extracted according to Specification 1 hold in the problem-space variability model, 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—cases we analyze separately. Such checks have been the standard approach in previous work on finding bugs in configurable systems [19, 26, 50], where inconsistencies between the model and implementation are identified as errors. In contrast, Specification 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.

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

We record execution time of each analysis step separately to measure the scalability of our approach. 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 can not be parallelized, we report the total computation time per system.

Results. 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 Specification 1 and Specification 2 are 93 % and 77 % accurate, respectively.

Both results show that we achieve a very high accuracy across all four systems. *Specification 1* is a reliable source of constraints where our tooling produces only few false positives (extracted constraints that do not hold in the model). Interestingly, a 77 % accuracy rate for Specification 2 suggests that variability models in fact prevent meaningless configurations to a high degree.

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

Table 3 shows execution times of our tools, which were executed on a server with two AMD Opteron processors (16 cores each) and 128GB RAM. Significant time is taken to parse files, which often explode after expanding all macros and #INCLUDE preprocessor directives. Our results show that our analysis scales reasonably where a system as large as Linux can be analyzed in parallel within twelve hours on our hardware.

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 though, 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 [27]. 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 PCs. In general, intricate details in Makefiles, such as shell calls [12], complicate their analysis [47]. 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 are currently investigating using symbolic

Table 4: Number (and percentage) of variability model hierarchy constraints recovered from each code analysis

	uClibc	BusyBox	eCos	Linux
# of VM Hierarchy Constraints	54	366	588	4,999
	Count (%) Recovered from code			
Specification 1				
Preprocessor Constr.	0 (0 %)	0 (0 %)	0 (0 %)	1 (0 %)
Parser Constr.	0 (0 %)	0 (0 %)	3 (0 %)	1 (0 %)
Type Checking Constr.	0 (0 %)	1 (0 %)	0 (0 %)	0 (0 %)
Linker Constr.	0 (0 %)	1 (0 %)	1 (0 %)	1 (0 %)
Total (Unique)	0 (0 %)	2 (1 %)	4 (1 %)	3 (0 %)
Specification 2				
Feature effect Constr.	8 (15 %)	251 (69 %)	60 (10 %)	325 (7 %)
Feature effect - Build Constr.	4 (7 %)	0 (0 %)	-	1,337 (27 %)
Total (Unique)	9 (17 %)	251 (69 %)	60 (10 %)	1,661 (33 %)
Total Unique Constraints Recovered	9 (17 %)	253 (69 %)	64 (11 %)	1,664 (33 %)

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			
Specification 1				
Preprocessor Constr.	2 (2 %)	1 (0 %)	5 (2 %)	6 (0 %)
Parser Constr.	0 (0 %)	0 (0 %)	9 (2 %)	2 (0 %)
Type Checking Constr.	8 (7 %)	15 (6 %)	1 (0 %)	3 (0 %)
Linker Constr.	12 (10 %)	21 (8 %)	1 (0 %)	19 (0 %)
Total (Unique)	16 (14 %)	37 (14 %)	15 (5 %)	28 (0 %)
Specification 2				
Feature effect Constr.	6 (5 %)	14 (5 %)	1 (0 %)	58 (1 %)
Feature effect - Build Constr.	3 (3 %)	0 (0 %)	-	316 (4 %)
Total (Unique)	7 (6 %)	14 (5 %)	1 (0 %)	374 (5 %)
Total Unique Constraints Recovered	22 (19%)	51 (19 %)	16 (5 %)	402 (5 %)

execution of build systems [47] in order to accurately identify which header files need to be included under different configurations.

Scalability Discussion. Our evaluation shows that our approach scales, in particular to systems sharing the size and complexity of the Linux kernel. However, we face many scalability issues when combining complex constraint expressions into one formula, mainly in Linux and eCos. Feature-effect constraints were particularly problematic due to the unique existential quantification (see Section 3.3), which causes an explosion in the number of disjunctions in many expressions, thus adding complexity to the SAT solver. To overcome this, we omit expressions including more than ten features when aggregating the feature effect formula. This resulted in using only 17 % and 51 % of the feature-effect constraints in Linux and eCos respectively. The threshold was chosen due to the intuition that larger constraints are too complex and likely not modeled by developers.

We faced similar problems in deriving other formulas, such as the type formula in Linux, but mainly due to the huge number of constraints and not their individual complexity. This required several workarounds and required high memory consumption in the conversion of the formula into conjunctive normal form, required by our SAT solver. Thus, we conclude that extracting constraints according to our specifications scales, but can require workarounds or filtering expressions to deal with the explosion of constraint formulas. Refer to our online appendix [5] for more details.

4.3 O2: Recoverability

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

Measurement Strategy. While the extraction approach directly gives us individual constraints to count and compare, the situation is more challenging when measuring constraints from the variability model. Variability models in practice use different specification languages. Semantics of a variability model are typically expressed uniformly as a single large Boolean function expressed as a propositional formula describing the valid configurations. After experimenting with several slicing techniques for comparing these propositional formulas, we decide to exploit structural characteristics of variability models that are commonly found. In all analyzed models, we can identify child-parent relationships (*hierarchy constraints*), as well as inter-feature constraints (*cross-tree constraints*). This way, we count individual constraints as the developer modeled them, which is intuitive to interpret, and allows us to investigate the different types of model constraints. Note that we 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 in Section 4.2: we compare whether each individual problem-space constraint ψ_c holds in the conjunction of all extracted solution-space constraints ϕ_i in each code analysis category, i.e., whether $(\bigwedge_i \phi_i) \Rightarrow \psi_c$ is a tautology.

Results. In Tables 4 and 5, we show how many of the variability models' hierarchy and cross-tree constraints can be recovered automatically from code. Since the same constraint can be recovered by different analyses, we also show the total number of unique constraints for each specification and for each system. Across the four systems, we recover 26 % of hierarchy constraints, and 10 % of cross-tree constraints.

To compare the two specifications 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 specifications 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 (Specification 1). Overall, we can recover 19 % of variability-model constraints using both specifications across the four systems.

Recoverability Discussion. We can see a pattern in terms of where variability-model hierarchy and cross-tree constraints are reflected in the code. As can be seen in Table 4, the structure of the variability model (hierarchy constraints) often mirrors the structure of the code. Specification 2 alone can extract an average 25 % of the hierarchy constraints. An interesting case is Linux where already 27 % of the hierarchy constraints are mirrored in the nested directory structure in the build system (i.e., file PCs). 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 [12, 33]. On the other hand, although harder to recover, cross-tree constraints seem to be scattered across different places in the code (e.g., linker and type information), and seem more related to preventing build errors than hierarchy constraints are. Interestingly, Figure 4 shows that there is no overlap (with the exception of one constraint in uClibc) between the two specifications 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 19 % of the variability-model constraints suggests that many of the remaining constraints require different types of analysis or stem from sources other than the implementation. We look at this in more details in our final objective.

4.4 O3: Classification of Variability Model Constraints

To investigate which parts of a variability model can be automatically extracted, our aim is to understand the kinds of constraints that

exist in variability models, and the analyses and knowledge needed to identify them.

Measurement Strategy. To automate parts of the investigation, we use the recoverability results from Section 4.3 to automatically classify a large number of constraints as technical and statically discoverable, which reduces manual investigation to the remaining ones. To manually investigate the remaining constraints, we randomly sample 144 non-recovered constraints (18 hierarchy and 18 cross-tree constraints from each subject systems). We then divide these constraints among the authors for manual investigation.

Results. From our manual investigation of 144 non-recovered constraints, we classify five cases in which constraints could not be statically detected from the code with our approach. In Figure 1, we summarize the overall classification of the sources of constraints including those automatically found through our static analysis.

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

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

Case 3. Domain Knowledge: For 40 (28 %), at least one of the features is not used in implementation. We find two cases where this occurs. The first is that the constraint is *configurator-related* where that feature is used only internally in the variability model to support its menu structure and constraint propagation in the configurator. For example, HAS_NETWORK_SUPPORT in uClibc is a *menuconfig* [41], which helps organizing networking features in the configurator into a menu format. This happens in 27 (19 %) constraints. From their domain knowledge, developers usually know which features are related and are, thus, grouped together in the same menu. For the remaining constraints, we find that this unused feature represents some form of platform or hardware knowledge. For example, in Linux, SERIO_CT82C710 $\rightarrow$ X86_64, where the first feature controls the port connection in that particular chip, but which seems to only work with an X86_64 architecture. Such hardware dependencies are not statically detectable in the code and can only be found through testing the software on the different platforms. We believe that developers use their domain expertise (usually gained from previous testing experiences) to enforce such dependencies.

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

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

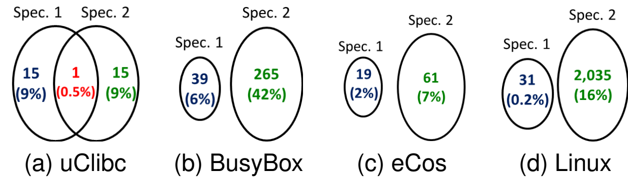

Figure 4: Overlap between Specifications 1 and 2 in recovering variability-model constraints. An overlap means that the same model constraint can be recovered by both specifications

Classification Discussion. Our classification shows that many (19 %) of the variability-model constraints can be statically extracted with our approach. This seems motivating for automated extraction tools. We have especially seen that 15 % of constraints are reflected in the nesting structure and can be easily extracted using *Specification 2*, since it only depends on extracting the file PCs and lexing the files, which are cheaper steps in the analysis (see Table 3). However, our manual analysis of the remaining constraints also shows that many of the constraints can only be found through more expensive analysis, such as testing. Additionally, it seems that several constraints in the model are non-technical and are simply responsible for organizing the structure of the model for configuration purposes. We have also come across constraints that could only stem from domain knowledge. Both these facts suggest that additional developer and expert input may always be needed to create a complete model.

Finally, the constraints we find in Case 2 of our manual analysis explain why an analysis may produce accurate constraints and yet recover no variability-model constraints. For example, the type analysis in Linux extracts over 0.25 million constraints which are 97% accurate (Table 2), and yet only recovers 3 cross-tree constraints in Table 5. We plan to investigate the feasibility of comparing non-binary constraints to overcome this.

5. THREATS TO VALIDITY

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

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. However, the benefit gained from performing such expensive analyses still needs investigation.

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

Construct validity. Different transformations or interpretations of the variability model may lead to different comparison results than the ones achieved (e.g., additionally looking at ternary relationships in the model). Properly comparing constraints is a difficult problem, and we believe the comparison methods we 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.

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

6. 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 [26, 27, 30]. 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. The work is complementary to our prior reverse-engineering approach for feature models [42] (an academic variability modeling notation [24]), 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.

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 [8, 15, 43, 46]. 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, 22]. In contrast to techniques using non-code and domain information, we extract *technical constraints* from code.

From a technical perspective, previous work has attempted to extract constraints from code with #IFDEF variability [28, 42, 48]. Most attempts focus on the preprocessor code exclusively [28, 48], 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) [35] or could only process specific usage patterns (incomplete) [7]. For instance, our previous work [42] used an inexact parser to approximate parts of our Specification 1 and 2. Our new infrastructure is sound and complete [26], 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 [38]. Reisner et al. use symbolic execution to identify interactions and constraints among configuration parameters by symbolically executing a system's test cases [39]. Such dynamic analysis can identify additional constraints as discussed in Section 4.4. However, scalability of symbolic execution is limited to medium size systems (up to 14K lines of code with up to 30 options in [39]), whereas our build-time analysis scales to systems as the Linux kernel. We also avoid using techniques such as data-flow analysis [16, 17, 30] 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 [16, 30, 34, 39] 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 [50]. However, most of this work has analyzed features in isolation, either in the problem space [14, 37, 41, 51] or in the solution space [29, 45] to identify modeling practices and feature usage. Some work has also looked at both sides to study co-evolution [31, 36] or to detect bugs due to inconsistencies between models and code [26, 27, 32, 48, 49]. 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.

7. CONCLUSIONS

We have engineered static analyses to extract configuration constraints and performed a large-scale study of constraints in four real-world systems. Our results raise four main conclusions.

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

8. ACKNOWLEDGMENTS

Partly supported by NSERC CGS-D2-425005, ARTEMIS JU grant n° 295397 VARIES, and NSF grant CCF-1318808.

9. REFERENCES

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

[2] FARCE. https://bitbucket.org/tberger/farce.

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

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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, 2013.

[17] 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.

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

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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`.

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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. accepted for publication Jan 30, 2014.

[51] 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.

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

[53] 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.

[54] 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.

[55] B. Zhang and M. Becker. Code-based variability model extraction for software product line improvement. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 91–98. ACM Press, 2012.

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