Feature Model Synthesis

by
Steven She

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2013

Author's Declaration

Abstract

Variability provides the ability to adapt and customize a software system's artifacts for a particular context or circumstance. Variability enables code reuse, but its mechanisms are often tangled within a software artifact or scattered over multiple artifacts. This makes the system harder to maintain for developers, and harder to understand for users that configure the software.

Feature models provide a centralized source for describing the variability in a software system. A feature model consists of a hierarchy of features—the common and variable system characteristics—with constraints between features. Constructing a feature model, however, is a arduous and time-consuming manual process.

We developed two techniques for feature model synthesis. The first, FEATURE-GRAPH-EXTRACTION, is an automated algorithm for extracting a feature graph from a propositional formula in either conjunctive normal form (CNF), or disjunctive normal form (DNF). A feature graph describes all feature diagrams that are complete with respect to the input. We evaluated our algorithms against related synthesis algorithms and found that our CNF variant was significantly faster than the previous comparable technique, and the DNF algorithm performed similarly to a comparable, but newer technique, with the exception of several models where our algorithm was faster.

The second, FEATURE-TREE-SYNTHESIS, is a semi-automated technique for building a feature model given a feature graph. This technique uses both logical constraints and text to address the most challenging part of feature model synthesis—constructing the feature hierarchy—by ranking potential parents of a feature with a textual similarity heuristic. We found that the procedure effectively reduced a modeler's choices from thousands, to five or less when synthesizing the Linux and eCos variability models.

Our third contribution is the analysis of Kconfig—a language similar to feature modeling used to specify the variability model of the Linux kernel. While large feature models are reportedly used in industry, these models have not been available to the research community for benchmarking feature model analysis and synthesis techniques. We compare Kconfig to feature modeling, reverse engineer formal semantics, and translate 12 open-source Kconfig models—including the Linux model with over 6000 features—to propositional logic.

Acknowledgements

I would like to thank Professors Krzysztof Czarnecki and Andrzej Wąsowski for providing the guidance and direction that has led to the completion of this dissertation.

Thank you to my colleagues in the Generative Software Development Lab and co-authors: Thorsten Berger, Rafael Lotufo, Yingfei Xiong, and Nele Andersen. A special thanks to Thorsten Berger for the many discussions that have contributed to the work in this thesis.

Finally, thank you to my friends and family—this work would not have been possible without their support.

Contents

List of Tables

List of Figures

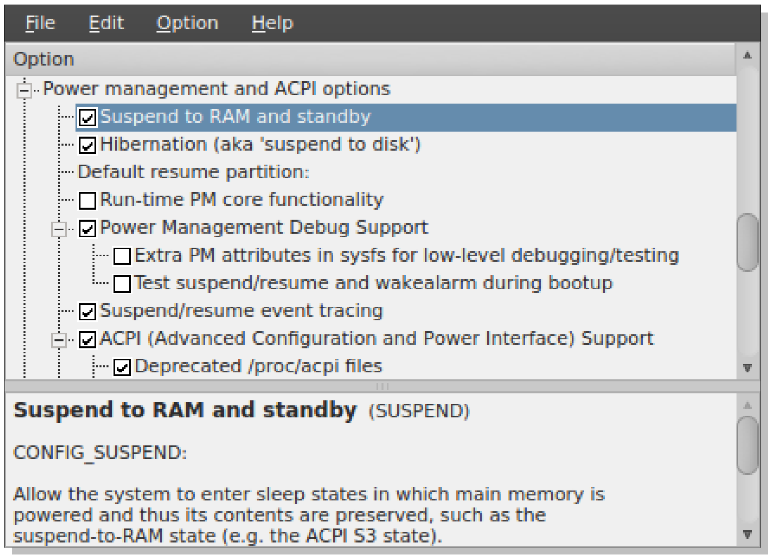
Chapter 1

Introduction

Variability provides the ability to adapt and customize a software system's artifacts for a particular context or circumstance [vGBS01]. Variability enables code reuse, but its mechanisms are often tangled within an artifact or scattered over multiple artifacts that span the problem space, solution space, or the mapping connecting the two spaces. *Variability-rich software*, in particular, pose a challenge for developers and users. Developers need to understand the impact of a change in order to add new features or dependencies to existing features. For users that configure the software system, complex feature interactions can lead to unsatisfied dependencies.

Feature models (FMs), first introduced by Kang et al. [KCH+90], describes *features*—the common or variable characteristics of the products in a SPL—as a visual hierarchy with additional constraints between features. Since feature models were first introduced, they have been used in a wide variety of tasks such as domain analysis [KCH+90], model management [Ach11], describing design or implementation constraints in variability-rich software [CE00], and product configuration. Feature model configurators provides intuitive graphical interfaces for configuring features and understanding their dependencies (Figure 1.1).

Figure 1.2 shows a feature model of a mobile phone product line. Features are represented as rectangles and may be *optional*—denoted by an empty circle at the top—or *mandatory*—denoted by a filled circle. An edge from one feature to another denotes a dependency, where a solid line denotes the feature hierarchy, and a dashed line with an arrow is a *cross-tree implies edge*. *Cross-tree excludes edges* can also exist in the diagram, but are not shown in this particular example. Constraints between sibling features can also be specified as part of a *feature group*. An XOR-group, shown with a clear arc, denotes that exactly one member of the group must be selected if its parent is selected. OR-groups, shown with a filled arc, where one or more members must be selected, and



Option

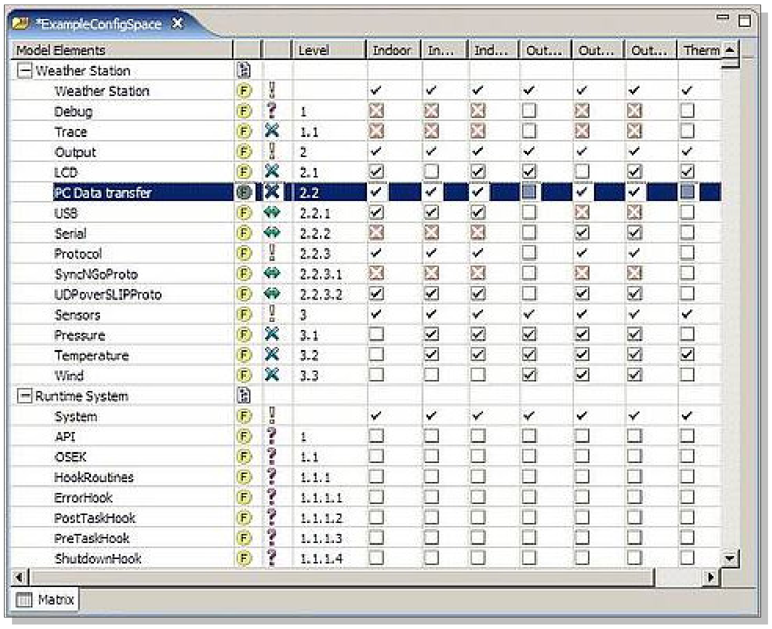
- Power management and ACPI options
  - ☑ Suspend to RAM and standby
  - ☑ Hibernation (aka 'suspend to disk')
  - Default resume partition:
  - ☐ Run-time PM core functionality
  - ☑ Power Management Debug Support
    - ☐ Extra PM attributes in sysfs for low-level debugging/testing
    - ☐ Test suspend/resume and wakealarm during bootup
  - ☑ Suspend/resume event tracing
  - ☑ ACPI (Advanced Configuration and Power Interface) Support
    - ☑ Deprecated /proc/acpi files

Suspend to RAM and standby (SUSPEND)

CONFIG_SUSPEND:

Allow the system to enter sleep states in which main memory is
powered and thus its contents are preserved, such as the
suspend-to-RAM state (e.g. the ACPI S3 state).

(a) A graphical configurator for the Linux kernel



Model Elements			Level	Indoor	In...	Ind...	Out...	Out...	Out...	Therm
Weather Station										
Weather Station	F	!		✓	✓	✓	✓	✓	✓	✓
Debug	F	?	1	☒	☒	☒	☐	☒	☒	☐
Trace	F	✗	1.1	☒	☒	☒	☐	☒	☒	☐
Output	F	!	2	✓	✓	✓	✓	✓	✓	✓
LCD	F	✗	2.1	☑	☐	☑	☑	☐	☑	☑
PC Data transfer	F	✗	2.2	✓	✓	✓	▨	✓	✓	▨
USB	F	↔	2.2.1	☑	☑	☑	☐	☒	☒	☐
Serial	F	↔	2.2.2	☒	☒	☒	☐	☑	☑	☐
Protocol	F	!	2.2.3	✓	✓	✓	☐	✓	✓	☐
SyncNGoProto	F	↔	2.2.3.1	☒	☒	☒	☐	☒	☒	☐
UDPoverSLIPProto	F	↔	2.2.3.2	☑	☑	☑	☐	☑	☑	☐
Sensors	F	!	3	✓	✓	✓	✓	✓	✓	✓
Pressure	F	✗	3.1	☐	☑	☑	☑	☑	☑	☐
Temperature	F	✗	3.2	☐	☑	☑	☑	☑	☑	☑
Wind	F	✗	3.3	☐	☐	☐	☑	☑	☑	☐
Runtime System										
System	F	!		✓	✓	✓	✓	✓	✓	✓
API	F	?	1	☐	☐	☐	☐	☐	☐	☐
OSEK	F	?	1.1	☐	☐	☐	☐	☐	☐	☐
HookRoutines	F	?	1.1.1	☐	☐	☐	☐	☐	☐	☐
ErrorHook	F	?	1.1.1.1	☐	☐	☐	☐	☐	☐	☐
PostTaskHook	F	?	1.1.1.2	☐	☐	☐	☐	☐	☐	☐
PreTaskHook	F	?	1.1.1.3	☐	☐	☐	☐	☐	☐	☐
ShutdownHook	F	?	1.1.1.4	☐	☐	☐	☐	☐	☐	☐

(b) Variant matrix editor from pure::variants [pur12]

Figure 1.1: Graphical configurators for feature models

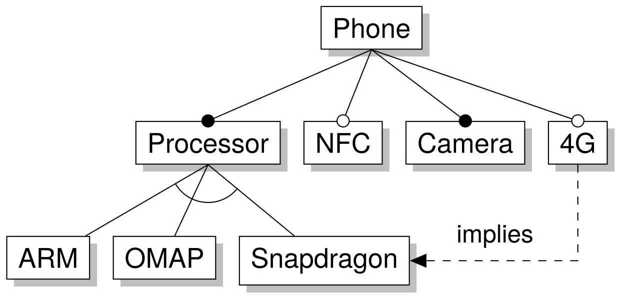


Figure 1.2: Feature model of a model phone product line

{ Phone, Processor, ARM, Camera },
{ Phone, Processor, ARM, Camera, NFC },
{ Phone, Processor, OMAP, Camera },
{ Phone, Processor, Snapdragon, Camera, 4G },
...

Figure 1.3: Subset of legal configurations of the mobile phone feature model

MUTEX-groups, where zero or one members must be selected, also exist, but are not shown in the figure. The graphical components—i.e., the feature hierarchy, feature groups, and cross-tree edges—form the *feature diagram*. A feature model consists of the feature diagram and an additional *cross-tree formula* to describe constraints that could not be represented in the diagram. The configuration semantics of a feature model is a set of *legal configurations* defined by the satisfying assignments for its translation to propositional logic [Bat05]. For example, a subset of the legal configurations for the feature model in Figure 1.2 is the set of configurations in Figure 1.3.

Since their introduction, feature models have become widely used in literature and in industry. At the time of writing, the SPLOT model repository contained over 250 feature models gathered from academia, or contributed by industry[1] [MBC09]. While these models originate from different sources, the models are all small in size, with the largest model having 280 features[2]. Large feature models with hundreds, and even thousands of features exist in industry. However, these model are not available to the research community [MBC09, BSRC10]. Open source, variability rich projects, such as the Linux or eCos kernels have large, explicitly defined variability models. Linux uses the Kconfig language to specify its variability model and eCos has the Component Definition Language (CDL). These language are quite similar to feature modeling and

[1]http://splot-research.org
[2]As of December 9, 2012

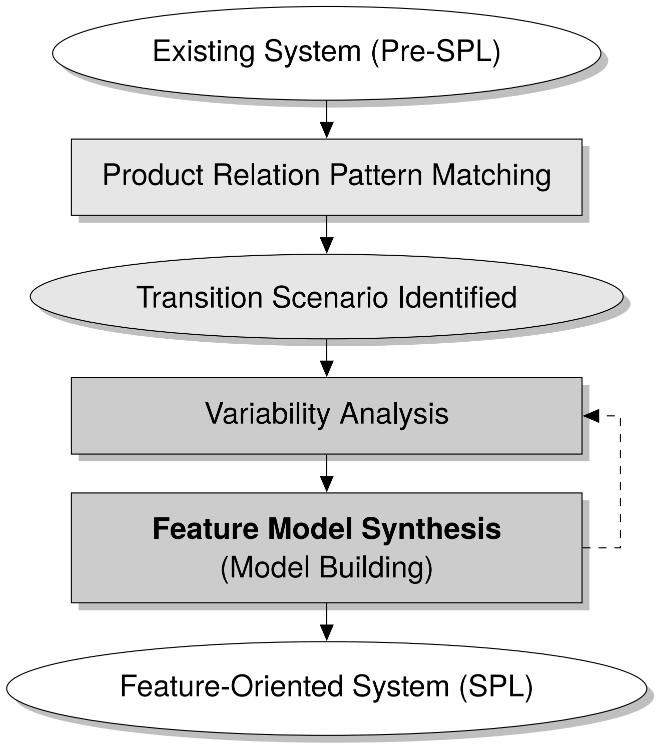

Existing System (Pre-SPL)
Product Relation Pattern Matching
Transition Scenario Identified
Variability Analysis
Feature Model Synthesis
(Model Building)
Feature-Oriented System (SPL)

Figure 1.4: Steps for software migration to SPLs (adapted from [Beu06])

their models can be interpreted as feature models [SSSPS07, SLB+10, BSL+10b]. We derived a mapping from Kconfig concepts to feature modeling and identified unique concepts of Kconfig [SLB+10, BSL+10b]. We reverse engineered formal semantics for the Kconfig language [SB10] and used the semantics to translate a Kconfig model to propositional logic. We describe our research on the Kconfig language in Chapter 4.

Figure 1.4 is a workflow by Beuche for migrating a software system to a SPL architecture [Beu06]. In the figure, *product relation pattern matching* is the process of identifying the relation between the products in the envisioned product line. This process involves identifying shared software assets, similarities between user-facing features, and whether the existing system used any form of systematic variability management between products. Next, a *transition scenario* that describes the goal of the migration process is identified. For example, one scenario involves merging separate products into a product line. Another scenario involves improving reuse between software assets in an existing product line. Chapter 3 describes several feature model synthesis scenarios that we gathered from literature and from industry experience reports [SCW12]. *Variability analysis* identifies features and variability in the project

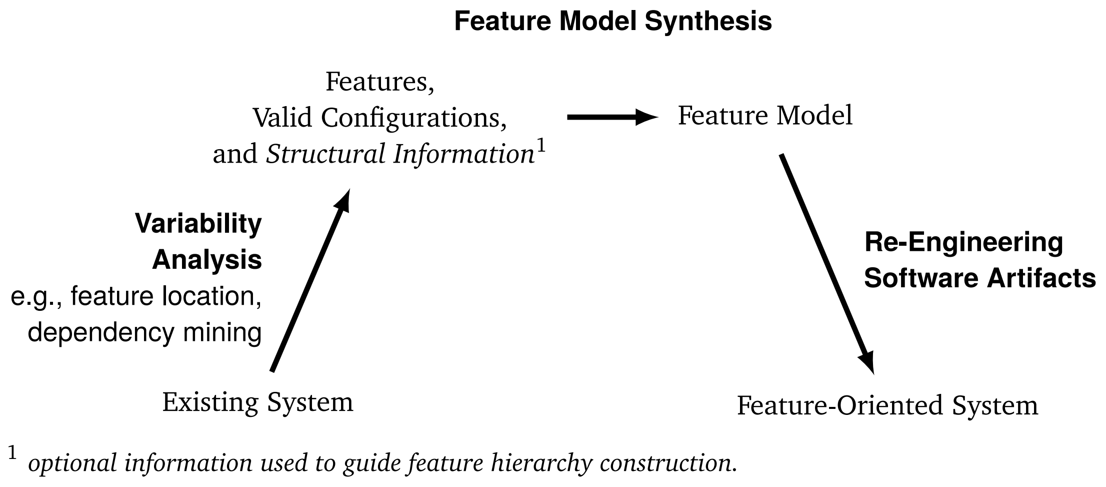

Feature Model Synthesis
Features,
Valid Configurations,
and *Structural Information*[1]
Feature Model
Variability Analysis
e.g., feature location, dependency mining
Re-Engineering Software Artifacts
Existing System
Feature-Oriented System
[1] *optional information used to guide feature hierarchy construction.*

Figure 1.5: Abstract steps in a feature-oriented re-engineering of a software system

artifacts. The last step of the process is *feature model synthesis*, the focus of this thesis. Feature model synthesis involves the construction and design of a feature model using the features and variability data identified in the previous steps of the workflow. In Beuche's original slides [Beu06], he describe the variability analysis and model building stages as iterative and incremental. In this dissertation, we address a one-way, batch process for feature model synthesis. We leave the incremental synthesis process as future work.

We elaborate on the variability analysis and feature model synthesis stages in Figure 1.5. Variability analysis is performed by a domain export to identify features and variability in a system with automated tooling such as feature identification and location [DRGP11]. Valid configurations can be identified with either dependency mining [BSL$^+$10a, Käs10], or extracting configurations from variants [RPK11]. The analysis stage also involves identifying structural information for the synthesis technique if needed. In our synthesis technique, we use supplemental feature descriptions to rank potential parents of a features (Chapter 6). We give other examples of analysis techniques for extracting the needed synthesis input in Chapter 3. In the feature model synthesis step, a domain expert constructs and designs a feature model given the identified features, valid configurations, and any supplemental information. This step is tool-assisted, since it requires user input to build and design a model. After a feature model is constructed, the remaining software artifacts in the project, such as the build system or source code, is adapted to use the synthesized feature model as its source for variability. In this thesis, we address the middle step in this process: feature model synthesis.

*STACK enables the stack(9) facility... **stack(9) will also be compiled in automatically if DDB(4) is compiled into the kernel**.*

(a) Documentation with a dependency between STACK and DDB

```
#ifdef DDB
#ifndef KDB
#error KDB must be enabled for DDB to work!
#endif
#endif
```

(b) Code snippet with a dependency between DDB and KDB

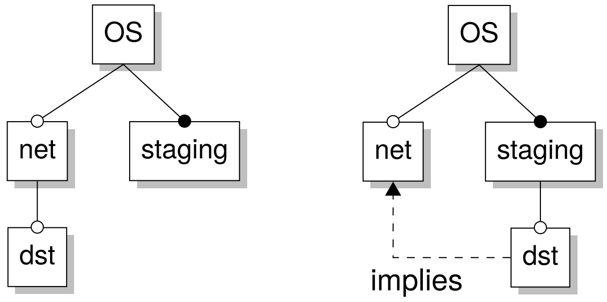
```
# Debugging for use in -current
options  KDB # Enable kernel debugger support.
options  DDB # Support DDB.
options  GDB # Support remote GDB.
...

nooption NATIVE
options  MCLSHIFT=12
```

(c) Snippet of a hardware configuration for i386 XEN support

Figure 1.6: Snippets of a FreeBSD variable artifacts

{ OS, staging }
{ OS, staging, net }
{ OS, staging, net, dst }

(a) A set of configurations



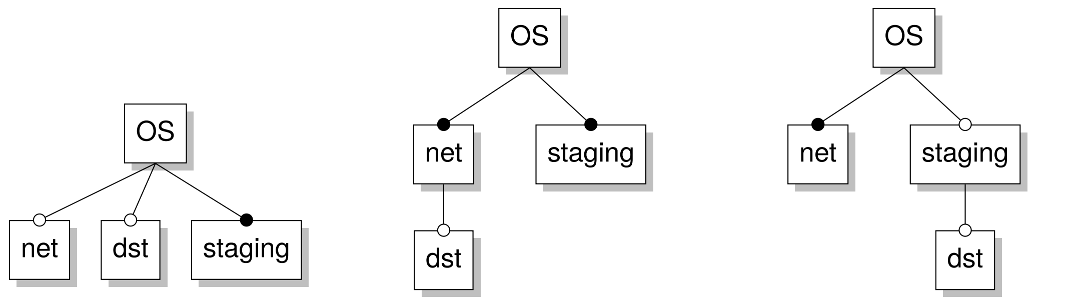
(b) A FM (c) Another FM

Figure 1.7: Two feature models with the same set of legal configurations [SLB+11]

Next, we will describe a concrete scenario involving feature model synthesis. The FreeBSD kernel is a system without a feature model and could benefit from a feature-oriented re-engineering. Variability is scattered over a mixture of variable artifacts and instances. Features and dependencies are described in an ad-hoc manner—features are scattered in documentation and dependencies are hidden in code and feature configurations for different hardware architectures and devices. Figure 1.6a is a snippet from documentation in FreeBSD describing a dependency between two features: STACK and DDB. Dependencies involving the DDB feature is not localized to just documentation; a dependency between DDB and KDB also exists in the FreeBSD source code (Figure 1.6b). A configuration in FreeBSD is simply a list of features and their values (Figure 1.6c). A user looking to understand the dependencies of the DDB feature would have to examine configuration templates, feature documentation, and source code. FreeBSD is a system that would benefit from an explicit representation of features and their dependencies.

Re-engineering the variability of a software system is just one scenario involving feature model synthesis. Feature models can also be synthesized for domain analysis from requirements, drive configuration of existing and future products from code variants, or as part of a model management operation such as a model merge from a set of other feature models [SCW12]. We elaborate on these scenarios in Chapter 3.

Feature models provide an intuitive graphical notation for describing a set of legal configurations. However, given a set of configurations, there could be many different feature models that describe these configurations. For example, Figures 1.7b and 1.7c both describe the same set of configurations in Figure 1.7a. Other models could describe more configurations (Figure 1.8a), or less configurations (Figure 1.8b). Even though the feature model in Figure 1.8a describes more configurations than the input,



(a) More configurations (b) Less configurations (c) Arbitrary configurations

Figure 1.8: Feature models that describe different configurations from Figure 1.7

this model could very well describe best what the modeler intended. For example, synthesis scenarios involving domain analysis from requirements, or synthesizing a feature model from individual variants to describe a product line require a feature model that is more general than the input configurations. Our feature model synthesis algorithms ensure that the synthesized feature model that is weaker than the input configurations. In other words, the configurations in the synthesized feature model are a superset of the configurations in the input, making the model complete with respect to the input configurations. Additional constraints can be added in the stage following feature model synthesis.

We developed an efficient, automated algorithm called *Feature Graph Extraction* (FGE) that recovers a feature model that is complete with respect to the input configurations. FGE takes an input propositional formula, and extracts a special *feature graph* that describes all possible feature diagrams that are *complete* with respect to the input configurations, or *sound* with respect to the input dependencies [ACSW12, CW07]. Our algorithm is capable of synthesizing a feature graph from a set of features, a set of dependencies expressed as a formula in conjunctive normal form (CNF), or as a set of configurations expressed as a formula in disjunctive normal form (DNF). FGE acts as an intermediate step in a larger feature model synthesis scenario. For example, the resulting feature graph can be used as input to an interactive feature model building tool [JKW08] or as part of a feature model merge or projection operation [Ach11]. We use FGE as the basis of our own semi-automated feature model synthesis technique [SLB^{+}11]. We describe FGE in Chapter 5.

The feature graph recovered by FGE represents *a set* of feature diagrams and is not a valid feature diagram on its own. A distinct feature tree and a set of feature groups have to be selected from the feature graph for it be a valid feature model. If we look at the two feature models in Figures 1.7b and 1.7c again, both feature models describe the same set of input configurations. However, the two feature models differ in their feature hierarchies. The feature hierarchy and arrangement of feature groups reflect the domain semantics of the feature model [SLB+11]. As an example, let's assume that selecting the net feature enables networking support, dst enables distributed storage support, and that subfeatures of staging describes experimental features. If dst is an experimental feature, placing the feature under staging is more appropriate (Figure 1.7c). However, if dst is a stable feature, then placing it under net is more appropriate (Figure 1.7b). In this example, the appropriate feature hierarchy is determined by the meaning, or domain semantics of its features.

Determining the location of a feature in the feature hierarchy becomes a significant challenge when synthesizing a feature model with thousands of features. We developed a semi-automated technique to identify relevant parents for a feature by using logical dependencies and a textual similarity heuristic [SLB+11]. We use this technique to reverse engineer a feature model for a portion of the FreeBSD kernel and evaluated the effectiveness of our technique by comparing the results of our heuristic with reference feature models extracted from the Linux, eCos, and a portion of the FreeBSD kernel. We discuss this technique in Chapter 6.

Our FM synthesis techniques assumes features, and valid feature combinations expressed as a set of dependencies or configurations as input. Other FM synthesis techniques exist that use different inputs, from requirements documents [ASB+08, NE08b] to semi-structured product descriptions [ACP+12]. We discuss related FM synthesis techniques in Chapter 7.

1.1 Thesis Statement

We synthesize large-scale feature models with thousands of features by using SAT-based reasoning on propositional formulas and suggest a feature hierarchy by combining logical reasoning with textual similarity heuristics.

We establish this thesis by evaluating our synthesis algorithms against comparable algorithms on input derived from generated and real-world feature models. We evaluate our textual similarity heuristic on input derived from the Linux, eCos, and FreeBSD kernels.

1.2 Contributions

This thesis claims the following contributions:

- We collected feature model synthesis scenarios from literature and industry experience reports. We classify and show the workflows of the individual scenarios and derive an abstract workflow and requirements for feature model synthesis algorithms [SCW12]. Chapter 3 presents the scenarios and requirements for feature model synthesis algorithms.
- We analyzed the Kconfig variability modeling language and derived formal semantics [SB10], and a mapping from Kconfig to feature modeling [SLB+10, BSL+10b]. The Kconfig language developed specifically for Linux to specify its variability model. This model has over 6000 features and provides the largest benchmark for feature model analysis and synthesis tools. The Kconfig analysis tools contributed to the analysis of the evolution of Linux over 21 releases from v2.6.12 to v2.6.32 [LSB+10], and extracted models from 11 open-source projects that use Kconfig [BSL+12]. We discuss the Kconfig language in Chapter 4.
- We introduce the automated FEATURE-GRAPH-EXTRACTION algorithm [SRA+13, ACSW12] in Chapter 5. Given input as a set of features, and a propositional formula in either conjunctive normal form (CNF) or disjunctive normal form (DNF), this algorithm recovers a graph that describes all feature diagrams that entails the input. Our evaluation found that the CNF variant perform 10 to 1000 faster than a previous BDD-based algorithm [CW07] and could handle much larger input, including input derived from the Linux kernel. The DNF variant was comparable to a formal concept analysis-based algorithm by Ryssel et al. [RPK11].
- In Chapter 6, we describe our semi-automated FEATURE-TREE-SYNTHESIS algorithm [SLB+11]. This algorithm takes a set of features, dependencies, and feature descriptions to present potential parents for feature to help a modeler build the feature hierarchy. Given a feature, FEATURE-TREE-SYNTHESIS creates a list containing the implied features ranked by their textual similarity, and a second list containing all features ranked by their textual similarity for situations where input dependencies may be missing. This algorithm was the first to use both logical dependencies and textual similarity heuristics for feature model synthesis. Our evaluation found that given a feature, the algorithm identified the correct parent for 76% of features in the Linux variability model and 79% of features

from the eCos variability models, and the correct parent appeared in the top 3% to 6% of all features.

1.3 Publications

[SLB$^+$10] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki, “The variability model of the linux kernel,” in VaMoS, 2010.

[SB10] S. She and T. Berger, “Formal semantics of the Kconfig language,” Generative Software Development Lab, University of Waterloo, Technical Note, 2010. [Online]. Available: http://gsd.uwaterloo.ca/sites/default/files/kconfig_semantics.pdf.

[BSL$^+$10a] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wąsowski, “Feature-to-code mapping in two large product lines,” in Software Product Lines: Going Beyond, SPLC, 2010.

[BSL$^+$10b] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki, “Variability modeling in the real: a perspective from the operating systems domain,” in ASE, 2010.

[SLB$^+$11] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki, “Reverse engineering feature models,” in ICSE, 2011.

[BSL$^+$12] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki, “Variability modeling in the systems software domain,” Generative Software Development Lab, University of Waterloo, Tech. Rep. GSDLAB-TR 2012-07-06, 2012. [Online]. Available: http://gsd.uwaterloo.ca/sites/default/files/vm-2012-berger.pdf.

[ACSW12] N. Andersen, K. Czarnecki, S. She, and A. Wąsowski, “Efficient synthesis of feature models,” in SPLC, 2012.

[XHSC12] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki, “Generating range fixes for software configuration,” in ICSE 2012.

[SCW12] S. She, K. Czarnecki, and A. Wąsowski, “Usage scenarios for feature model synthesis,” in VARY, 2012.

[SRA+13] S. She, U. Ryssel, N. Andersen, A. Wąsowski, and K. Czarnecki, "Efficient synthesis of feature models," submitted for review in *Information and Software Technology*, 2013.

1.4 Thesis Organization

In this chapter, we described the motivation behind feature model synthesis and presented an overview of our work. Chapter 2 describes feature models in detail, and provides background on variability modeling and software product lines. Chapter 3 discusses feature model synthesis scenarios and derives requirements for synthesis techniques from the scenarios. Chapter 4 discusses the Kconfig language—the variability modeling language developed and used by the Linux kernel—and compares the properties of 12 Kconfig models against a set of 267 published feature models. Chapter 5 describes the first synthesis algorithm, Feature-Graph-Extraction, that given a set of features and dependencies, extracts a special feature graph describing all feature diagrams that are sound with respect to the input dependencies. Chapter 6 describes a semi-automated algorithm that takes a feature graph and recovers a distinct feature model. Chapter 7 discusses related synthesis techniques, and Chapter 8 concludes with ideas for future work.

Chapter 2

Background

Large software systems such as operating systems, automotive software, or embedded systems, contain significant variability. This variability is often scattered over multiple artifacts such as optional requirements, high level features, or low level run-time options. Variability in code often takes the form of code fragments annotated with a condition. A variability model makes the variability in a software system explicit.

Feature modeling is one form of variability modeling. Feature modeling was first introduced in 1990 as part of the feature-oriented domain analysis (FODA) [KCH$^+$90]. Since then, feature modeling has become popular due to its ability to codify the critical information for reuse (e.g., variation points and variants), simplicity, understandability, and practicality [Kan10]. We discuss feature models in depth in Section 2.1.

Feature models are not the only form of variability modeling. For example, the popularity of variability modeling has led the Object Management Group (OMG) to work towards a standard called the Common Variability Language (CVL) [Obj12]. Other variability modeling languages include decision modeling [DGR11] and Clafer [BCW10]. The open-source community has developed the Kconfig [Zc] and Component Definition Languages (CDL) [BV00]. We discuss CVL, decision modeling and other variability modeling languages in the following section. We defer the discussion of the Kconfig language to Chapter 4.

Chapter Organization Section 2.1 introduces feature models and discusses their configuration semantics, domain semantics, and tool support. We briefly discuss extended feature models, e.g., feature attributes, and non-propositional feature models. In Section 2.2, we discuss variability modeling languages other than feature modeling. These include decision models, the Common Variability Language (CVL), Clafer, and

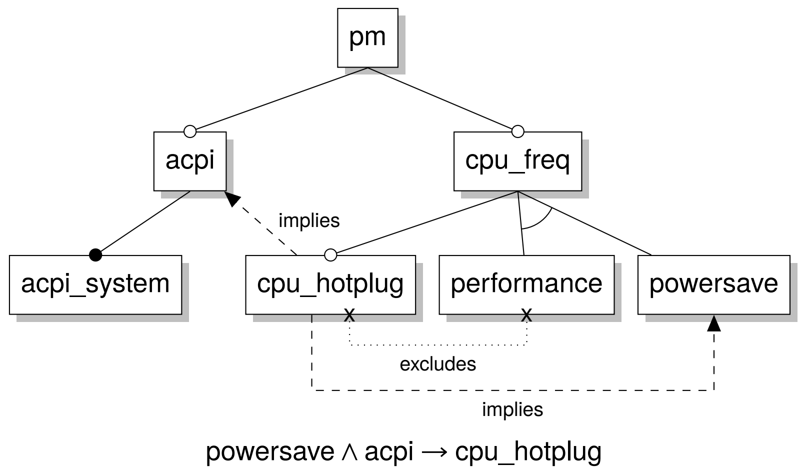


Figure 2.1: Power management feature model [SLB+11]

the Component Definition Language (CDL). Sections 2.3 and 2.4 describe how feature modeling fits into Software Product Lines (SPLs) and Feature-Oriented Software Development (FOSD).

2.1 Feature Modeling

Feature modeling was introduced by Kang et al. as part of feature-oriented domain analysis (FODA) [KCH+90]. A feature model (FM) consists of features arranged in (i) a feature diagram, and (ii) additional constraints between features. A *feature* is some property that is relevant to some stakeholder [CE00].

The *feature diagram* is a visual hierarchy of features. A feature may have subfeatures, i.e., a parent feature may have one or more children features. The topmost feature is called the *root feature* and represents the concept described in the feature model. All other features are either solitary or grouped. Solitary features can be *optional*—where the feature may or may not be selected if its parent is selected, or *mandatory*—where the feature must be selected if its parent is selected. Grouped features belong to feature groups that is either a MUTEX-, OR-, or XOR-group. A MUTEX-group requires that either one or none of the grouped features are selected. A OR-group requires that at least one of the grouped features are selected. An XOR-group requires exactly one grouped feature be selected. Additional cross-tree constraints in the form of *implies* and *excludes* edges can shown in the feature diagram. Arbitrary constraints can be added in a cross-tree formula shown below the feature diagram. The feature diagram along with the cross-tree formula form a feature model.

Figure 2.1 is a feature model of a power management subsystem. In this model, pm is the root feature. Features acpi and cpu_freq are optional features of pm. acpi_system is a mandatory feature. Features performance and powersave belong in an XOR-group. MUTEX- and OR-groups are not shown in this feature diagram. There are implies edges from cpu_hotplug to acpi, and cpu_hotplug to powersave, and an excludes edge between cpu_hotplug and performance. An additional cross-tree formula is shown below the diagram.

Definition 2.1. A *feature diagram* is a tuple $\mathsf{FD} = (\mathcal{F}, E, (E_m, E_i, E_x), (G_o, G_x, G_m))$ where:

- $\mathcal{F}$ is a finite set of features,
- $E \subseteq \mathcal{F} \times \mathcal{F}$ is a set of directed child-parent edges;
- $E_m \subseteq E$ is a set of mandatory edges,
- $E_i \subseteq ((\mathcal{F} \times \mathcal{F}) - E)$ is a set of *implies* edges,
- $E_x \subseteq 2^{\mathcal{F}}$ is a set of undirected *excludes* edges where $|e| = 2$ for all $e \in E_x$;
- G_o, G_x, G_m are sets that contain non-overlapping subsets of E that participate in OR-, XOR-, or MUTEX-groups respectively—each set in any of G_o, G_x, and G_m is disjoint from any other set in G_o, G_x, and G_m.

The following well-formedness constraints must hold in FD:

1. $(\mathcal{F}, E)$ is a rooted tree connecting all features in $\mathcal{F}$.
2. All edges in a group share the same parent, so if $g \in G_i$ for $i \in \{o, x, m\}$ and if $(f_1, f_2), (f_3, f_4) \in g$, then $f_2 = f_4$.
3. Sets E, E_i, E_x are pairwise disjoint.

Based on the definition above, features not having a mandatory edge are considered *optional features*. Feature groups in the diagram are also restricted to only OR-, XOR-, and MUTEX-groups. The original feature diagram notation introduced in FODA included only XOR-groups with $\langle 1..1 \rangle$ cardinality [KCH+90]. Czarnecki and Eisenecker extended the feature diagram to include OR-groups with $\langle 1..n \rangle$ cardinality [CE00]. While there are feature diagram notations that allow arbitrary $\langle m..n \rangle$ group cardinalities [SHTB07], we found that these group cardinalities were very uncommon in practice—in a dataset of 267 feature models contained in a feature model repository [MBC09], none of the models had feature groups with $\langle m..n \rangle$ cardinalities.

Definition 2.2. A *feature model*, $\mathsf{FM} = (\mathsf{FD}, \phi)$, where FD is the feature diagram and ϕ is a propositional formula over $\mathcal{F}$.

Feature models have been used to drive product derivation [AC04], domain analysis [KCH+90, ASB+08], model management, and describe design and implementation constraints in a software system. At its core, feature models describe variability as a set of legal configurations. In the following section, we define the configuration and domain semantics of a feature model.

2.1.1 Configuration Semantics

The configuration semantics of a feature model is a set of *legal configurations*—sets of selected features that respect the dependencies entailed by the diagram and the cross-tree constraints. The configuration semantics can be specified via translation to logic [Bat05]. We decompose the configuration semantics of a feature diagram by their graphical components in Table 2.1 and by using the formal definition in Definition 2.1 as follows:

Root Feature The root feature must be present in all configurations forming the following propositional constraint:

$$r \text{ where } r \text{ is the root of the tree } (\mathcal{F}, E) \tag{2.1}$$

Child-Parent Implications The feature hierarchy is represented as implications from a child feature to a parent feature. Given a child feature f_c (e.g., powersave) and a parent feature f_p (e.g., cpu_freq), the following constraints are added:

$$\bigwedge_{(c,p)\in E} (c \to p) \tag{2.2}$$

Mandatory Features The set of edges in E_m represent mandatory features—features that must be selected if its parent is selected. Mandatory features add the following

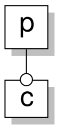
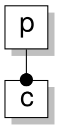
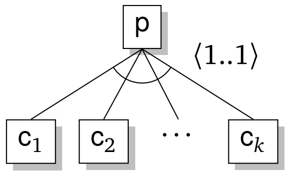
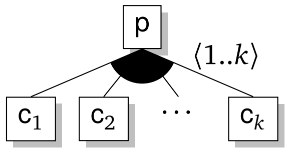
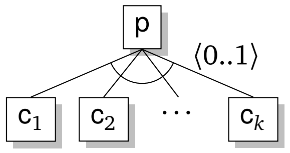
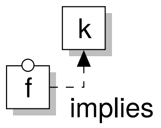
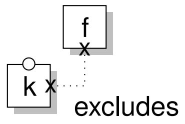
Feature diagram syntax	Propositional translation
c is an optional subfeature of p	
	$(c \to p)$
c is a mandatory subfeature of p	
	$(c \to p) \wedge (p \to c)$
$\{c_1, c_2, \ldots, c_k\}$ are an XOR-group of p	
	$(c_1 \vee \ldots \vee c_k \to p) \wedge$ $(p \to c_1 \vee \ldots \vee c_k) \wedge$ $\bigwedge_{\substack{i,j=1..k \\ i \neq j}} (c_i \to \neg c_j)$
$\{c_1, c_2, \ldots, c_k\}$ are an OR-group of p	
	$(c_1 \vee \ldots \vee c_k \to p) \wedge$ $(p \to c_1 \vee \ldots \vee c_k)$
$\{c_1, c_2, \ldots, c_k\}$ are a MUTEX-group of p	
	$(c_1 \vee \ldots \vee c_k \to p) \wedge$ $\bigwedge_{\substack{i,j=1..k \\ i \neq j}} (c_i \to \neg c_j)$
f has an implies edge to k	f and k share an excludes edge
 $(f \to k)$	 $(f \to \neg k)$

Table 2.1: Concrete syntax of feature diagrams and the mapping to propositional logic (adapted from [Käs10], [ACSW12], [Ber12])

constraint, in addition to the child-parent implications above:

$$\bigwedge_{(c,p)\in E_m} (p \rightarrow c) \tag{2.3}$$

Implies Edges An implies edge in E_i naturally adds an implication to the set of constraints:

$$\bigwedge_{(f,k)\in E_i} (f \rightarrow k) \tag{2.4}$$

Excludes Edges An excludes edge describe a mutual exclusion between two features. Excludes edges contribute the following constraints:

$$\bigwedge_{(f,k)\in E_x} (f \rightarrow \neg k) \tag{2.5}$$

Feature Groups G_o, G_x, G_m sets contains sets of non-overlapping edges that define the OR-, XOR-, MUTEX-groups respectively. Feature groups contribute two kinds of constraints: a requirement that at least one of the group members must be present, and mutual exclusions between group members. OR-groups have the former constraint, MUTEX-groups have the latter constraint, and XOR-groups have both. Feature groups contribute the following constraints:

$$\bigwedge_{\{(c_1,p),\ldots,(c_k,p)\}\in G_o\cup G_x} p \rightarrow c_1 \vee \cdots \vee c_k \tag{2.6}$$

$$\bigwedge_{\{(c_1,p),\ldots,(c_k,p)\}\in G_m\cup G_x} \left(\bigwedge_{i,j\in\{1,\ldots,k\} \text{ where } i\neq j} c_i \rightarrow \neg c_j \right) \tag{2.7}$$

Definition 2.3. The function $\mathsf{p}(\cdot)$ translates a feature diagram or a feature model to propositional logic, interpreting features as variable names. For a feature diagram $\mathsf{FD} = (\mathcal{F}, E, (E_m, E_i, E_x), (G_o, G_x, G_m))$, we define the configuration semantics as:

$$
\begin{array}{lrcl}
 & (\mathsf{acpi} & \rightarrow & \mathsf{acpi\_system} \land \mathsf{pm}) \\
\land & (\mathsf{acpi\_system} & \rightarrow & \mathsf{acpi}) \\
\land & (\mathsf{cpu\_freq} & \rightarrow & \mathsf{pm}) \\
\land & (\mathsf{cpu\_freq} & \rightarrow & \mathsf{powersave} \lor \mathsf{performance}) \\
\land & (\mathsf{cpu\_hotplug} & \rightarrow & \mathsf{powersave}) \\
\land & (\mathsf{cpu\_hotplug} & \rightarrow & \neg\mathsf{performance}) \\
\land & (\mathsf{cpu\_hotplug} & \rightarrow & \mathsf{acpi} \land \mathsf{cpu\_freq}) \\
\land & (\mathsf{powersave} & \rightarrow & \neg\mathsf{performance}) \\
\land & (\mathsf{powersave} & \rightarrow & \mathsf{cpu\_freq}) \\
\land & (\mathsf{performance} & \rightarrow & \mathsf{cpu\_freq}) \\
\land & (\mathsf{powersave} \land \mathsf{acpi} & \rightarrow & \mathsf{cpu\_hotplug})
\end{array}
$$

Figure 2.2: Propositional translation of the feature model in Figure 2.1

$$
\begin{array}{llr}
\mathsf{p}(\mathsf{FD}) = & r \land & \textit{root feature} \\
 & \bigwedge_{(c,p)\in E} (c \rightarrow p) \land & \textit{child-parent implications} \\
 & \bigwedge_{(f,k)\in I} (f \rightarrow k) \land & \textit{mandatory and implies edges} \\
 & \bigwedge_{(f,k)\in X} (f \rightarrow \neg k) \land & \textit{excludes edges} \\
 & \bigwedge_{\substack{\{(c_1,p),\ldots,(c_k,p)\}\\ \in G_o \cup G_x}} \big(p \rightarrow (c_1 \lor \cdots \lor c_k)\big) \land & \textit{feature groups} \\
 & \bigwedge_{\substack{\{(c_1,p),..,(c_k,p)\}\\ \in G_m \cup G_x}} \bigwedge_{\substack{i,j\in\{1,\ldots,k\}\\ i\neq j}} (c_i \rightarrow \neg c_j) &
\end{array}
\tag{2.8}
$$

where r is the root feature of the tree formed by the features and edges of $(\mathcal{F}, E)$.

Definition 2.4. Given $\mathsf{FM} = (\mathsf{FD}, \phi)$, the configuration semantics of a feature model, FM, is defined as $\mathsf{p}(\mathsf{FM}) = \mathsf{p}(\mathsf{FD}) \land \phi$.

Applying the propositional translation, $\mathsf{p}(\cdot)$, to the feature model in Figure 2.1 results in the formula in Figure 2.2. Any satisfying variable assignment to this formula is a legal configuration.

2.1.2 Domain Semantics

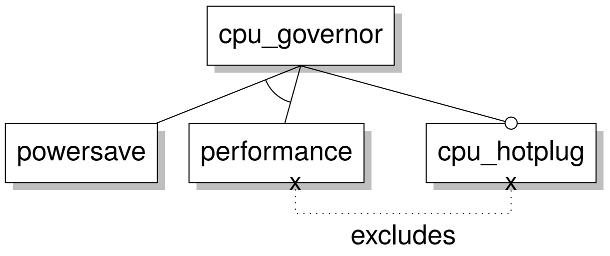
Feature models describe a set of legal configurations with its configuration semantics. A second semantics, the *domain semantics* (called ontological semantics in [SLB+11]), describe the domain in the structure of the model. The domain semantics describes the meaning of the features and is reflected in a model's hierarchy and feature group arrangement.

As we saw earlier in Figure 1.7, two feature models can describe the same configurations, but have different hierarchies. The different hierarchies convey a different domain semantics. The feature hierarchy is not the only part of a feature model that affects its domain semantics—the feature groups and cross-tree constraints also have an effect. For example, the feature models in Figure 2.3 all describe the same three configurations:

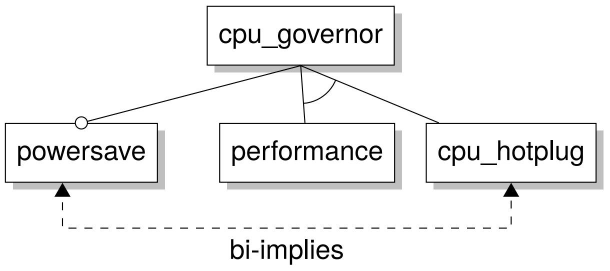
$$\begin{gathered}\{\mathsf{cpu\_governor}, \mathsf{powersave}\},\\ \{\mathsf{cpu\_governor}, \mathsf{powersave}, \mathsf{cpu\_hotplug}\},\\ \{\mathsf{cpu\_governor}, \mathsf{performance}\}.\end{gathered}$$

The features powersave and performance are both CPU governors, and cpu_hotplug controls whether the CPU can be enabled or disabled dynamically. If we examine the feature models in Figure 2.3, the models all have the same configuration semantics, but different domain semantics. In Figure 2.3a, powersave and performance are grouped together and an excludes edge exists between performance and cpu_hotplug. The features are grouped differently in the feature model in Figure 2.3b. Between the feature diagrams in Figures 2.3a and 2.3b, the arrangement of features in Figure 2.3b matches the domain relations of the features better.

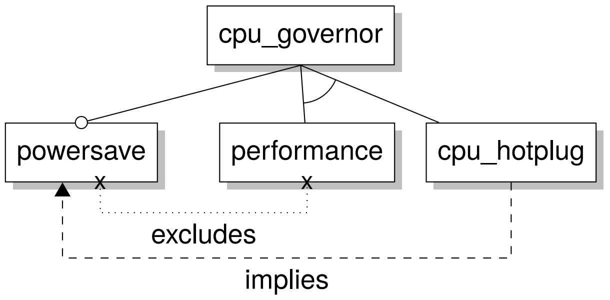
Between Figures 2.3b and 2.3c, the feature grouping and hierarchy are the same, however, the cross-tree constraints are different. How cross-tree constraints are represented in the feature model also impact the domain semantics of the model. Finally, in Figure 2.3d, the bi-implies edge (i.e., a shorthand for two implies edges) is redundant in terms of the configuration semantics—the edge may be removed without affecting the modeled set of configurations. However, the model builder may want to include this bi-implies edge to make the relation between powersave and cpu_hotplug exist.



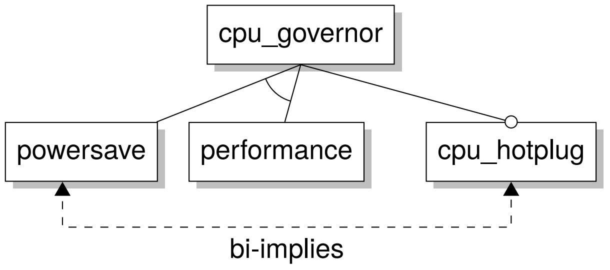
(a) powersave and performance are in a feature group



(b) performance and cpu_hotplug are in a feature group



(c) Same hierarchy and feature groups as b, but with different cross-tree constraints



(d) Redundant cross-tree constraint

Figure 2.3: Feature models that describe the same configurations, but have different domain semantics

The domain semantics are reflected in the hierarchy, feature groupings, and cross-tree constraints.

2.1.3 Tool Support

As feature models grow in size and complexity, tool support and automated analysis become a necessity [BPSP04]. Feature modeling tools include graphical configurators and automated analysis such as dead feature detection, valid configuration enumeration [BSRC10], fix generation [XHSC12], choice propagation and model completion [WSB+08]. Feature models have support for staged configuration [CHE05b], conflict resolution strategies [NE10], and support for different perspectives and collaboration for configuration [Hub12].

Examples of feature modeling tools are the FaMa-FW framework and the FAMILIAR domain-specific language. FaMa-FW is a framework for the automated analysis of feature models [TBRC+08]. It supports multiple reasoners, such as BDDs, SAT solvers, or constraint satisfaction problem solvers. FaMa-FW supports operations such as calculating the legal configurations, error detection and explanation. Acher et al. developed FAMILIAR—a domain specific language for reasoning on feature models [Ach11]. FAMILIAR supports operations such as feature configuration, detecting dead features, and counting or enumerating legal configurations. The key feature of FAMILIAR is its support for composing feature models. It integrates our feature model synthesis algorithm (Chapter 5) to perform operations between propositional logic and feature models. We discuss how our synthesis algorithm is used for feature model composition as one of the scenarios in Chapter 3.

TVL (Text-based Variability Language) is a language for specifying feature models with a C-like syntax [BCFH10]. TVL offers modularity mechanisms and goes beyond propositional feature models to supports feature cardinalities and attributes. Reasoning on feature models with feature cardinalities and attributes is beyond propositional logic and requires a reasoner such as a constraint satisfaction problem (CSP) solver, model checker, or a satisfiability modulo theories (SMT) solver. We discuss feature cardinalities and attributes in the following section. GUIDSL is another textual feature modeling language that is part of the AHEAD tool suite [Bat05, Bat04].

Pure-systems' pure::variants[1] and BigLever Software's Gears[2] are two commercial

[1]http://www.pure-systems.com/
[2]http://www.biglever.com/solution/product.html

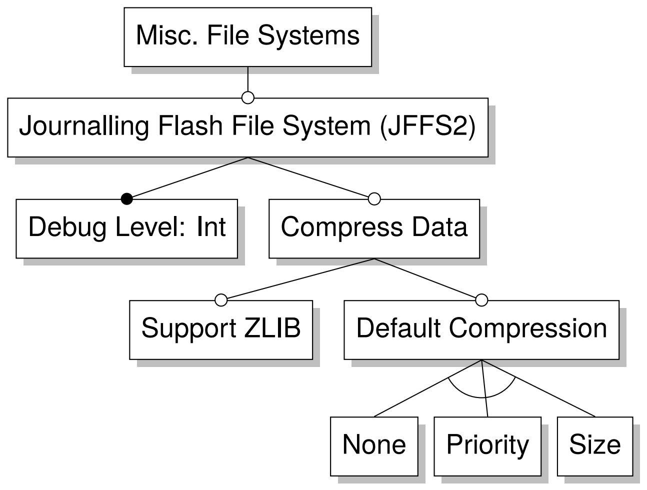
feature modeling tools. Both pure::variants and Gears go beyond feature modeling and support other aspects of a software product line workflow. For example, these tools are capable of managing configurations (i.e., variants) and support integration with other software artifacts such as requirements and build systems. We give an introduction to software product lines in Section 2.3.

In the open-source community, the Kconfig and Component Definition Languages (CDL) share many similar concepts with feature modeling, such as a feature hierarchy and feature groups [SSSPS07, SLB$^+$10]. Both tools have a graphical configurator. However, unlike feature modeling tools that use reasoners (e.g., BDDs or SAT solvers), Kconfig and CDL tools rely on an imperative implementation for performing configuration validation and conflict resolution. We describe our study on the Kconfig language in Chapter 4. For details on the CDL language, refer to Berger's dissertation [Ber12] and related work [BSL$^+$10b, BSL$^+$12].

2.1.4 Extended Feature Models

Since feature models were first introduced by Kang et al. [KCH$^+$90], feature models have been extended with feature attributes [KCH$^+$90, BSRC10], group cardinalities [RBSP02], feature cardinalities [CHE05a], or probabilities [CSW08, She08]. Schobbens et al. derive a new feature diagram notation—varied feature diagram (VFD)—that integrate various feature model extensions into a single feature diagram structure and providing a formal semantics [SHTB07]. We discuss several of these feature modeling extensions in this section.

Feature Attributes Figure 2.4 shows a feature model of the Journaling Flash File System, inspired by the Linux variability model. In this model, Debug Level is a mandatory feature with an integer attribute. A cross-tree constraint restricts this attribute to the values: 0, 1 or 2. While a standardized notation for feature attributes in feature models does not exist, an attribute consists of at least a name, a domain (e.g., integer, real, string), and a value [BSRC10]. The notation in Figure 2.4 is derived from the original feature model notation from FODA [KCH$^+$90]. Benavides et al. propose an alternative notation where attributes are attached to features with a separate box and line [BMAC05]. An advantage of their notation is that a feature can have more than one attribute, whereas the notation in Figure 2.4 allows a feature to have at most one attribute.



Support ZLIB $\rightarrow$ ZLIB Inflate

JFFS2 $\rightarrow$ CRC $\wedge$ MTD

$0 \leq$ Debug Level ≤ 2

Figure 2.4: A feature model of the Journalling Flash File System (JFFS2) with a feature attribute [BSL+10b, BSL+12]

Group Cardinalities Riebisch et al. propose group cardinalities [RBSP02] on feature models, where arbitrary multiplicities for features can be specified. For example, Riebisch present a notation for optional alternative features (i.e., what we refer to as a MUTEX-group) and feature groups with $[m..n]$ cardinality. We do not handle $[m..n]$ feature groups in our synthesis algorithms. However, our algorithms can be extended to detect these groups since their semantics can be described with propositional logic.

Feature Cardinalities Czarnecki et al. introduced cardinality-based feature models where a feature and its subfeatures can be cloned [CHE05a]. A feature cardinality is a restriction on the number of times a feature and its subfeatures can be reproduced. In a cardinality-based feature model, an optional feature is a special case of a feature with a feature cardinality of [0..1], and a mandatory feature has a cardinality of [1..1]. Czarnecki et al. formalize cardinality-based feature models by translating it to a context-free grammar [CHE05a]. Feature cardinalities share similarities with concept and class modeling. The Clafer language (Section 2.2.3) combines feature modeling with class modeling. We do not consider feature cardinalities since our algorithms reason on propositional logic.

Probabilistic Feature Models In our previous work, we introduced probabilistic feature models—feature models with support for soft constraints [CSW08, She08]. A *soft constraint* is one that should be satisfied by most, but not necessarily all configurations. A probabilistic FM models a distribution of configurations. The legal configurations of a probabilistic feature model can be represented as a Bayesian network [CSW08]. In this thesis, we do not consider synthesizing probabilistic feature models. Our previous work described a technique based on association rule mining to synthesize propositional logic [She08]. We discuss other probabilistic synthesis techniques in our chapter on related work in Chapter 7.

2.2 Other Variability Modeling Languages

A variability model makes variability in a software system explicit. The model provides a centralized artifact for describing domain analysis [KCH+90, ASB+08], driving configuration [CE00], or describing variability across multiple models [BCW10, Ach11]. Feature modeling is only one of many different variability languages. In this section,

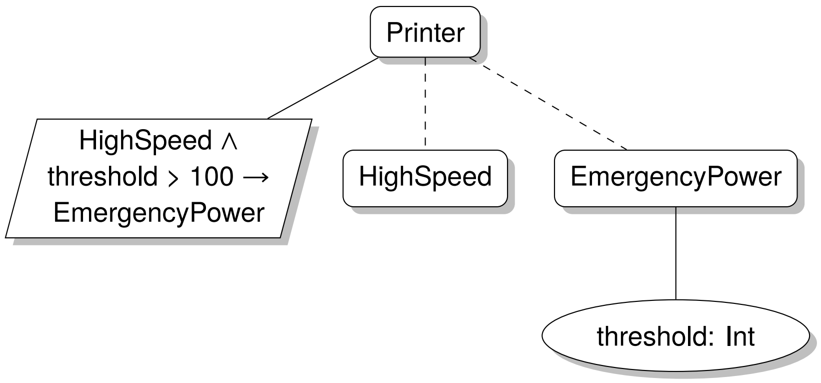


Figure 2.5: VSpec tree in CVL (adapted from [Obj12])

we briefly describe three other variability modeling languages: decision modeling, the Common Variability Modeling language (CVL), and concept modeling.

2.2.1 Decision Modeling

Decision models contain *"a set of decisions that are adequate to distinguish among the members of an application engineering product family and to guide adaptation of application engineering work product"* [Sof93, CGR+12]. Decisions describe the variation points in a product line and define the set of choices available at a certain point in time when deriving a product [DGR11]. As a result, derivation is a key application of decision modeling [CGR+12]. This differs from feature modeling, where feature models are also used to describe a domain. The DOPLER (Decision-Oriented Product Line Engineering for Effective Reuse) tool was developed by Dhungana et al. to support domain-specific definition of dependencies between model elements [DGR07, DGR11].

Czarnecki et al. compared feature modeling and decision modeling along ten dimensions [CGR+12]. Unlike feature modeling, decision models do not have a hierarchy. Instead, decisions are usually described as a list or in a table notation. Decisions in DOPLER can have a visibility condition that determines which decisions are visible during derivation [CGR+12]. Visibility conditions do not affect legal configurations. A hierarchy of decisions is derived from dependencies in visibility conditions.

```
Printer
  HighSpeed?
  EmergencyPower?
    threshold: int
  [HighSpeed && threshold > 100 => EmergencyPower]
```

Figure 2.6: Clafer model of the Printer from Figure 2.5

2.2.2 Common Variability Language (CVL)

The *Common Variability Language (CVL)* is an upcoming OMG standard for defining and resolving variability [Obj12]. Variation points are defined on a base model that can be any Meta Object Facility (MOF) model [Obj06]. The basic unit of variability in CVL is a variability specification (VSpec). VSpecs are arranged in a tree forming a CVL model such as the one in Figure 2.5. The rounded rectangles are choices that requires a *yes* or *no* decision, the ellipse is a variable requiring a value of a specified type, and the parallelogram represents propositional constraints (i.e., the constraint involving HighSpeed and threshold). A dashed edge represents an optional choice, while a solid edge represents a non-choice (i.e., the choice must always be yes). A *variation point* is bound to exactly one VSpec. The concrete syntax of VSpec trees is similar to feature models [Obj12]. However, a VSpec is more closely related to a decision in decision modeling than a feature in that it represents a variation point and not necessarily a program feature [CGR$^+$12].

2.2.3 Concept Modeling

Clafer is a lightweight modeling language with first-class support for feature modeling [BCW10]. Clafer provides a uniform representation of meta-modeling and feature modeling allowing the user to mix both feature and class models together in the same Clafer model. Clafer can act as a common language connecting different specialized variability modeling or meta-modeling languages. Figure 2.6 shows the Clafer model in feature modeling notation for the Printer meta-model from Figure 2.5. In Clafer, indentation is used to specify hierarchical nesting. The question mark indicates an optional feature or class. Clafer models translate to relational logic, and can be reasoned using a constraint solver for relational logic, such as KodKod [Tor09, TJ09].

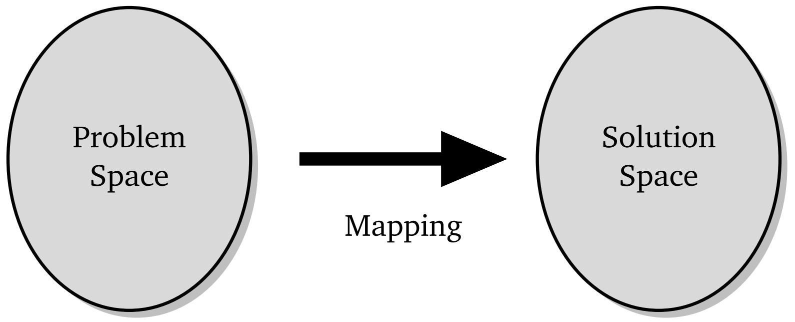


Figure 2.7: The problem space, mapping and solution space in SPLs (adapted from [Cza04])

2.2.4 Component Definition Language (CDL)

Variability modeling languages have also been developed outside of academic research. The Kconfig and Component Definition Language (CDL) were developed by the open-source community to support the variability modeling of the Linux and eCos operation system kernels respectively. We discuss the Kconfig language in Chapter 4.

CDL is used to specify the variability model in the eCos operating system[3]. In CDL, features are organized in a tree similar to feature modeling. Features in CDL have a state and an optional data value that acts like an attribute in feature modeling. A child feature can only be selected if its parent is selected. However, CDL allows features to be placed under a parent different from the one that it was syntactically declared by using the parent keyword. Configuration and visibility constraints are specified with the active_if and calculated clauses. We described the relation between feature modeling, CDL, and the Kconfig language in [BSL+10b, BSL+12]. Berger expands on CDL in his dissertation [Ber12].

2.3 Software Product Lines

Variability models are widely used in the development of software product lines, where they are used to describe static (compile-time) and dynamic (runtime) variability [CE00, SV06]. Software Product Lines (SPLs) enable systematic reuse of code across a family of related products with common and variable product characteristics [CN01]. Product

[3]http://ecos.sourceware.org/

line engineering, also called systems-family engineering, seeks to exploit commonalities in a given problem domain while managing the variabilities among them in a systematic way [Cza04, WL99, CN01]. Product line engineering separates the development process into two parts: domain engineering and application engineering [Cza04]. Domain engineering involves domain analysis, where commonalities and variabilities between system family members are identified. Reusable assets (e.g., domain models, reusable code components) are designed and implemented for realizing systematic reuse in the family members. Application engineering involves constructing concrete applications using the reusable assets developed in the domain engineering phase.

Software artifacts can be classified into the problem space, solution space, or the mapping (Figure 2.7). The *problem space* describes domain-specific abstractions that are common or variable across the family members. The *solution space* consists of implementation-specific abstractions. Solution space artifacts specify variability using either a compositional (i.e., features as modules) or an annotative approach (e.g., C preprocessor using `#ifdef` directives, Java annotations) [KAK08]. The *mapping* between the problem and solution spaces creates a specialized implementation given a specification from the problem space [Cza04].

Feature models are a core artifact of a feature-oriented software product line. Feature models are problem space artifacts that describes domain abstractions as features and their constraints. In the solution space, the solution space could consists code with annotated variability (e.g., C preprocessor with `#ifdef` conditions). The mapping could consists of a build system that invokes the C preprocessor (CPP) using values set through a FM configurator.

2.4 Feature-Oriented Software Development

Feature-Oriented Software Development (FOSD) is a paradigm for synthesizing programs for software product lines and domain engineering [AK09]. FOSD favors the systematic application of the feature concept to all phases of the software life cycle [AK09]. FOSD treats features as a first-class entity. Examples of tools include GenVoca and AHEAD tool suite which support FOSD and features as first-class entities through mixin modules [Bat04].

FOSD has four distinct phases: domain analysis, domain design and specification, domain implementation, and product configuration and generation [AK09]. The domain analysis stage determines the features and dependencies that make up the software system. Features are typically represented as a feature model. The design and implementation phases involve creating the implementation in the form of feature artifacts. A user configures and generates a variant in the configuration and generation phase respectively.

Feature implementations need to be specified in solution space artifacts. Specifying features can be separated into two approaches: compositional and annotative. Compositional approaches assume artifacts are decomposed by features and rely on a merging algorithm to compose artifacts to create a final implementation. Annotative approaches rely on conditional annotations placed within artifacts to separate feature implementations.

Feature model synthesis is useful for migrating an existing system to one that uses FOSD. Once features are identified, our synthesis algorithms can be used to construct a feature model.

Chapter 3

Scenarios and Requirements

FMs were first introduced for domain analysis as part of the feature-oriented domain analysis (FODA) [KCH+90]. Since then, feature models have been used for model management [Ach11], and describing design or implementation constraints [CE00]. The increasing uses of feature modeling have also led to an increasing number of scenarios requiring feature models synthesis. In this section, we introduce an abstract workflow for feature model synthesis, and describe each scenario as concrete instances of this workflow. We begin by giving a breakdown of software artifacts containing variability, then describe each scenario according to their input artifacts.

Chapter Organization In Section 3.1, we give an overview of the abstract feature model synthesis workflow and describe concrete workflows involving the two algorithms that we describe in this thesis. Section 3.2 describes the criteria used to classify the synthesis scenarios and Section 3.3 describes the scenarios. Finally, Section 3.4 extracts requirements from the scenarios for feature model synthesis techniques.

Publications Portions of the chapter were published in [SCW12].

3.1 Overview of Feature Model Synthesis

We begin by presenting an abstract workflow for feature model synthesis in Figure 3.1. This workflow consists of two stages: variability analysis and feature model synthesis. Variability analysis is responsible for analyzing the input artifacts and deriving the needed input for feature model synthesis. Feature model synthesis is responsible for

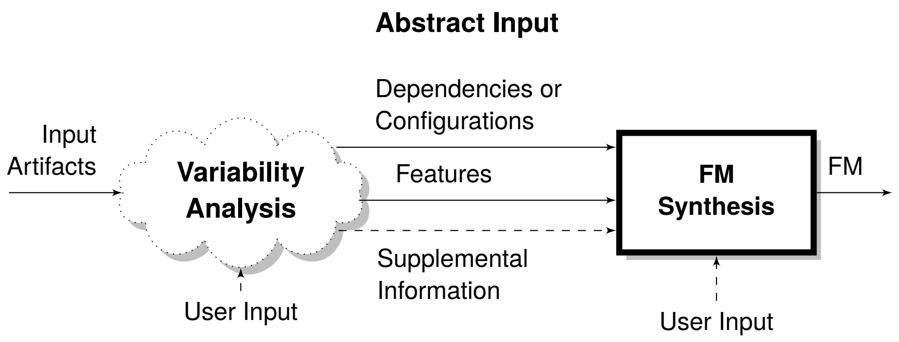


Figure 3.1: Variability analysis and feature model synthesis

creating a feature model given the derived input. We describe the stages in detail below.

Variability Analysis Variability analysis is responsible for recovering the *abstract input* that consists of (1) a set of features; and (2) valid feature combinations, represented as feature dependencies or configurations; and (3) any supplemental information used to help with the tree or group recovery. Whether dependencies or configurations are recovered depends on the type of input artifacts (Table 3.1). For example, dependencies can be recovered from a variable artifact, while a set of configurations is more appropriately recovered from a set of variants.

We separate the input artifacts of a variability-rich software project into the six categories in Table 3.1. The input artifacts are classified in terms of their level of abstraction by the columns. The rows describe whether the artifact is a *variable artifact* where variability is symbolically represented in the artifact, or enumerated as a *a set of instances*. We borrow terminology from the Common Variability Language (CVL) to distinguish an artifact's level of abstraction [Obj12]. An artifact that is a *variability abstraction* is at a high level of abstraction and contains concepts such as features [KCH+90]—properties that are relevant to some stakeholder [CE00]—or decision [SRG11]. An artifact is a *variability realization* if it describes how variability is realized or implemented in the system.

Different input artifacts require different forms of variability analysis. For example, given a set of requirements documents, where each document realizes a single variant [WCR09, NE08a], the analysis would involve analyzing each requirements document to extract features that are described in each. In another example, extracting variability from source code requires a form of code analysis such as TypeChef for C preprocessor annotated code [Käs10]. Berger et al. developed analyses for analyzing FreeBSD configuration templates and mining dependencies from its build system and documentation [BSL+10a]. A set of feature configurations could also be used directly as input to the algorithm, e.g., hardware configurations in FreeBSD. Our work focuses solely on feature model synthesis and relies on existing work to perform the variability analysis stage.

Feature Model Synthesis The stage following variability analysis, *feature model synthesis*, is responsible for building a feature model given the extracted abstract input. We decompose feature model synthesis into three stages (Figure 3.3): (1) DAG hierarchy recovery, (2) group and cross-tree constraint (CTC) recovery, and (3) tree hierarchy selection. *DAG hierarchy recovery* takes a set of features and a formula, and recovers a DAG that describe all hierarchies that imply the input formula. *Group and CTC recovery* identifies all feature groups and cross-tree constraints (CTCs) given the propositional formula, DAG and an optional tree hierarchy. Both the DAG hierarchy recovery, and the Group and CTC recovery stages are fully automated; no user input is required. Finally, since the hierarchy of feature models is a tree, the *tree hierarchy selection* stage selects a single tree from the set of possible trees from the DAG. Both DAG hierarchy and group and CTC recovery are fully automated steps; only tree hierarchy selection may need user input.

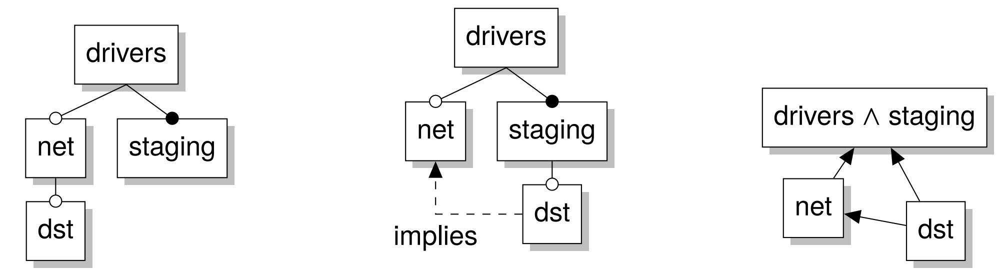
The *tree hierarchy selection* stage selects a distinct tree hierarchy from the possible hierarchies that describe the same set of configurations. We demonstrate why this stage is needed with the input in Figure 3.2a. The input describe three legal configurations as a disjunctive normal form (DNF) formula. The feature diagrams in Figure 3.2b and Figure 3.2c both describe the same input configurations. However, the meaning of the features are different depending on the selected hierarchy. In Figure 3.2b, dst, which

	Variability Abstraction	Abstraction-Realization Interface	Variability Realization
Variable Artifacts	Feature model	VPs and feature-to-VP mapping	Configurable platform *requirements, models, code, etc.*
Instances	Feature configurations	VP configurations	Variants *requirements, models, code, etc.*

Table 3.1: Breakdown of artifacts in variability-rich software

$$\begin{aligned} & (\mathsf{drivers} \land \mathsf{staging}) \\ \lor\ & (\mathsf{drivers} \land \mathsf{staging} \land \mathsf{net}) \\ \lor\ & (\mathsf{drivers} \land \mathsf{staging} \land \mathsf{net} \land \mathsf{dst}) \end{aligned}$$

(a) A set of configurations expressed as a propositional formula in DNF



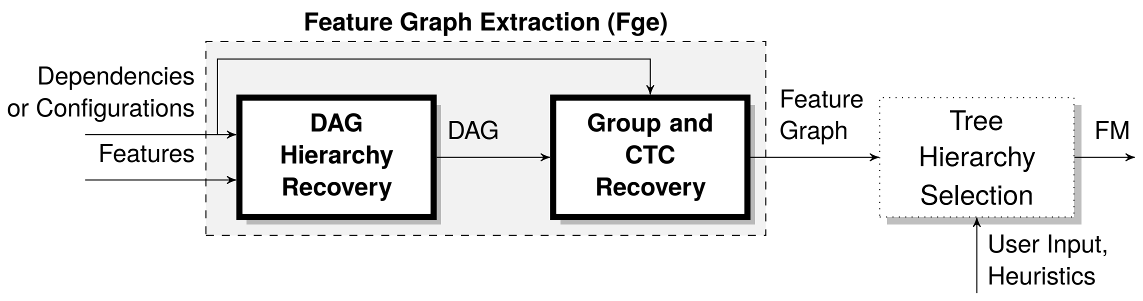
(b) A feature diagram (c) Another feature diagram (d) Feature graph

Figure 3.2: Different feature diagrams, and a feature graph, with the same configuration semantics describing the same set of configurations

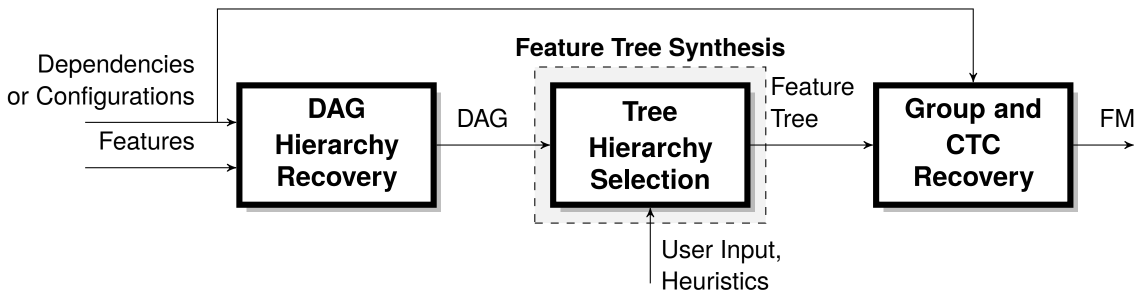
stands for *distributed storage*, is a subfeature of net where mature, tested networking features as placed. In Figure 3.2c, dst is a subfeature of staging, where features that are experimental or untested are placed. The meaning of dst is altered depending on the selected hierarchy. In general, some form of additional input is required to derive a tree hierarchy from the DAG in the tree hierarchy selection stage. For example, a modeler can use a tool-assisted approach that suggests parents based feature descriptions and dependencies to determine which features and dependencies to place in the hierarchy.

Finally, the last step consists of recovering feature groups (i.e., MUTEX-, OR-, or XOR-groups) and cross-tree constraint (CTC).

We describe two workflows for feature model synthesis that correspond to our two algorithms in this thesis. The first workflow with *late hierarchy selection* (Figure 3.3a), describes the FEATURE-GRAPH-EXTRACTION algorithm (Chapter 5). In this algorithm, we synthesize a feature graph that describes all possible feature diagrams that are implied by the input. The input is specified as dependencies as a formula in conjunctive normal form (CNF), or as a set of configurations as a formula in disjunctive normal form (DNF). The resulting feature graph can be used as a structure for other tools, like the interactive model builder by Janota et al. [JKW08]. The *early hierarchy selection* workflow (Figure 3.3b), selects a feature tree prior to recovering feature groups and



(a) Late hierarchy selection (automated)



(b) Early hierarchy selection (semi-automated)

Figure 3.3: Abstract workflows for feature model synthesis

cross-tree constraints. Early hierarchy selection reduces the search space for feature groups—the most computationally complex part of feature model synthesis—and is particularly useful for scenarios involving complex constraints and thousands of features, e.g., reverse-engineering a feature model for an operating system kernel. FEATURE-TREE-SYNTHESIS (Chapter 6) is our algorithm for the tree hierarchy selection stage that uses supplement feature descriptions and a textual similarity measure to rank and suggest potential parents for a feature. The algorithms in this thesis address all three components of feature model synthesis.

3.2 Scenario Criteria

Before we describe the scenarios, we first introduce the criteria used to classify the synthesis scenarios.

Input Artifacts Variability can come from many different input sources in a software project such as the categories that we showed in Table 3.1. The input could be variable artifacts (i.e. artifacts with variability), such as a feature model, or a requirements document, or code with variation points (VPs). The other input types are feature configurations (i.e. sets of selected features) or variants (i.e. artifacts with resolved variability), such a requirements document or code for a particular product.

Precision of Configuration Analysis The variability analysis stage is responsible for recovering the abstract input, i.e., features, valid feature configurations, and any supplemental information, needed by the synthesis algorithm. This criterion classifies the precision of recovering feature configurations from the input artifacts. We assume that the full set of features is recovered by the variability analysis prior to feature model synthesis.

We classify a scenario's analysis precision according to the four categories in Figure 3.4. A *sound and complete*, or exact recovery is one that is able to recover the exact set of configurations present in the input artifacts. A *complete* recovery is one that does not lose any configurations present in the input artifacts. Conversely, a *sound* recovery is one that does not add new configurations compared to what is the input.

Thüm et al. classified edits on a feature model into four categories: refactoring, specialization, refactoring, and arbitrary edit [TBK09]. A refactoring is equivalent

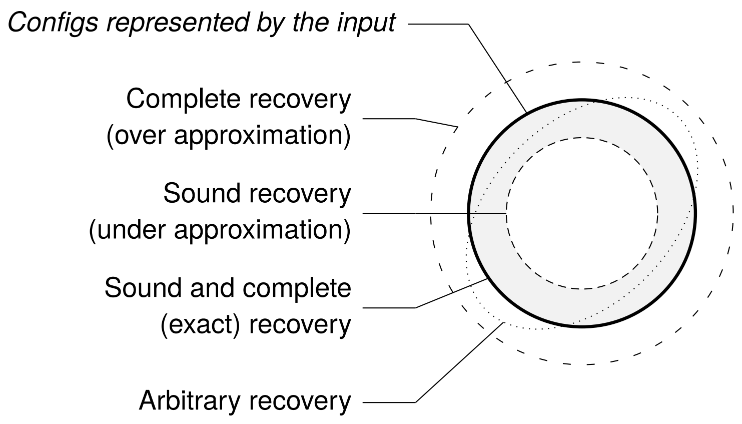

Configs represented by the input
Complete recovery (over approximation)
Sound recovery (under approximation)
Sound and complete (exact) recovery
Arbitrary recovery

Figure 3.4: Property of a transformation step

to a sound and complete recovery in our terminology. Specialization is the same as a sound recovery and a generalization is the same as a complete recovery. Finally, an arbitrary edit is the same as an arbitrary recovery. Thüm's terminology is specific to feature model edits, while our terminology is used to describe the precision of both configuration analysis and the feature model synthesis algorithm.

Required Synthesis Precision A feature model synthesis algorithm takes the abstract input recovered in the analysis stage and returns a feature model that describe the input. Different scenarios call for different synthesis precisions. For example, constructing a feature model to drive derivation could require an exact synthesis so that the resulting model describes exactly the input configurations. On the other hand, domain analysis may require a complete synthesis procedure to generalize the input configurations to better reflect the entire domain being modeled.

The classification of the required synthesis precision is based on the precision of the abstract input in Figure 3.4. For example, an exact recovery is one that synthesizes a feature that describes exactly the configurations in the abstract input.

Note that our definition of soundness and completeness is based on the set of configurations described in the model. There is a duality between configurations and dependencies such that a *sound* technique in terms of configurations is *complete* in terms of *the set of dependencies* and vice versa. In this dissertation, we primarily use the definition of sound and completeness with respect to the set of configurations.

Size We classify the size of a scenario based on its description and examples that we found in literature. In our classification, a *small* scenario has several hundred features. A *medium* scenario has roughly a thousand features, and a *large* scenario has several thousand features. We base this categorization on existing models found in literature and feature model repositories. The models in the SPLOT model repository have a median of 20 features and at most 290 features[1]. BigLever and pure-systems have reported that models in industry are typically in the range of hundreds of features. In the systems domain, we collected a set of 13 feature models with a median of 1600 features [BSL+12]. The smallest model in this collection was ToyBox with 71 features. The largest model was the Linux x86 kernel model (v2.6.32) with 6320 features. The FreeBSD kernel (v8.0.0) has 1203 features [SLB+11]. Finally, feature models with 5,000–10,000 features have been generated for testing feature model analysis tools on large-scale models [MWC09]. However, these numbers may change as more feature models become available to researchers in the future.

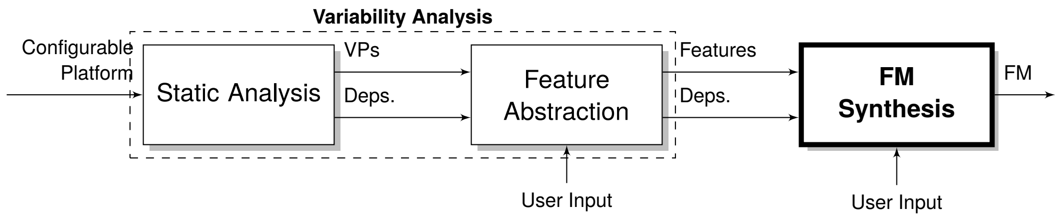
3.3 Scenarios

3.3.1 Scenario 1: Synthesis From a Configurable Platform

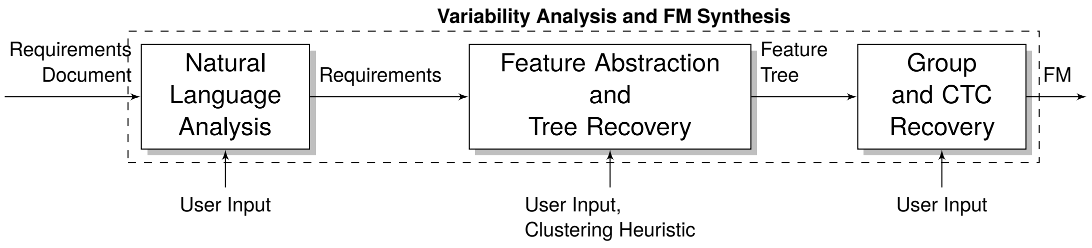
We begin describing our scenarios with synthesizing a feature model from a configurable platform consisting of variability-rich assets with variation points (VPs). The platform could be of different artifact types, such as requirements, models, or code, with each artifact containing VPs. We distinguish two cases based on the type of the input artifacts below.

Scenario 1a: Configurable platform (code) The input to this scenario is a configurable platform of code with VPs. For example, the FreeBSD kernel with 1203 features, is such a platform where the implementation is given in C with VPs defined using #ifdef preprocessor statements. The first stage in this scenario is to identify VPs and dependencies in the code using static analysis (Figure 3.5a). These VPs are fine-grained

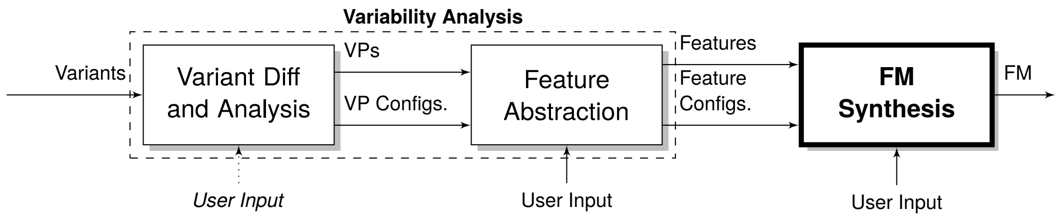
[1]based on the 232 models from http://www.splot-research.org as of August 5, 2012



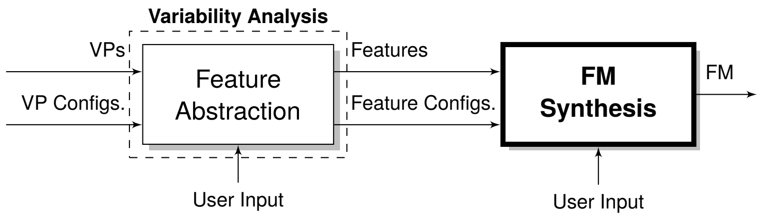
(a) **Scn. 1a.** Configurable platform: code [SLB+11]


Variability Analysis and FM Synthesis
Requirements Document
Natural Language Analysis
Requirements
Feature Abstraction and Tree Recovery
Feature Tree
Group and CTC Recovery
FM
User Input
User Input, Clustering Heuristic
User Input

(b) **Scn. 1b.** Configurable platform: requirements [NE08b, WCR09]

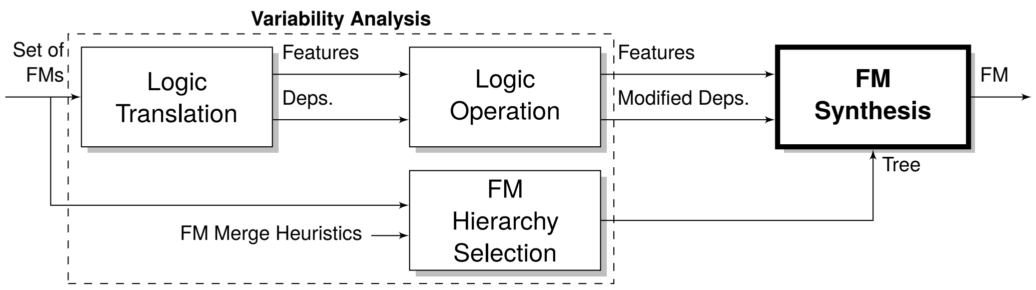


(c) **Scn. 2a, 2b, 2c.** Variants: requirements, models [RPK11], code [JDB07]

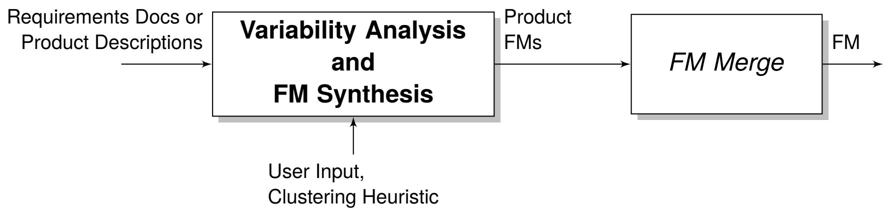


(d) **Scn. 2d.** Variants: VP configs

Figure 3.5: Scenario workflows



(e) **Scn. 3.** Feature model operations [Ach11]


Requirements Docs or Product Descriptions
Variability Analysis and FM Synthesis
Product FMs
FM Merge
FM
User Input, Clustering Heuristic

(f) **Scn. 4.** Approaches relying on FM merge:
product descriptions [ACP$^+$12], requirements [ASB$^+$08, CZZM05], and product FMs [HTM09]

Figure 3.5: Scenario workflows (cont.)

Scn.	Input Artifacts	Form*	Configuration Analysis Precision	Required Synthesis Precision	Size
1a.	Platform (code)	D	Complete	Sound	Med–Large
1b.	Platform (requirements)	N	Exact†	Arbitrary	Small
2a.	Variants (models)	C	Exact	Exact	Small–Med
2b.	Variants (requirements)	C	Exact†	Complete	N/A
2c.	Variants (code)	C	Exact	Complete	Small–Med
2d.	Variants (VP configs)	C	Exact	Complete	Small
3.	Feature models	D	Sound	Exact	Medium
4a.	Platform / Variants (descriptions)	M	Exact	Exact	Small
4b.	Platform / Variants (requirements)	M	Exact†	Arbitrary	Small
4c.	Platform / Variants (product FMs)	M	Exact	Exact	Small

*Abstract Input Form. C is *configurations*, D is *dependencies*, M is *product FMs*, and N is *natural language text*.
†Analysis performed manually.

Table 3.2: Scenario summary

and are closely related to the solution space and need to undergo a further feature abstraction step. While static analysis is typically automatic, it is not guaranteed to find all dependencies between VPs. As a result, the recovery of valid feature combinations is complete but unsound (i.e., an over-approximation of the configurations allowed by the platform). A sound synthesis is needed to compensate for the over-approximated abstract input. In other words, additional dependencies may need to be introduced during the synthesis. We summarized the properties of this scenario in Table 3.2. This scenario motivates our previous work on reverse engineering FMs [SLB+11].

Scenario 1b: Configurable platform (requirements) In this scenario, the input is a requirements document containing variability, i.e., optional requirements, that describes a product line. Niu et al. [NE08b] and Weston et al. [WCR09] both use this scenario to motivate their synthesis techniques. In Figure 3.5b, we show the workflow for this scenario. Requirements are first identified by a domain expert with the help of natural language analysis. Both Niu and Weston treat a feature as a group of requirements and use clustering to build the feature tree. For each cluster, an abstract feature is introduced with the clustered requirements as sub-features. This approach leads to the variability analysis and feature synthesis stages being intertwined. User input is needed in this stage to name abstract features or adjust properties of the clustering algorithm. This scenario also matches the experience described to us by an industry partner in the automotive sector, but in their case, the requirements clustering and feature model building were performed manually. Niu extracted 22 features from two sample applications [NE08b]. Weston used requirements documents describing a Smart Home with 87 requirements in total [WCR09].

3.3.2 Scenario 2: Synthesis from Variants

The second group of scenarios uses a set of variants where variability in each artifact is resolved. We identified four different kinds of artifacts containing variability among each variant.

Scenario 2a: Variants (models) In this case, the variants used as input for feature model synthesis are a set of related models. In this scenario's workflow (Figure 3.5c), the first step is to compare the variants and extract a set of VPs and VP configs. Rubin and Chechik describe a variability analysis for identifying similarities between model instances by comparing and matching model elements [RC10, RC12]. Ryssel et al. also described an algorithm that uses model matching and difference to identify VPs and abstract the model variants into a set of VP configurations [RPK12]. Since the input artifacts are in the solution space, VPs are at a much finer granularity than features and must undergo a feature abstraction step. In terms of the synthesis precision, Ryssel's scenario required that the feature model describe the exact configurations as the input input [RPK11]. For the size of this scenario, Ryssel's technique was evaluated on two case studies [RPK12]. The first case study involved a set of related models with 14 features across 17 variants [RPK12]. The second case study had 415 features over 49 variants.

Scenario 2b: Variants (requirements) This case was described to us by an industry partner and uses a set of requirements, each describing a specific variant as input. The workflow is identical to Scenario 2a. The requirements are compared and VPs are identified between each document. Each requirements document represents a VP configuration. The goal of variability analysis is to describe the precise set of configurations (i.e., products) making the variability analysis exact. However, the resulting feature model is intended for domain analysis in this scenario, thus requiring a complete synthesis that allows the modeler to generalize the feature model to describe the domain.

Scenario 2c: Variants (code) In this scenario at Danfoss drives described in an experience report by Jepsen et al. [JDB07], code variants developed using a clone-and-own approach are used as input. The developers first compared and merged the code variants into a single codebase by placing conditional compiler flags on code fragments based on the products that contain the fragment. Each conditional fragment is a VP. A VP configuration consists of all included code fragments for a product. These VPs were then abstracted to features and a feature model is synthesized from the resulting set of features and configurations. Similar to Scenario 2b, the synthesis is intended to allow more configurations making it complete.

Scenario 2d: Variants (VP configs) This case uses a set of VPs and VP configurations directly as input where each configuration represents a product in the product line. This scenario was described to us by an industry partner in the automotive sector. The company wanted to build a feature model that described their existing product line and supported instantiation of future products. The input configurations are exact, however, complete synthesis is required to support additional configurations in the synthesized feature model.

3.3.3 Scenario 3: Feature Model Operations

Scenarios 1 and 2 involved synthesizing a feature model from software artifacts containing variability. In this scenario, feature models themselves are used as input to feature model synthesis. Acher describes model management operations on feature models that include merge, difference, or projection [Ach11]. Similarly, Fahrenberg et al. discuss use cases for a semantic difference between models using feature models as an example [FLW11].

Figure 3.5e depicts the workflow for faeture models operations as described by Acher [Ach11]. The analysis stage consists of translating the input feature models to propositional logic based on their configuration semantics [Bat05]. The operation is performed on the propositional formulas and used as input to a feature model synthesis technique [CW07]. The input formula describes the set of configurations from the input models, modulo the operations semantics, making the analysis step sound.

In this scenario, the feature model operations are automated. Unlike the previous scenarios where user input is required for the synthesis stage, Acher uses heuristics for automatically determining the resulting feature tree based on the input feature models [Ach11].

3.3.4 Scenario 4: Feature Model Merge Workflows

Our last set of scenarios synthesize an individual feature that describe each product, then rely on a merge algorithm to create the final feature model describing all products.

Scenario 4a: FM merge-based (product descriptions) In this scenario described by Acher et al. [ACP+12], semi-structured product descriptions are used as input. The product descriptions are similar to feature configurations, however, product descriptions can contain variability. For example, a product could support one or more storage methods, or exactly one operating system. Each product description is transformed into a product feature model that describes the specific description and its variability. The resulting set of product feature models are then merged to create a featur emodel describing variability in all products. Acher used product descriptions ranging from 9 to 190 features as input [ACP+12].

Scenario 4b: FM merge-based (requirements) In this case, a set of requirements documents, each describing a single product, is used as input. Alves et al. [ASB+08] applied clustering to build a feature model describing each document, then merged these individual models to create a feature model describing all documents. Chen et al. also applied clustering to synthesize an individual feature model for each requirements document where each document describes a single variant [CZZM05]. The individual feature models are merged in a subsequent step. Alves' scenario used two requirements document as input. One had 23 requirements while the other had 59 requirements [ASB+08]. In terms of the scenario's size, Chen used requirements from two sample applications where one application had 21 requirements [CZZM05].

Scenario 4c: FM merge-based (supplier-specific FMs) Hartmann et al. described a scenario where a software product line has several suppliers for its components where suppliers may overlap in terms of their functionality [HTM09]. Each supplier models the variability in their component as a feature model. These feature models are then merged to form a so called supplier independent feature model—a feature model describing the product line across all suppliers. In this scenario, the variability analysis is exact since the exact configuration semantics of the input supplier feature models are maintained. Similarly, the synthesis of the supplier independent feature models must maintain the exact configurations described by the input supplier feature models.

3.4 Discussion and Requirements for Feature Model Synthesis

We summarized the properties of the scenarios in Table 3.2. Based on these properties, we derive requirements for feature model synthesis algorithms and describe each of these requirements below:

Variability Analysis Different input artifacts require different variability analysis techniques. Analysis for a variable artifact is significantly different compared to analysis for a set of instances—even for the same artifact type. For example, a configurable platform consisting of preprocessor annotated C code requires a different variability analysis compared to a set of variants, each implemented as individual program variants. Examples of variability analysis tools include TypeChef by Kästner et al. [Käs10]. TypeChef is a parser and type checker that can analyze a configurable of C preprocessor annotated code. Analyzing individual program variants would require a code difference tool to identify VPs.

Synthesis Input Form While the types of software artifacts containing variability is very large, the forms of input for a synthesis algorithm is considerably less. In Scenarios 1 and 2, these artifacts abstract to either dependencies or a set of configurations. Scenario 3, operates on feature models that are translated to a propositional formula for synthesis. A propositional formula can be considered as dependencies (i.e., a formula in CNF, or a BDD) or a set of configurations (i.e., a formula in DNF) depending on its form. From the scenarios, we identified four input forms for FM synthesis techniques:

Configurations A configuration consists of a enumeration of features present in a single instance of the software system. For example, the input artifacts to the scenarios in Scenario 2 are a set of variants and naturally translate to a set of configurations.

Dependencies Dependencies are a symbol representation of a set of configurations. In Scenario 1a, a configurable platform such as FreeBSD does not have a predetermined portfolio of products. Instead, instances of the platform are derived by a user based on configuration templates. Any configuration that satisfies the dependencies described in the platform (i.e., in code or documentation) are valid configurations. As a result, the set of valid configurations can quickly grow to be very large as the number of VPs or features grows in the platform. Using dependencies as input for feature model synthesis provides a compact, symbolic representation of the set of valid configurations. Scenario 3 also requires dependencies as input to synthesis. In feature model operations, the feature models are first translated to dependencies through their configuration semantics [Bat05] and an operation is applied on the dependencies to create a modified formula. This formula is used as input to the synthesis algorithm.

Natural Language Requirements Requirements make a natural source of variability in a software system. The scenarios involving requirements often combined both the variability and analysis stages (e.g., Figure 3.5b) since the synthesis techniques relied on clustering to both identify features, as well as construct the feature hierarchy. When a set of requirements for variants were used as input, the variability analysis and synthesis stages were separate (Scenario 2b). In this scenario, the synthesis algorithm can relies not on the natural language text, but rather on the extract VPs between the requirements document. As a result, the synthesis a algorithm can simply consider a set of configurations as input.

Product FMs The last input form consists of a set of individual feature models, where each feature model represents a single product in a product line. Scenarios that first synthesize a feature model for each product require two synthesis stages. The first stage synthesizes a feature model describing an individual product using properties specific to the input artifact. For example, Acher's approach relies on a specification and heuristics to build a feature model from a product description [ACP$^+$12]. Chen et al. apply clustering on manually constructed requirements resource graphs (RRGs) to create feature models [CZZM05].

Hierarchy Selection Given a set of valid feature combinations, there is more than one possible feature hierarchy. Figure 3.6 shows two feature models that describe the same configurations but with different hierarchies. To address this problem, some synthesis

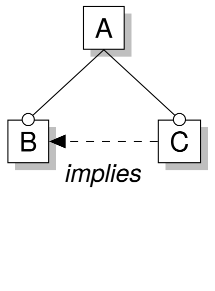


Figure 3.6: Another example of two feature models with the same configurations

techniques recover a directed acyclic graph (DAG) that represents all possible feature hierarchies given the abstract input [ACSW12, CSW08, CW07, DGH^{+}11, RPK11].

Techniques have also been proposed to select a distinct feature tree. Our own semi-automated technique uses a textual similarity measure to identify relevant parents given a feature [SLB^{+}11] (Chapter 6). Janota et al. developed an interactive model building tool that used a set of dependencies and a recovered DAG as input to restrict the possible model editing operations so that the resulting feature model was entailed by the input [JB08]. The two techniques that we mentioned are both semi-automated approaches. Hierarchy selection can be made automatic by using heuristics to select the hierarchy. For feature model operations (Scenario 3), Acher determines a feature tree automatically by using heuristics based on the input FMs [Ach11]. Acher et al. combines both heuristics and a manual specification for deriving the hierarchy from product descriptions (Scenario 4) [ACP^{+}12]. Clustering techniques build hierarchy by grouping related requirements under abstract features [ASB^{+}08, CZZM05, NE08b, WCR09].

Synthesis Precision The precision of a synthesis technique is dependant upon the precision of the analysis step and the expected use of the FM. For example, in Scenario 1a, the abstracted feature combinations is complete, or an over approximation. As a result, a sound synthesis technique is needed to add additional constraints to remove unwanted configurations. However, the abstracted feature combinations in Scenario 2b are exact, but require a complete synthesis to remove unwanted constraints and support additional configurations. Our previous work [ACSW12, CW07] and Ryssel's approach [RPK11] derives an exact FM describing the input. We later extended our approach with a feature similarity heuristic to deal with complete, but unsound input [SLB^{+}11]. The interactive model building tool by Janota et al. support building FMs that are complete with respect to the input. Clustering techniques are not focused on maintaining a set of input feature combinations, but are geared towards exploratory activities such as domain analysis. As a result, we classified the required synthesis precision as arbitrary, since constraints can be added or removed during FM synthesis.

Scalability The input form affects the scalability of a synthesis technique. Techniques that use dependencies as input are more scalable than techniques that use a set of configurations since dependencies represent a set of configurations symbolically. Other factors that affect a synthesis technique's scalability is the required amount of user interaction. Techniques requiring significant user interaction become intractable as the size of the models grow. However, user interaction can be replaced by using heuristics to automate the process and is largely dependent on the supplemental information extracted from the input artifacts (e.g. hierarchy data). Finally, the reasoning technique can affect a synthesis technique's scalability. We found that our SAT-based implementation significantly improved upon our BDD-based approach [ACSW12]. Ryssel et al. use a FCA-based approach [RPK11]. An evaluation comparing the different reasoning techniques is planned for future work.

Probabilistic Feature Models A special case involves synthesizing a probabilistic FM from configurations. A probabilistic model provides soft constraints that are valid in most configurations, but not necessarily all [CSW08]. Our previous work recovered a probabilistic FM describing framework usage in a set of framework applications [She08]. Dumitru et al. recover a probabilistic FM to for a recommender system that provides features for domain analysis [DGH+11]. Fukuda et al. build a probabilistic model to identify trends in a product transaction database [FAY11]. We proposed a technique for building a probabilistic FM by using association rule mining on a dataset of configurations [CSW08].

3.5 Scenario Conclusions

Different scenarios impose different requirements on a feature model synthesis workflow. For example, scenarios involving domain analysis may require a natural language analysis tool to extract variability from requirements, and a synthesis algorithm that constructs a feature diagram that is weaker, or complete, with respect to the input configurations. Synthesizing a feature model from source code on the other hand, would require a variability-aware static analysis tool, and a sound synthesis capable of handling the dependencies that were not identified by the analysis. Other considerations for synthesis algorithms include the size of the input and whether a probabilistic model is needed. The scenarios and requirements that we derived could be used to encourage research on new synthesis techniques and promote new applications and improvements to existing synthesis techniques.

Chapter 4

Real World Variability Models

Variability modeling and supporting tools have grown in popularity over the years. However, realistic benchmarks for evaluating variability modeling tools have generally been inaccessible with the growing interest from both tool vendors and researchers [SRC09]. While some variability models are already available, e.g., SPLOT model repository[1], very few of them originate from realistic processes; most are small examples from research publications, or outcomes of student run case studies. Given the lack of realistic large-scale models, many tool builders have resorted to using randomly generated models [WSB$^+$08, TBRC$^+$08, MWC09, BSRC10].

This chapter describes our research on realistic variability models extracted from open-source projects. We focused our study on the variability model of the Linux kernel and its variability modeling language called Kconfig. Kconfig was developed specifically for supporting the variability modeling and configuration of Linux. The Kconfig language share so many similarities with feature modeling that a Kconfig model can be interpreted as a feature model [SSSPS07]. Our study includes a total of 12 variability models specified with the Kconfig language and compare their properties against a set of 267 feature models gathered from the SPLOT model repository.

Chapter Organization In Section 4.1, we introduce the Kconfig language and its concepts. We present the abstract syntax for the Kconfig language in Section 4.2 and describe its mapping to feature modeling concepts in Section 4.3. In Section 4.4, we compare models properties of the variability models of the Linux kernel and 11 other projects with those of the models gathered from the SPLOT model repository. The formal semantics of Kconfig are available in Appendix A for those interested.

[1] www.splot-research.org

Publications Our first publication involving the Kconfig language was a study of the Linux variability model [SLB+10]. In [BSL+10b], we compared variability modeling concepts, semantics, usage and tools in Kconfig, CDL and feature modeling using Linux and eCos operating system kernels as the study subjects. We expanded on this study in [BSL+12], to include 12 Kconfig models, variants of the eCos model, and 267 SPLOT models. The Kconfig semantics were published as a technical note [SB10].

External Contributions The qualitative study in Section 4.4.6 was performed by Rafael Lotufo and published in [SLB+10]. The study of the Kconfig language and the Linux variability model has material from [BSL+10b] and [BSL+12]. The Kconfig semantics [SB10] in Appendix A were developed collaboratively with Thorsten Berger.

4.1 Kconfig Language

The Linux kernel originally used the Configuration Menu Language (CML) as its configuration language until v2.5.45 in 2002, where it was replaced with Kconfig (referred to as LinuxKernelConf at the time) [Wik12]. The Kconfig language is supported by several configurators. The 'menuconfig' tool is terminal interface for configuring a Kconfig model, the 'nconfig' tool that replaces the menuconfig tool using the ncurses[2] framework, and the 'xconfig' is a graphical configurator written using the Qt framework. The structure of a Kconfig model closely resembles a feature model in that configuration options are nested to form a hierarchy. We describe the features of the Kconfig language next.

Feature Kinds There are four types of nodes in a Kconfig tree: menu, menuconfig, config, choice. Menuconfigs and configs declare symbols that appear in a configuration. Choices and menus are grouping constructs that do not appear in a configuration. A Kconfig configuration is simply a list of key-value pairs, where symbols can be undefined if the corresponding config is unselected. We discuss the four feature kinds using the Kconfig snippet in Figure 4.1:

[2] https://www.gnu.org/software/ncurses/

```
menuconfig MISC_FILESYSTEMS
  bool "Miscellaneous filesystems"


if MISC_FILESYSTEMS

config JFFS2_FS
  tristate "Journalling Flash File System" if MTD
  select CRC32 if MTD


 config JFFS2_FS_DEBUG
  int "JFFS2 Debug level (0=quiet, 2=noisy)"
  depends on JFFS2_FS
  default 0
  range 0 2
  --- help ---
    Debug verbosity of ...


 config JFFS2_FS_WRITEBUFFER
  bool
  depends on JFFS2_FS
  default HAS_IOMEM


 config JFFS2_COMPRESS
  bool "Advanced compression options for JFFS2"
  depends on JFFS2_FS


 config JFFS2_ZLIB
  bool "Compress w/zlib..." if JFFS2_COMPRESS
  depends on JFFS2_FS
  select ZLIB_INFLATE
  default y


 choice
  prompt "Default compression" if JFFS2_COMPRESS
  default JFFS2_CMODE_PRIORITY
  depends on JFFS2_FS

  config JFFS2_CMODE_NONE
   bool "no compression"

  config JFFS2_CMODE_PRIORITY
   bool "priority"

  config JFFS2_CMODE_SIZE
   bool "size (EXPERIMENTAL)"
 endchoice
endif
```

Figure 4.1: Kconfig excerpt adapted from [BSL+10b]

configs Configs declare a unique symbol used to represent individual configuration options. Configs are the basic symbols in a Kconfig model and can hold a Boolean, tristate (i.e., three-valued), integer, hexadecimal, or string value. JFFS2_FS at Line 7 in Figure 4.1 is a tristate config in our snippet. Nesting between configs are shown as a tree-view with indentation in the graphical configurator. See Figure 4.2 for the graphical rendering of the Kconfig snippet.

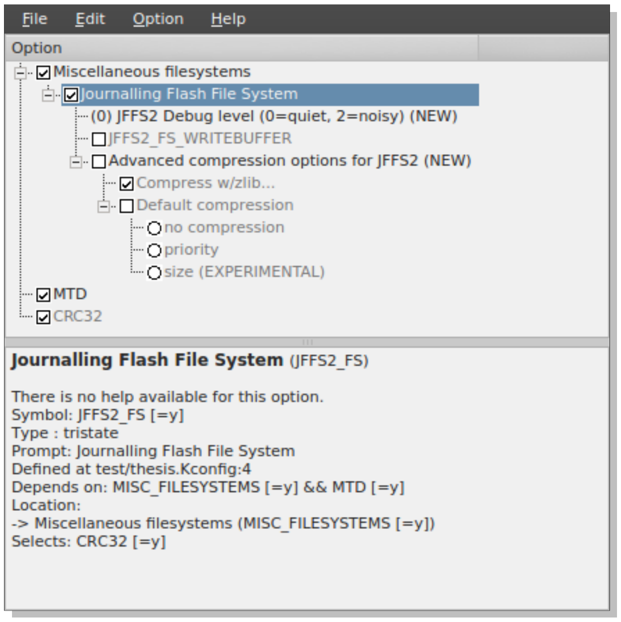
menus Menus are used purely for lexically grouping features in the configurator. Menus do not declare a symbol, and thus, do not appear in a configuration.

menuconfigs Menuconfigs have the same appearance as menus in the graphical configurator, however, menuconfigs also declare a symbol and can hold a value just like configs. MISC_FILESYSTEMS is a menuconfig and all features within the following if condition (Line 5) are children of MISC_FILESYSTEMS.

choices Choices are similar to feature groups in feature modeling where they are used to impose grouping constraints on its children, called choice members. Choices can be of Boolean or tristate type. A Boolean choice allows at most one choice member to be selected, and a tristate choice allows more more than one choice menmber to be selected. By default, a choice requires that at least one child be selected, however, choices can be made optional with the optional keyword. Choice members are declared as configs. *Default compression* in Line 39 is an example of a choice grouping.

Feature Representation In Kconfig, a configuration assigns a single value to each feature. A special empty value is used to denote the absence of a feature. There are five types that restrict the valid values of configuration options in Kconfig: bool, tristate, int, hex, and string.

The bool type has two possible values, `y` and `n`, internally represented by the numbers 2 and 0; 0 denotes feature absence, while 2 means that the feature's implementation is compiled statically into the kernel. The tristate type is an extension of bool with an additional `m` value, represented internally by 1. The `m` value denotes that the feature should be compiled as a dynamically loadable module. For example, JFFS2_ZLIB has type bool and JFFS2_FS is tristate. Kconfig supports two integer types: int (decimal) and hex (hexadecimal). Both types also allow an empty value, which is used to encode the absence of the numeric feature. The type string is ambiguous when handling the empty string: a string feature with the empty value can be seen as a present feature with



```
☑ Miscellaneous filesystems
    ☑ Journalling Flash File System
        (0) JFFS2 Debug level (0=quiet, 2=noisy) (NEW)
        ☐ JFFS2_FS_WRITEBUFFER
        ☐ Advanced compression options for JFFS2 (NEW)
            ☑ Compress w/zlib...
            ☐ Default compression
                ○ no compression
                ○ priority
                ○ size (EXPERIMENTAL)
☑ MTD
☑ CRC32
```

```
Journalling Flash File System (JFFS2_FS)

There is no help available for this option.
Symbol: JFFS2_FS [=y]
Type : tristate
Prompt: Journalling Flash File System
Defined at test/thesis.Kconfig:4
Depends on: MISC_FILESYSTEMS [=y] && MTD [=y]
Location:
-> Miscellaneous filesystems (MISC_FILESYSTEMS [=y])
Selects: CRC32 [=y]
```

Figure 4.2: xconfig rendering of the Kconfig snippet in Figure 4.1

that value or an absent feature; the two cases are indistinguishable when examining a Kconfig configuration.

Hierarchy Configuration options in Kconfig are organized into a hierarchy similar to feature modeling. In feature modeling, the hierarchy imposes a configuration constraint where a child feature implies the presence of its parent feature. In Kconfig, the parent-child implication is not enforced—a child feature can be selected when its parent is not. Furthermore, a symbol can appear in multiple places in the hierarchy, but are ultimately the same feature and share the same values. These properties leads to a separation between a Kconfig model's *syntactic hierarchy* (as it appears in the configurator) and the *configuration semantics*—constraints imposed on the set of configurations.

Each node in Kconfig has an associated prompt condition that controls its visibility. The visibility condition specifies when a feature is visible in the configurator and can be changed by a user. A visibility condition that evaluates to `m` or `y` causes a feature to be visible in the configurator. Furthermore, the visibility conditions of features is used to determine the syntactic hierarchy of a Kconfig model. In terms of the configuration semantics, the visibility condition imposes an *upper bound* on the allowed values of the feature.

Operator	Kconfig	Evaluation
no	`n`	0
mod	`m`	1
yes	`y`	2
and	`A && B`	min(A,B)
or	`A || B`	max(A,B)
not	`!A`	2 - A
equals	`A = B`	if (toString(A) = toString(B)) 2 else 0

Figure 4.3: Evaluating operators and literals in the three-valued logic of Kconfig

The hierarchy rendered by the *xconfig* configurator for the Kconfig snippet in Figure 4.1 is shown in Figure 4.2. The MISC_FILESYSTEMS feature is at the top of the hierarchy because of the enclosing if condition over all other features. Features that follow JFFS2_FS depend on JFFS2_FS making it their parent. The JFFS2_ZLIB feature has a specific prompt condition, i.e., visibility condition, that causes it to be nested under JFFS2_COMPRESS; however, JFFS2_ZLIB does not depend on JFFS2_COMPRESS according to its configuration semantics. In other words, JFFS2_ZLIB can be selected even if JFFS2_COMPRESS is deselected, thus, violating the parent-child constraint seen in feature modeling.

Constraints and Expression Language Kconfig uses a three-valued logic for expressing and evaluating expressions. At its core, Kconfig has three literals: `n` (no), `m` (module), `y` (yes). There are five operators: and, or, not, equals, and not equals. Figure 4.3 shows the rules for evaluating these operators. The equals operator has a unique property in that it uses the string value of its arguments. For example, the literal `y` and the string "y" would be equal in Kconfig.

A symbol can declare the following constraints: prompts, defaults, select clauses, or valid ranges for int or hex symbols. Each constraint may have a condition to describe when the constraint is active. The prompt condition controls when a symbol is visible and *changeable by the user* in the configurator. As a result, the prompt condition determines the visibility condition of the config.

A default condition is used to set a default value for a symbol. Under most circumstances, a default enforces a soft constraint—a suggestion for the user that can be overridden and does not enforce a constraint on the set of valid configurations. However, a default can interact with prompt conditions creating a hard configuration constraint. This situation occurs if a symbol's prompt condition evaluates to *false*; the value of the symbol cannot be changed, and the symbol takes the value of the first active default.

A select clause enforces the selection of another symbol if the current symbol is also selected. Select clauses stem from the imperative nature of the Kconfig configurators.

The depends on clause is syntactic sugar for applying a condition across all constraints in a symbol's declaration. For example, in line 14, the depends on statement adds JFFS2_FS as a dependency to the prompt, default and range.

4.2 Abstract Syntax

Now that we have introduced the concrete syntax, we derive an abstract syntax to describe the core concepts of Kconfig. Kconfig models consists of a set of configs nodes and a set of choices nodes. Let Kconfig denote the set of all possible models in the Kconfig language:

$$\mathsf{Kconfig} = \mathcal{P}(\mathsf{Configs}) \times \mathcal{P}(\mathsf{Choices}) \tag{4.1}$$

A single Kconfig model is denoted by $m \in \mathsf{Kconfig}$. Configs are the main components of a Kconfig model. In the abstract syntax, Configs consists of both config and menuconfig nodes from the concrete syntax. We omit menus from the abstract syntax since they are used purely for grouping features in the configurator. Constraints declared on menus can be propagated to its ancestor nodes when translating a Kconfig model in the concrete syntax to its abstract syntax.

4.2.1 Configs

A config is defined as the tuple:

$$\mathsf{Configs} = \underset{\text{Identifier}}{\underset{\uparrow}{\mathsf{Id}}} \times \underset{\text{Type}}{\underset{\uparrow}{\mathsf{Type}}} \times \underset{\substack{\text{Prompt}\\ \text{Condition}}}{\underset{\uparrow}{KExpr(\mathsf{Id})}} \times \underset{\text{Defaults}}{\underset{\uparrow}{\mathsf{Default}*}} \times \underset{\substack{\text{Reverse}\\ \text{Dependency}}}{\underset{\uparrow}{KExpr(\mathsf{Id})}} \times \underset{\text{Ranges}}{\underset{\uparrow}{\mathcal{P}(\mathsf{Range})}} \tag{4.2}$$

where,

- Id is the set of possible string identifiers;
- $\mathsf{Type} = \{\text{boolean}, \text{tristate}, \text{int}, \text{hex}, \text{string}\}$;
- $\mathit{KExpr}(\mathsf{Id})$ is the set of all three-valued expressions over Id in the Kconfig language. We define the abstract syntax of the expression language in the following section;
- $\mathsf{Default} = \mathit{KExpr}(\mathsf{Id}) \times \mathit{KExpr}(\mathsf{Id})$ denotes defaults. The first *KExpr* in the default denotes a default value—the value that the config will take if this default was active. The second *KExpr* denotes the condition required for the default to become active;
- $\mathsf{Range} = (\mathsf{Int} \cup \mathsf{Hex} \cup \mathsf{Id}) \times (\mathsf{Int} \cup \mathsf{Hex} \cup \mathsf{Id}) \times \mathit{KExpr}(\mathsf{Id})$ is a triple consisting of a lower bound, an upper bound and a condition.

The first component of a config is a unique identifier used in expression references and in configurations. The second component denotes the type of the config. The third component is the prompt expression—the condition that determines when the config is visible and changeable by the user. The fourth component is a ordered set of defaults. Defaults are ordered such that the config takes the value of the first enabled default. The fifth component is the reverse dependency expression—the disjunction of the configs and their select conditions in the Kconfig model. For example, the following snippet in the concrete Kconfig syntax translates to the reverse dependency expression A && C || B && D for config E:

```
config A                          config C ...
    bool "Feature A"              config D ...
    select E if C                 config E ...

config B
    bool "Feature B"
    select E if D
```

The last component of a config is a range triple. The range triples only apply to configs with a numeric type (i.e., int or hex) and defines a lower bound, and upper bound and a condition when the range constraint is active.

4.2.2 Choices

The second component of a Kconfig model is a set of choice nodes. A choice is an abstract construct that defines no symbol in the configuration, however, it imposes additional constraints on its members. We define choices as a quadruple consisting of a type where boolean or tristate are the only valid types, a flag indicating whether the choice is mandatory, a prompt condition, and a set of identifiers indicating its members. The set Choices is defined as:

$$\mathsf{Choices} = \{\text{boolean}, \text{tristate}\} \times \mathsf{Bool} \times \mathit{KExpr}(\mathsf{Id}) \times \mathcal{P}(\mathsf{Id}(m)) \tag{4.3}$$

4.2.3 Identifiers and Expressions

Let $\mathsf{Id} \in \mathcal{P}(\mathsf{String})$ be a set of names identifying a config. For example, JFFS2_FS in Figure 4.1 is an identifier. Let $\mathsf{Const} = \mathsf{Tri} \cup \mathsf{String} \cup \mathsf{Hex} \cup \mathsf{Int}$ be the set of values assignable to configs and available as constants in expressions, where $\mathsf{Tri} = \{0_t, 1_t, 2_t\}$. Tri is ordered such that $0_t < 1_t < 2_t$. The Tri, String, Hex, and Int domains are disjoint (i.e. mutually exclusive).

Expressions are defined as a set $\mathit{KExpr}(\mathsf{Id})$ over Id generated by the following grammar, where $e \in \mathit{KExpr}(\mathsf{Id})$, $iv \in \mathsf{Id} \cup \mathsf{Const}$, $\otimes \in \{\text{or}, \text{and}\}$, $\ominus \in \{=, \neq\}$:

$$e ::= e \otimes e \mid \text{not } e \mid iv \ominus iv \mid iv \tag{4.4}$$

4.3 Mapping to Feature Modeling Concepts

Kconfig declares symbols (i.e., configs) that represent features in the software system. Features are organized in a tree hierarchy in the same way as a feature model. Also similar to feature groups in feature modeling, features can be grouped in a *choice* node such that exactly one group member, or one or more group members must be selected. Kconfig supports cross-tree constraints just like feature modeling.

Sincero et al. described a mapping from feature modeling concepts to Kconfig [SSSPS07]. In this section, we describe the reverse mapping—a mapping from Kconfig concepts to feature modeling concepts. Table 4.1 shows the mapping of Kconfig to feature modeling concepts. We will describe each section of the mapping next.

Concept	Kconfig	Feature Modeling
Feature Kinds		
Grouping	menu, menuconfig, choice	feature (non-leaf)
Individual	config	feature (leaf)
Feature Representation		
Composition	single value	bool. value with opt. attribute
Feature Type		
Switch	bool, tristate	optional
Data	int, hex, string	integer, string
None	menu	mandatory
Group Constraints		
Mutex [0..1]	optional bool. choice	MUTEX-group
Or [1..n]	mandatory tristate choice	OR-group
Xor [1..1]	mandatory Boolean choice	XOR-group
Feature Constraints		
Configuration	select	cross-tree constraint
Value Restriction	range	cross-tree constraint[1]
Derived Features	non-prompt config[2]	rare [DHR10]
Soft Defaults	prompt default	rare [CE00]
Visibility Conditions	prompt condition	rare [DHR10, SJ04]
Expression Operators	&&, ||, !, =, !=	not standardized[3]
Binding Modes	three-valued logic	rare [CE00, SJ04]
Other		
Textual Content	prompt, help	description
Modularization	textual inclusion	rare[1,4]
Build Symbols	one-to-one mapping	unspecified[1]
Code Mappings	no, uses Kbuild	unspecified[1]

[1] Not supported by models in the SPLOT model repository
[2] Derived features are described in more detail in Section 4.3.3
[3] SPLOT models support propositional logic operators
[4] [BCFH10, BEGB11, BMB+10, DGRN10]

Table 4.1: Mapping of concepts between Kconfig and feature modeling (adapted from [BSL+12])

```
menu "Operating System"

  config SCHEDULER
    prompt "CPU scheduler"

endmenu
```

(a) Lexical nesting

```
config PM
  prompt "Power management support"

config PM_DEBUG
  prompt "Debug support"
  depends on PM
```

(b) Nesting using common dependencies

Figure 4.4: Specifying Hierarchy in Kconfig

```
choice
  prompt "An XOR-group"

  config A
    bool "Option A"

  config B
    bool "Option B"

endchoice
```

(a) XOR-group

```
choice
  prompt "A tristate group"

  config C
    tristate "Option C"

  config D
    tristate "Option D"

endchoice
```

(b) Tristate group

```
choice
  prompt "A MUTEX-group"
  optional

  config E
    bool "Option E"

  config F
    bool "Option F"

endchoice
```

(c) MUTEX-group

Figure 4.5: Feature group representation in Kconfig

4.3.1 Hierarchy

The hierarchy in Kconfig is specified using lexical nesting with menus (e.g., Figure 4.4a) and common dependencies with configs (e.g., Figure 4.4b). In this example, PM_DEBUG will be nested under PM because of its dependency. Interestingly, the nesting is determined purely based on syntactic dependencies. If PM_DEBUG had the dependency of !PM, i.e., it requires that PM is deselected for the config to be selected, PM_DEBUG would still be nested under PM.

4.3.2 Feature Groups

The Kconfig choice construct is similar to a feature group in feature modeling. We interpret a choice with boolean members (Figure 4.5a) as an XOR-group since exactly

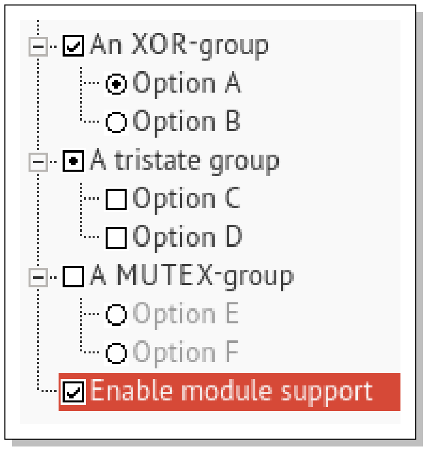


Figure 4.6: Feature group rendering in xconfig configurator

one member must be selected if the choice is active. A choice with tristate members is OR-group (Figure 4.5b). A MUTEX-group is a choice with boolean members and the optional keyword (Figure 4.5c).

Figure 4.6 shows the feature group rendering in the xconfig configurator. The XOR-group is rendered as radio buttons since one choice must be made. The OR-group can be switched to accept either Boolean or tristate values. When the choice is switched to accept tristate values, both “Option C” and “Option D” can be selected like a OR-group. The circle in a square box indicates that the config is set to *module* (m). The MUTEX-group can be disabled so that no member is selected.

Kconfig has no syntactic structure equivalent to an OR-group in feature modeling[3]. A tristate choice can be toggled between a tristate or Boolean state. When in the Boolean state, the choice operates like an XOR-group—only one member may be selected. However, when in the tristate state, zero or more choice members may be selected. In Figure 4.6, the tristate group is in the tristate state. None of the group members have to be selected. The user can click on the tristate choice to change it to a Boolean state requiring either “Option C” or “Option D” to be selected in the figure.

[3]Our previous publications [SLB+10, BSL+10b] interpreted tristate choices (Figure 4.5b) as OR-groups. We have since identified that tristate choices behave differently than OR-groups, where the member cardinality constraint changes depending on whether it is in a tristate or Boolean state.

```
config JFFS2_COMPRESS
  bool “Advanced compression options”

config JFFS2_ZLIB
  bool “Compress w/zlib” if JFFS2_COMPRESS
  default y
```

Figure 4.7: A conditionally derived config in the Linux kernel

4.3.3 Feature Constraints

Cross-tree constraints are realized with specific combinations of the prompt condition and defaults or reverse dependencies in Kconfig. In most cases, a default provide a suggested value that can be overridden by the user. However, defaults can interact with the prompt condition that determines whether the config can be changed by the user. A config's prompt condition determines the condition that it is configurable by the user. Figure 4.7 shows an example of an prompt condition on the JFFS2_ZLIB config.

We categorize configs into three constraint categories based on the interaction between their prompt and default conditions: (1) configs configs, (2) derived configs, and (3) conditionally derived configs.

For the following definitions, given a config $c \in \mathsf{Configs}$, we define functions $\mathsf{prompt}(c)$ and $\mathsf{defaults}(c)$ that retrieve the condition on the respective property for the config as a *set* of satisfying assignments. Similarly, function $\mathsf{rev\_dep}(c)$ retrieves the reverse dependency of the config as a set of satisfying assignments. For defaults, $\mathsf{defaults}(c)$ is the union of the satisfying assignments of all defaults for the config c. We use set notation to distinguish operations on the three-value logic of Kconfig compared to the propositional operations on propositional logic.

Definition 4.1. A *derived config* is never visible, thus its value is derived from a default or reverse dependency (which is none are specified, is the value n):

$$\{c \mid c \in \mathsf{Configs} \wedge \mathsf{prompt}(c) = \emptyset\} \tag{4.5}$$

Definition 4.2. *Configured configs* are selected (i.e., have a value of y or m) iff the config is visible, otherwise it must remain deselected (i.e., have a value of n):

$$\{c \mid c \in \mathsf{Configs} \wedge \mathsf{prompt}(c) \neq \emptyset \wedge \mathsf{prompt}(c) = (\mathsf{defaults}(c) \cup \mathsf{rev\_dep}(c))\} \tag{4.6}$$

Definition 4.3. A *conditionally derived config* can be in a selected state when the config is both visible or invisible. For this occur, the config must have a prompt condition, and there must also be a default condition or reverse dependency that differs from the prompt condition:

$$\{c \mid c \in \mathsf{Configs} \wedge \mathsf{prompt}(c) \neq \emptyset \wedge \mathsf{prompt}(c) \neq (\mathsf{defaults}(c) \cup \mathsf{rev\_dep}(c))\} \qquad (4.7)$$

The config JFFS2_ZLIB in Figure 4.7 is a conditionally derived config. JFFS2_ZLIB is visible (i.e., its prompt condition is satisfied) when JFFS2_COMPRESS is selected. The user is free to select a value for the config in this case. If JFFS2_COMPRESS was unselected, then JFFS2_ZLIB is invisible and takes on its default value of y since there is no condition attached to the default.

4.3.4 Feature Descriptions and Code Mappings

Features in Kconfig can declare a short description with a prompt, or a longer description as help text. We extract the feature names and descriptions for our textual similarity heuristic in Chapter 6. Each config in Kconfig declares a symbol that directly corresponds to a C preprocessor symbol. For example, the config JFFS2_FS in Figure 4.1 generates the build symbol CONFIG_JFFS2_FS with the configured value. The implementation references this build symbol using C preprocessor #ifdef statements. The mapping from features to files is specified in the Makefile-based Kbuild infrastructure for the Linux kernel [BSL+10a].

4.4 Comparison with Available Feature Models

The Kconfig language was originally developed to support variability management and configuration in the Linux kernel. Since its introduction, other projects have adopted Kconfig. We gathered a total of 12 Kconfig models, including the Linux variability model and compared them with the 267 published in the SPLOT model repository.

4.4.1 Kconfig Models

Our dataset of Kconfig models include 11 projects, including the model from the x86 architecture of Linux v2.6.32 [BSL+12]. We compared their statistics with the models gathered from the SPLOT model repository. We give a brief description of each Kconfig project below:

Linux Linux is an open-source operating system kernel. Linux uses the Kconfig language to specify its variability model used to configure the kernel for compilation. The Kconfig language has been used by Linux since 2002 and was developed to address the shortcomings of the previous configuration language used by Linux, CML. The Linux variability model is the largest open source variability model known to date.

axTLS axTLS is a small library implementing the TLS v1 SSL protocol.

BuildRoot BuildRoot is a set of Makefiles for creating a complete embedded Linux system.

BusyBox BusyBox combines many UNIX commands into a single executable. BusyBox is optimized to reduce size and to operate on devices with limited resources.

CoreBoot CoreBoot is a free, open-source replacement for proprietary BIOS in a range of computer systems.

EmbToolKit EmbToolKit stands for Embedded Systems ToolKit, and is a tool for building all necessary tools, including the root filesystem, for an embedded Linux system.

Fiasco Fiasco is a real-time, microkernel that supports a range of architectures such as x86, ARM, or PowerPC.

Freetz Freetz is a free, open-source firmware for the AWM FritzBox internet routers.

ToyBox ToyBox, like BusyBox, combines a set of UNIX commands into a single executable. ToyBox is a smaller project than BusyBox and was mostly developed by its creator.

uCLinux uCLinux is a Linux distribution for embedded systems. It contains a modified version of the Linux kernel designed for microcontrollers and also included libraries and packages to build a complete system for embedded systems. uCLinux employs a multi-level configuration process where the first model (ucLinux-base) provides architecture and library choices. The second model is that of the Linux kernel. The choices in the first model determine a default configuration for The second model. The third model (ucLinux-dist) is used select software packages to be included in the system distribution.

uClibc uClibc is a C library designed for embedded systems. It is smaller in size than glibc, and nearly all applications that work on glibc will also work with uClibc.

4.4.2 SPLOT Models

The SPLOT model repository contains feature models gathered from academic papers and experience reports, created by users with their online model editor, or contributed by industry. We gathered a total of 267 models for our dataset. The SPLOT models range in their domain; there are models that describes electronic shops, video games or mobile phones. We list the SPLOT models that we analyzed in Appendix B.

4.4.3 Model Structure

In this section, we example properties of the Kconfig models and contrast them with the properties of the feature models from SPLOT.

Configs Violating Hierarchy Rules In Kconfig, the hierarchy is derived from a combination of syntactic nesting and common dependencies between configs. Unlike feature modeling, a child config does not have to imply its parent config. We identified the configs that violated hierarchy rules in our Kconfig models using a SAT solver with the propositional translation of the models based on their formal semantics (Appendix A). Figure 4.8 shows the proportion of configs violating hierarchy rules in the Kconfig models.

In Linux v.2.6.32, about 6% of configs do not imply their parent. Many of these were part of a sub-tree of configs under DVB_FE_CUSTOMISE, that stands for *"Customise the frontend modules to build"*. Subfeatures of DVB_FE_CUSTOMISE were specific chipsets that were selected automatically through defaults when selecting support for digital TV adapters in a separate tree. When DVB_FE_CUSTOMISE was deselected, the subfeatures were not visible in the configurator. Enabling DVB_FE_CUSTOMISE

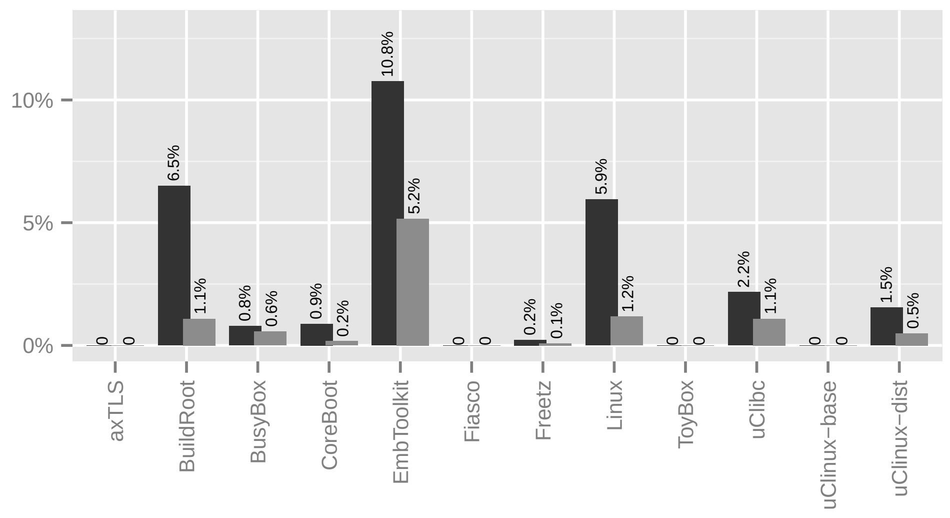


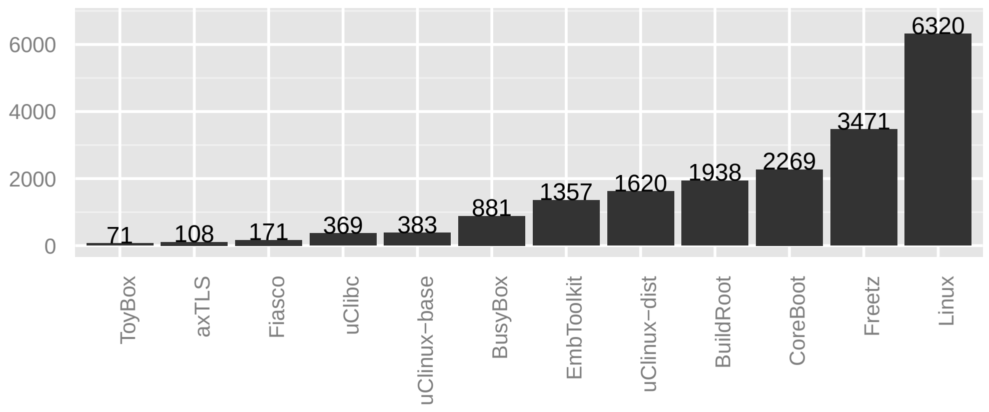
Figure 4.8: Proportion of configs that violate hierarchy rules in the Kconfig models. The black bar represents configs that do not imply their parents; the grey bar represents parents with children that are do not imply them.

would make the subfeatures appear in the configurator, enabling the user to fine-tune, and select the exact chipsets to compile and include in the kernel.

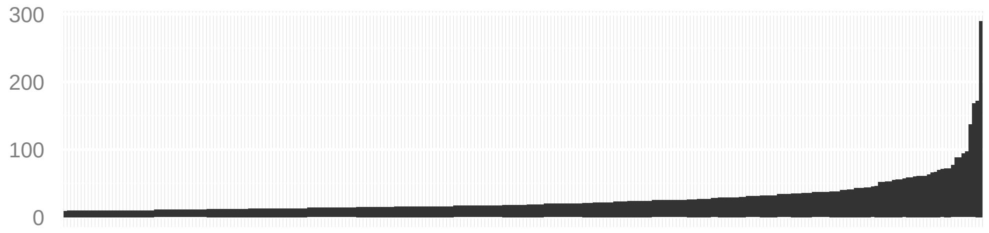
Size Figure 4.9a shows the size of the Kconfig models in our dataset. The Linux model is by far the largest, with 6320 features, and ToyBox is the smallest with 71 features. The models have a median of 1119 features, and a mean of 1580. The feature counts include all nodes in the Kconfig models—configs, menuconfigs, menus, and choices.

The size of the SPLOT models are shown in Figure 4.9b. The models in this dataset are significantly smaller, with the largest model having 290 features, and the smallest model having only 9 features. The models have a median of 18 features and a mean of 27. The models in the Kconfig dataset are significantly larger than the feature models available in the SPLOT model repository. The Linux Kconfig model is also more than 20 times larger than the largest model in the SPLOT repository.

Feature Types For the Kconfig models, we further partitioned the features into five categories: grouping, grouped, data, boolean, and tristate. Grouping features are

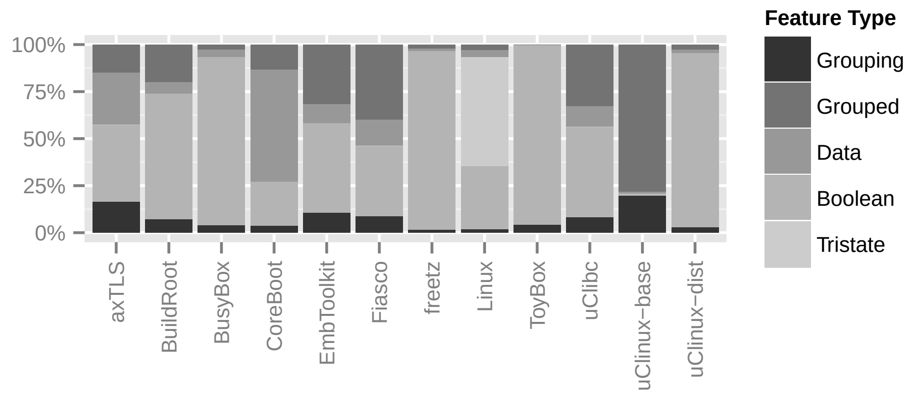


(a) Number of features in Kconfig models

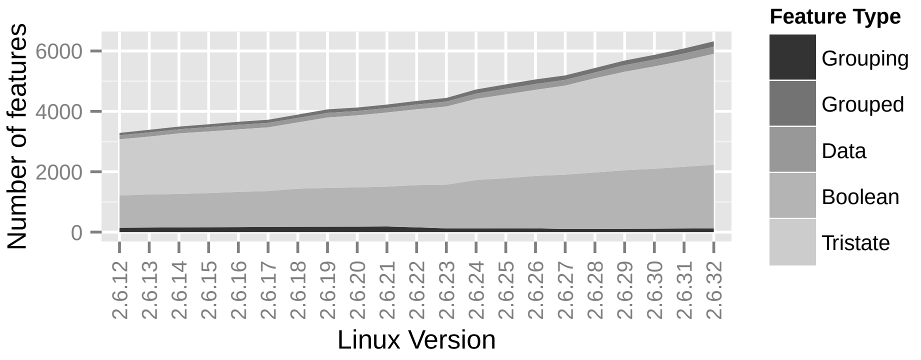


(b) Number of features in SPLOT models

Figure 4.9: Size of Kconfig and SPLOT models



(a) Feature types in the Kconfig models



(b) Feature types in the Linux Kconfig model over several versions

Figure 4.10: Feature types in the Kconfig models

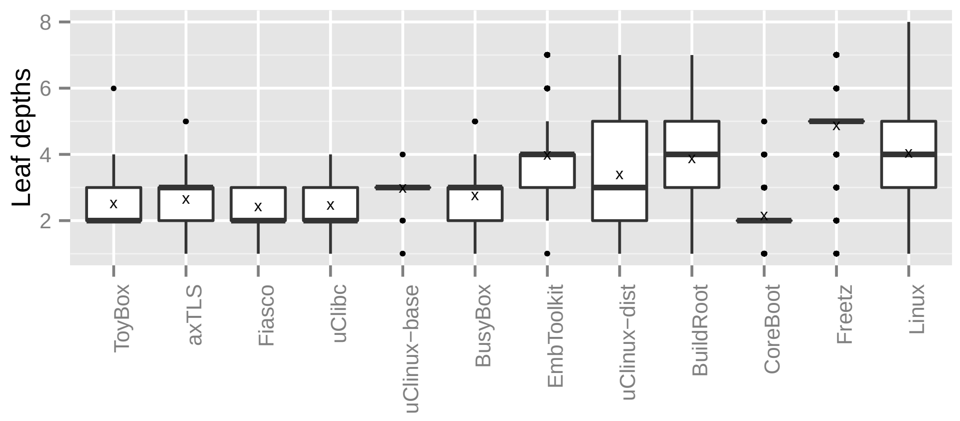


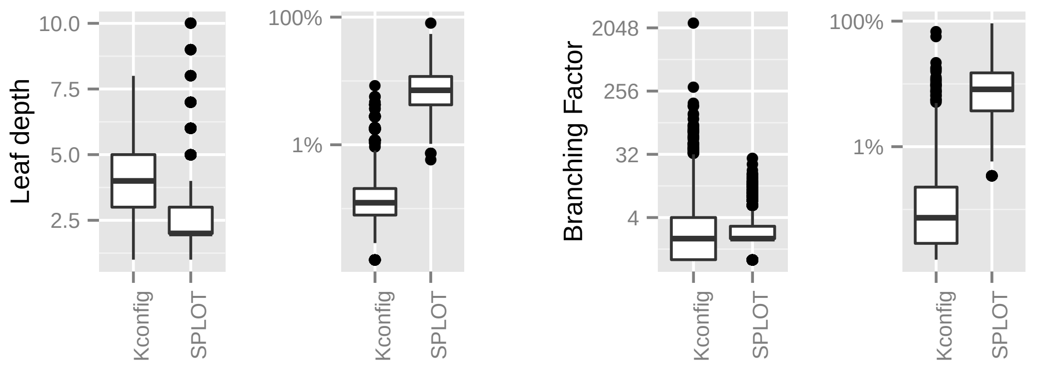
Figure 4.11: Leaf depths for the Kconfig models ordered by model size

menus and choices—features that do not declare a symbol. Grouped features are the immediate configs beneath a choice node. Data features are string, int, and hex configs. Boolean and tristate are the respective config types. Figure 4.10a shows the breakdown of the feature types in the Kconfig models. Linux is the only project that uses tristate features. The majority of uCLinux-base is choice nodes and grouped configs, while ucLinux-dist has mainly Boolean configs. CoreBoot has the largest proportion of data features out of all Kconfig models.

Figure 4.10b shows the growth of the different feature types over 20 versions of the Linux kernel variability model. We used this data for our analysis of the evoluation of the Linux kernel [LSB^{+}10]. Tristate features show the most growth, with boolean features being second. The number of grouping, grouped, and data features stay relatively constant.

Leaf Depth and Branching Factor Figure 4.11 compares the leaf depths of the individual Kconfig models. The models are ordered by size—from smallest to largest where Linux is the largest model. The small 'x' indicates the mean leaf depth for a feature in each model. There appears to be a positive correlation between model size and leaf depth in the Kconfig models.

Figure 4.12a compares the leaf depth of all features in the Kconfig models against all features in the SPLOT models. These plots ignore the individual models and consider all features across all models as a set. The models in Kconfig tend to be deeper than



(a) Leaf depth, absolute and relative to size (b) Branching, absolute and relative to size

Figure 4.12: Structure of the Kconfig and SPLOT models across all features in the dataset

SPLOT models by their absolute leaf depth, with a median leaf depth of 4 and a mean of 3.7. The SPLOT models have a median of 2 and a mean leaf depth of 2.7. However, since the SPLOT models are significantly smaller than Kconfig models, features are typically nested deeper in SPLOT models relative to the size of the models than the Kconfig models.

Figure 4.12b contains two plots of the branching factor for features in the Kconfig and SPLOT models. While the absolute numbers for Kconfig and SPLOT are similar, the plot with the relative branching factor shows a significant difference between the two datasets. Features in SPLOT models tend to have more children relative to their size than features in Kconfig models. There are several outliers in the Kconfig models however; a single feature in the Freetz model has 2377 children with a total model size of 3471.

4.4.4 Feature Groups

Figure 4.13 shows the number of feature groups in the Kconfig models. EmbToolkit has the most groups out of all models, with 111 XOR-groups, while ToyBox contained no feature groups at all. The high number of groups in EmbToolKit could be attributed to the choices for selecting a specific version of software, e.g., version of the gcc compiler to use, specific sub-version of the Linux kernel. Relative to the total number of feature nodes (i.e., configs, menus, and choices), uClinux-base contained the most feature

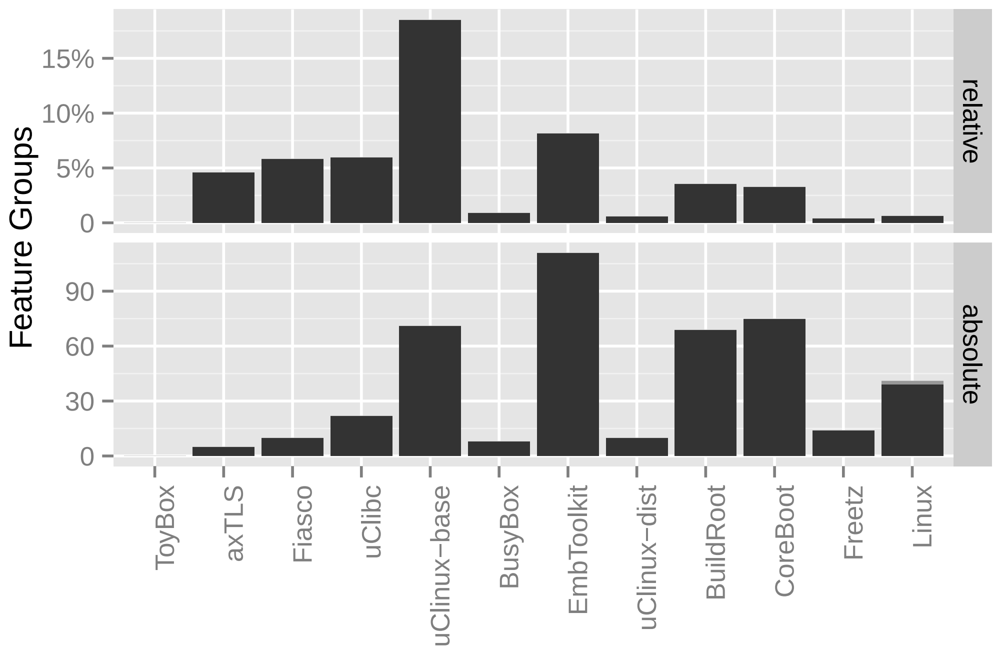
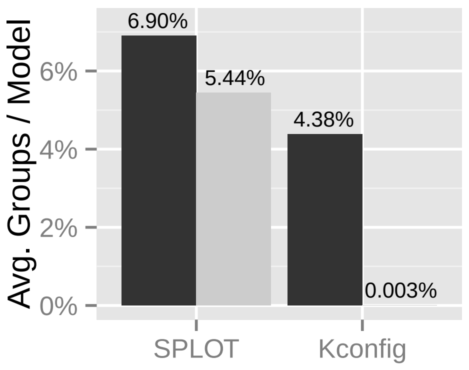
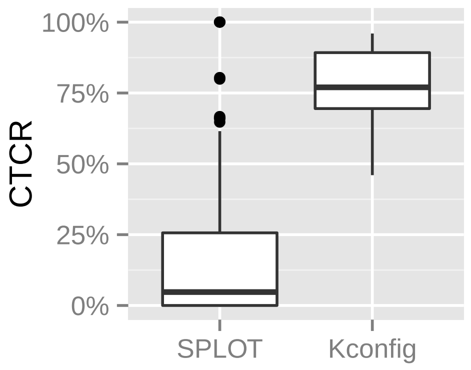


Figure 4.13: Feature groups in Kconfig models. The x-axis is ordered by model size, the black bars are XOR-groups, and the grey bars are tristate groups (Linux). The top plot is the number of groups relative to the number of total features, and the bottom plot is the absolute number of groups.



(a) Average feature groups per model. The black bar is the XOR-groups, and the grey bar is OR-groups.



(b) Cross-tree constraint ratio

Figure 4.14: Feature groups and CTCR across all features in SPLOT and Kconfig models

groups with 19% of nodes being XOR-groups. These feature groups are used to model the choices for specific chipsets from each manufacturer.

Recall from Figure 4.5 that an XOR-group in Kconfig is interpreted as a choice node that is Boolean and does not have the optional keyword. An OR-group is a tristate choice, and a MUTEX-group contains the optional keyword. In the Linux kernel (v2.6.32), we found 39 instances of XOR-groups, 2 instances of tristate groups, and no MUTEX-groups. Linux was the only model with tristate groups since it was the only model that contained any tristate features. There were no occurrences of MUTEX-groups in any of the Kconfig models.

Figure 4.14a compares the average XOR- and OR-groups per model in the SPLOT and Kconfig datasets. Since feature groups are actual nodes in the Kconfig models, we also count the feature groups as nodes in our calculations for this figure. We use a similar computation for SPLOT models where the percentage of feature groups is calculated as:

$$\%\text{ of feature groups } = \frac{\#\text{ of feature groups}}{\#\text{ of features} + \#\text{ of feature groups}} \tag{4.8}$$

We see in the figure that SPLOT models have a higher average proportion of XOR- and OR-groups than Kconfig models. The result with OR-groups is not surprising since only the Linux model had OR-groups, while OR-groups are a common structure in feature modeling.

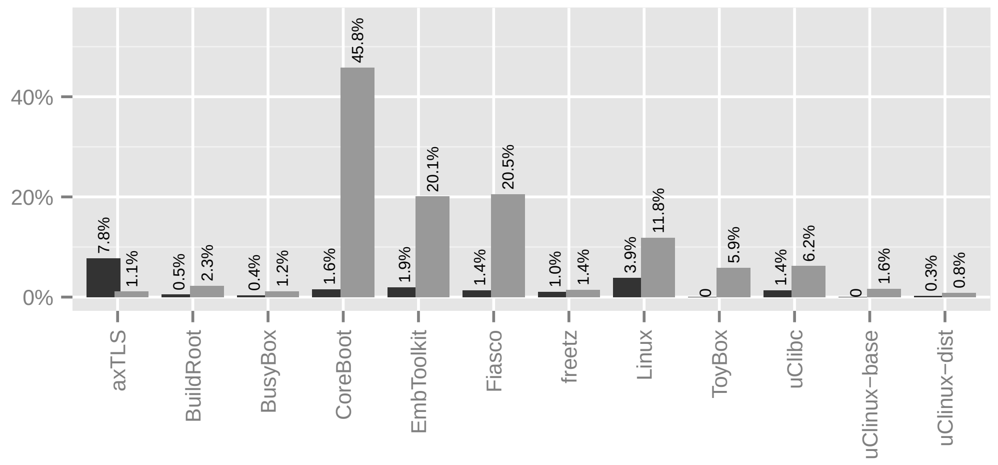
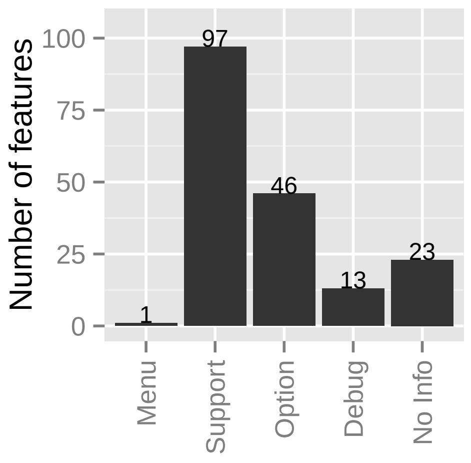


Figure 4.15: Proportion of derived and conditionally derived configs in the Kconfig models. The black bar is for conditionally derived configs, and the grey bar is for derived configs.

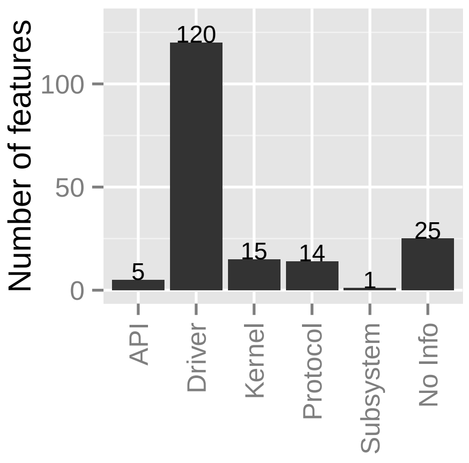
4.4.5 Cross-Tree Constraints

Mendonca et al. found that constraints embedded in a feature model's hierarchy and in feature groups were easy to reason with SAT solvers [MWC09]. However, the addition of cross-tree constraints can cause a model to become difficult for solvers. The *cross-tree constraint ratio (CTCR)* is a measure of the complexity of the cross-tree constraints and is defined as the number of features participating in cross-tree constraints, divided by the number of total features. We adapt the CTCR to Kconfig by taking the number of features participating in some constraint divided by the total number of features. A constraint in Kconfig could be a prompt condition, a default, a select, or a range.

Figure 4.14b compares the CTCRs of the SPLOT models to those of the Kconfig models. The SPLOT models generally have a lower CTCR than Kconfig models, however, there are several outliers with high CTCR. These include the *Dell Laptop / Notebook* feature model with a CTCR of 80%. This model uses cross-tree constraints to restrict options depending on the laptop type (e.g., smaller laptops vs. larger laptops). The models with 100% CTCR contained a trivial cross-tree constraint that enumerated all features, including the root feature, as a disjunction. These constraints may be used as an implementation artifact for a reasoner.



(a) User-based granularity categories of Linux features



(b) Feature types in the Linux kernel

Figure 4.16: Characterization of a small sample of features in the Linux v2.6.28.6 kernel

The higher CTCR in Kconfig models is partially due to Kconfig having more language constructs that contribute some form of constraints. For example, Kconfig supports defaults that do not impose a constraint on the configuration space, but instead provide suggested values. We count these defaults when calculating the CTCR for the Kconfig models.

In Section 4.3.3, we categorized configs based on their cross-tree constraints. We found in v2.6.32 of Linux that 12% of configs were unconditionally derived, 4% of configs were conditionally derived, and the remainder were configured configs. Figure 4.15 show the percentage of configs that are derived and conditionally derived in the Kconfig models respectively. The remainder of the configs not shown in the figure are user configured.

4.4.6 Qualitative Characteristics

We performed a manual categorization of 180 randomly selected features to identify the granularity and types of features in the Linux variability model [LSB+10]. The selected categories characterize the granularity of features from the user's perspective—whether a feature enables support for a device, or whether the feature is related to another feature, e.g., related to a driver, protocol, or API.

We categorized features based on the feature descriptions and by querying the web when needed. We left features with no description in Kconfig uncategorized. Since the classification is subjective, we cross checked the categories for 18 features with another author of the paper. We found a discrepancy of only 8% and so believe that the categorization is sound and relevant. We used the following user-based granularity categories, from the most course-grained to the most fine-grained:

Menus Features used to group subfeatures, e.g., IO_SCHEDULERS, which groups read / write schedulers for block devices;

Support Features that enable support certain devices or protocols, e.g., HID_-SAMSUNG, which enables support for Samsung's InfraRed remote control;

Option Features enabling or disabling an ability in another feature, e.g., DASD, which enables support for direct access storage devices in block devices;

Debug Developer-related features, e.g., BOOT_TRACER, which activates run-time collection to assist in boot-time optimizations, or MEMSTICK_DEBUG, which activates additional debugging information for memory stick devices.

The results of the user-based granularity categorization is shown in Figure 4.16a. Out of the 180 sampled features, 97 of those features were the relatively coarse-grained support features. There was only a single menu in our sample. We were unable to categorize 23 features since they were missing feature descriptions. We additionally categorized the sample into the following feature types:

API features that provide an external programming interface, e.g., CRYPTO_CTR, which enables the API for a block cipher algorithm;

Driver features that enable support for a device, e.g., SND_ADLIB, which enables support for AdLib audio cards;

Kernel features that add or remove an low level ability to the kernel, e.g., FAIL-SLAB, which enables fault-injection capability for kmalloc—a memory allocator for the kernel;

Protocol features that implement a protocol, e.g., LLC2, which enables support for PF_LLC sockets;

Subsystem features that enable support for a whole subsystem, e.g., BT, which enables the Bluetooth subsystem.

Kconfig	Boolean	
	x_1	x_2
n	0	0
m	1	0
y	1	1

Figure 4.17: Representation of Kconfig's three values using two Boolean variables

The majority of features in our sample were drivers. We were unable to identify the type for 25 features.

4.5 Tooling: Linux Variability Analysis Tools

We developed the *Linux Variability Analysis Tools* (LVAT)[4] to extract and analyze Kconfig models. LVAT uses a component written using the Kconfig infrastructure in the Linux kernel to parse the concrete Kconfig syntax and output an intermediate format as a protocol buffer[5], or as an easier-to-parse plain text file. We use LVAT to gather the statistics presented in this chapter.

The analysis portion of LVAT parses the intermediate format to an AST that matches the concrete syntax of Kconfig. The AST could then be transformed to the abstract syntax of Kconfig described in Section 4.2.

One of the key features of LVAT is the translation of a Kconfig model to a propositional formula based on the formal semantics in Appendix A. Since a config supports three values, the translation uses two Boolean variables per config. For a config x, We show the representation of the three Kconfig values in Figure 4.17 using the Boolean variables x_1, and x_2. We disallow the case $\{x_1 \mapsto 0, x_2 \mapsto 1\}$ by adding this constraint to the formula. The current translation doe not support string, int, and hex configs and assumes any constraint involving these configs are satisfied.

We use the propositional translation for the statistics dependent on dependencies and cross-tree constraints in the chapter, e.g., identifying configs that violate hierarchy rules. The propositional formula has also been used in other tools, such as TypeChef—a parser for programs written with the C preprocessor [Käs10]. TypeChef uses the

[4]http://linux-variability-analysis-tools.googlecode.com
[5]http://protobuf.googlecode.com

propositional translation of a Kconfig model to identify errors in #ifdef conditions and prune unreachable alternatives from its parse tree.

4.6 Kconfig Semantics

We also reverse engineered formal semantics in denotational style [Sch86] for the Kconfig language. We based these semantics first on the Kconfig specification [Zc], then when in doubt, on how the graphical *xconfig* configurator performed, and also examined the source code when needed. The semantics are available in Appendix A.

Chapter 5

Feature Graph Extraction

In this chapter, we introduce the first of our feature model synthesis algorithms called FEATURE-GRAPH-EXTRACTION (FGE). Given a set of features and a propositional formula, FGE is an automated algorithm that recovers a feature graph describing all feature diagrams that is entailed by the input. This algorithm recovers all components of a feature model: its possible hierarchies, feature cardinalities (i.e., optional and mandatory features), and possible feature groups. We describe two variants of FGE that operate on propositional formulas in conjunctive normal form (CNF) and disjunctive normal form (DNF) called FGE-CNF and FGE-DNF respectively.

The recovered feature graph can be used as an intermediate format for other synthesis algorithms that derive feature models. For example, the feature graph can be used as part of an interactive model builder [JKW08], or as part of a feature model merge operation [Ach11]. We discussed several scenarios involving FGE in Chapter 3. In Chapter 6, we present our own approach for selecting a distinct feature model by applying a textual similarity heuristic on feature names and descriptions.

We performed two experiments to evaluate the scalability of FGE-CNF and FGE-DNF against existing feature model synthesis algorithms with input representative of that used in practical synthesis scenarios. For FGE-CNF, we compare it to the BDD-based algorithm introduced by Czarnecki and Wąsowski [CW07] that we call FGE-BDD. FGE-CNF and FGE-BDD both take the same input and output the same feature graph. For FGE-DNF, we compare it with the formal concept analysis-based algorithm by Ryssel et al. [RPK11] that we present as FGE-FCA in this chapter. FGE-FCA takes input in a similar form as FGE-DNF and outputs the same feature graph. We show that FGE-CNF is significantly faster than FGE-BDD, while FGE-DNF performs comparably with FGE-FCA, except in five cases, where FGE-DNF was faster.

Chapter Organization We begin this chapter by summarizing three feature model synthesis scenarios that FEATURE-GRAPH-EXTRACTION addresses in Section 5.1. Section 5.2 defines the feature model synthesis and the feature graph synthesis problem. We describe the generic FGE algorithm in Section 5.3. Sections 5.4 and 5.5 describes FGE-CNF and FGE-DNF respectively. Next, we describe the two algorithms that we use in our experimental evaluation according to the FGE algorithm. In Section 5.6, we describe the BDD-based algorithm by Czarnecki and Wąsowski and in Section 5.7, we describe the formal concept analysis-based algorithm by Ryssel et al. [RPK11]. We describe our experimental evaluation of FGE-CNF against baseline implementations of the BDD-based algorithm and a formal concept analysis (FCA)-based algorithm in Section 5.9.

Publications The FGE and the SAT-based variants that operate on CNF and DNF input was published in [ACSW12]. We submitted an extended journal version of the paper [SRA+13] with a more extensive evaluation comparing our algorithm with the FCA-based algorithm by Ryssel et al. [RPK11].

External Contributions FGE-BDD is based on the BDD synthesis algorithm by published by Czarnecki and Wąsowski [CW07]. FGE-CNF and FGE-DNF were developed and implemented with Andersen and the algorithms and the initial evaluation appear in [ACSW12] and her Master's thesis [And09]. We expand on the evaluation to include a larger dataset with the Linux variability model and 267 SPLOT feature models. We also conducted an evaluation comparing FGE-DNF with FGE-FCA, an adaptation of the synthesis algorithm by Ryssel et al. [RPK11] to the FGE framework. This section was written by Ryssel for [SRA+13] and adapted for this thesis.

5.1 Motivation and Scenarios

We use three synthesis scenarios from Chapter 3 to motivate our work on the FEATURE-GRAPH-EXTRACTION algorithm. We summarize the scenarios below.

Scenario 1: Tool-assisted feature model reverse engineering This scenario described reverse engineering a feature model from code. Variability rich software, such as the FreeBSD kernel, can benefit from feature model. The FreeBSD operating system kernel is configured prior to compilation to derive variations of the kernel functionality. Unlike the Linux kernel [BSL+10b], the FreeBSD kernel does not have a feature model that makes configuration easier for users, and variability management easier for developers.

In this scenario, the source code contains variability. This variability could be as modules, or annotative with C preprocessor `#ifdef` statement in FreeBSD. A variability analysis, such as a static code analysis, can uncover variation points (VPs) and dependencies between these. The resulting VPs and dependencies are used as input to the feature model synthesis algorithm. Dependencies extracted from source code can be translated to a CNF formula, and FGE-CNF can be used to identify all possible feature diagrams for the input. FEATURE-GRAPH-EXTRACTION along with FEATURE-TREE-SYNTHESIS (Chapter 6) provide a scalable feature model synthesis infrastructure capable of synthesizing large feature models with several thousand features.

Scenario 2: Feature model synthesis from product configurations The next scenario involves synthesizing a feature model from a set of variants that describe a product line. These variants could include code developed with a clone-and-own approach [JDB07] or as individual model variants. Examples of variant analysis for this scenario include Rubin and Chechik's technique for identifying similarities between model instances by comparing and matching model instances [RC12]. Ryssel et al. also described an algorithm that used model matching and difference to identify VPs and abstract the model variants into a set of VP configurations [RPK12]. In this scenario, features and feature configurations are used as input to FEATURE-GRAPH-EXTRACTION. Feature configurations are naturally represented as a DNF formula making FGE-DNF suitable for this scenario.

Scenario 3: Feature model merge operations Our third scenario involves feature model merge operations as described by Acher [Ach11]. A merge creates a new feature model that describes the intersection or union of configurations of two or more feature models. The input models are translated to their propositional formulas [Bat05], then a propositional merge is performed. The exact configuration semantics depends on the selected kind of merge. For the hierarchy, Acher applies heuristics for selecting the hierarchy based on the structures of the input models. These heuristics enable the automatic selection of a feature tree without further user input. Acher's feature model management infrastructure [Ach11] implements the operations using our previous BDD-based FM synthesis solution [CW07], which does not scale beyond small FMs, with few dozens of features. The FEATURE-GRAPH-EXTRACTION algorithm presented here can be used to improve the scalability of that infrastructure. The FGE-CNF algorithm in particular, is a perfect fit for this scenario since the configuration semantics of feature models translate naturally to a propositional formula in CNF.

5.2 Defining the Feature Model and Feature Graph Synthesis Problem

Before we discuss our synthesis algorithm, we must first define the problem that our algorithm is addressing. Given a set of features ($\mathcal{F}$) and a propositional formula (φ), the feature model synthesis problem is defined as synthesizing a feature diagram FD such that the input formula φ entails the configuration semantics of the feature diagram. We now define the problem using the definition of a feature diagram (Definition 2.1), where $\mathsf{FD} = (\mathcal{F}, E, (E_m, E_i, E_x), (G_o, G_x, G_m))$:

Definition 5.1. The *feature model synthesis problem (FMS)* is defined as given a consistent rooted formula φ over a set of feature $\mathcal{F}$, synthesize a feature diagram FD over $\mathcal{F}$, such that $\varphi \rightarrow [\![\mathsf{FD}]\!]$, and FD is *maximal* such that (i) no element can be added to the set of mandatory edges (E_m), implies edges (E_i), or excludes edges (E_x), OR-, MUTEX-, and XOR-groups (G_o, G_m, G_x) and (ii) no group can be moved from $G_o \cup G_m$ to G_x without violating $\varphi \rightarrow [\![\mathsf{FD}]\!]$.

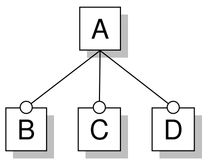
Note that our problem definition includes a condition such that the synthesized diagram is *maximal*. This condition is necessary to avoid synthesizing trivial models that are not useful in any synthesis scenario. For example, a feature diagram where all features are nested under the root is consistent with any rooted formula such as the one in Figure 5.1b. The feature diagram in Figure 5.1b is non-maximal with respect to the input in Figure 5.1a because it is missing an OR-group and implies edges from D to B and C. In an automated synthesis algorithm, *maximal* feature diagrams are more useful, since they contain the maximal constraints that are allowed by the feature modeling notation.

Theorem 5.1. The decision version of FMS is NP-hard.

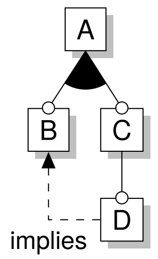
Proof sketch. We define the decision version of FMS as: for a given consistent rooted

$$\begin{aligned}(B \vee C \vee D \rightarrow A)\\ \wedge(D \rightarrow B \wedge C)\\ \wedge(A \rightarrow B \vee C)\end{aligned}$$

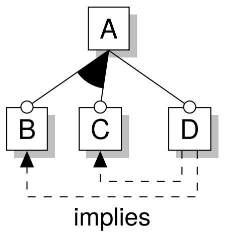
(a) Input



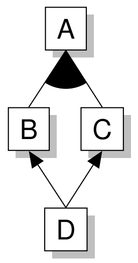
(b) A non-maximal feature diagram



(c) A maximal feature diagram



(d) Another maximal feature diagram



(e) Feature graph

Figure 5.1: Maximal feature diagrams and a feature graph

formula φ over the set of features F, is a given feature diagram FD over F actually maximal such that $\varphi \rightarrow [\![\mathsf{FD}]\!]$?

Iff the answer is *yes*, then for any two variables in F, there is no other binary implications entailed by φ than those that are edges in D. If such implications did exist, then the model would not be maximal. Thus, for an implication $x \rightarrow y$, we check if $\psi \land x \land \neg y$ is satisfiable (disproves $x \rightarrow y$) for an arbitrary consistent rooted formula ψ. Such a SAT check is NP-hard. □

The proof reduces the decision version of FMS to checking satisfiability of a propositional formula or the equality of two propositional formulas—both NP-hard problems.

Even with the concept of maximality, there could be more than one maximal feature diagram describing a given input. Figures 5.1c and 5.1d are two maximal feature diagrams. Figure 5.1c is maximal since all possible feature groups and implies edges are present. In Figure 5.1d, even though D could be a child of C, the diagram is still maximal since an implies edge takes the place of the hierarchy edge. The definition of maximality in Definition 5.1 does not require that each feature be as deeply nested as possible.

As we see from Figures 5.1c and 5.1d, there could still be more than one maximal feature diagram describing a given set of configurations or dependencies. We introduce the notion of a *feature graph* that encapsulates all maximal feature diagrams as a single structure, defined as:

Definition 5.2. A *feature graph* is a tuple $\mathsf{FG}(\mathcal{F}, E, E_{\mathrm{x}}, (G_o, G_x, G_m))$ where $\mathcal{F}$ is a set of features, $E \subseteq \mathcal{F} \times \mathcal{F}$ is a set of directed child-parent edges; $E_{\mathrm{x}} \subseteq 2^{\mathcal{F}}$ is a set of undirected *excludes* edges, for each $e \in E_{\mathrm{x}}$, $|e| = 2$; sets G_o, G_x, G_m contain subsets of E, participating in OR-groups, XOR-group and MUTEX-groups respectively. The following constraints hold in FG:

1. (F, E) is a connected transitively reduced DAG,
2. All edges in a group share the same parent, so if $g \in G_i$ for $i \in \{\mathrm{o,x,m}\}$ and $(f_1, f_2), (f_3, f_4) \in g$ then $f_2 = f_4$,
3. E, E_{x} are disjoint (i.e., no hierarchy edge is an exclude edge).

Compared to the definition of a feature diagram (Definition 2.1), a feature graph relaxes the constraints on the feature hierarchy such that a feature graph forms a DAG instead of a tree, and feature groups can overlap. The definition of feature diagram allows feature groups with $\langle m, n \rangle$ cardinality; however, we restrict the definition of a feature graph to only OR- $\langle 0, n \rangle$, XOR- $\langle 1, n \rangle$, and MUTEX-groups $\langle 0, 1 \rangle$ since only these cardinalities are supported by our synthesis algorithm. We also seen in our study of real world feature models (Chapter 4) that most feature groups in practice were XOR- or OR-groups. Compared to feature diagrams, implies edges are also missing from a feature graph—these edges are represented as part of the (F, E) DAG. Finally, mandatory features are absent in a feature graph since mandatory features co-occur with their parent in all legal configurations. Mandatory features are represented as a single node (i.e., an AND-group) in the feature graph. Figure 5.1e shows the feature graph for the input in Figure 5.1a.

Since feature graphs encapsulate all consistent maximal feature diagrams with respect to a set of input features and legal configurations, it provides a convenient target artifact for automated synthesis algorithms. We define the feature graph synthesis problem as:

> **Definition 5.3.** The *feature graph synthesis problem* is defined as given a consistent rooted formula φ over a set of feature $\mathcal{F}$, synthesize a feature graph FG over $\mathcal{F}$, such that $\varphi \rightarrow [\![\mathsf{FG}]\!]$, and given a FG, (i) no element can be added to the set of child-parent edges (E), excludes edges (E_x), OR-, MUTEX-, and XOR-groups (G_o, G_m, G_x) and (ii) no group can be moved from $G_o \cup G_m$ to G_x, without violating $\varphi \rightarrow [\![\mathsf{FG}]\!]$.

The problem definition assumes that the input is given as a set of dependencies or configurations expressed as a rooted formula φ over a set of features $\mathcal{F}$. If a root is not present in the formula, a new variable r' and implications $f_i \in \mathcal{F}, f_i \rightarrow r'$ can be introduced to φ. All possible implications from the input are also contained the DAG formed by the features and edges $(\mathcal{F}, E)$. Furthermore, all possible mandatory, implies, excludes edges, OR-, XOR-, and MUTEX-groups XOR are captured in the resulting feature graph. Finally, the last condition is that the set of XOR-groups is maximal.

5.3 Fge: Feature Graph Extraction Algorithm

We now introduce the pseudocode for the generic FEATURE-GRAPH-EXTRACTION (FGE) in Figure 5.2. Given a consistent, rooted propositional formula φ and a set of features

```
FEATURE-GRAPH-EXTRACTION ($\varphi$ : formula over $F$ rooted in $r$, $r \in F$)
       ▷ Find and remove all dead features
 1     $D = \{f \in F \mid \varphi \wedge r \rightarrow \neg f\}$
 2     $\varphi = \varphi[d \mapsto 0]_{d \in D}$

       ▷ Compute the implication graph $G(V, E)$
 3     $V = F \setminus D$
 4     $E = \{(u, v) \in V \times V \mid \varphi \wedge u \rightarrow v\}$

       ▷ Compute strongly connected components
 5     $V' = \{S \subseteq V \mid S$ is a SCC of $G\}$

       ▷ Make edges between SCCs creating a DAG
 6     $E' = \{(u, v) \in V' \times V' \mid u \neq v$ and
 7                              $\exists u' \in u, v' \in v.\ (u', v') \in E\}$

       ▷ Compute the mutex graph $M(V, E_x)$
 8     $E_x = \{\{u, v\} \subseteq V' \mid \exists u' \in u, v' \in v. \varphi \wedge u' \rightarrow \neg v'\}$

       ▷ Compute mutex-groups
 9     $G_m = \{\{(f_1, p), \ldots, (f_k, p)\} \mid \{f_1, \ldots, f_k\}$ is
10                    a maximal clique in $M$ and $\forall f_i. (f_i, p) \in E'\}$

       ▷ Compute or-groups
11     $G_o = \{\{(f_1, p), \ldots, (f_k, p)\} \mid f'_1 \vee \cdots \vee f'_k$ is a prime implicate of $\varphi \wedge p'$ where
12                       $p' \in p$ and
13                       $\forall i \in \{1, \ldots, k\}.\ f'_i \in f_i \wedge (f_i, p) \in E'\}$

       ▷ Compute xor-groups
14     $G_x = G_o \cap G_m$

15     return FG$(V', E', E_x, (G_o \setminus G_x, G_x, G_m \setminus G_x))$
```

Figure 5.2: Generic feature graph extraction algorithm, adapted from [ACSW12, CW07]

F, FGE constructs a feature graph as defined in Definition 5.2. This algorithm was first introduced by Czarnecki and Wąsowski and assumed the use of binary decision diagrams (BDDs) for reasoning [CW07]. Our contribution is the adaptation of this algorithm to support reasoning with a SAT solver on input in conjunctive normal form (Section 5.4) and disjunction normal form (Section 5.5). We begin by describing each step of the generic FGE algorithm:

Lines 1–2, Dead Features: The first step is the removal of dead features (Lines 1–2). A dead feature in φ is one that is *false* in all satisfying assignments of φ. Dead features do not contribute to the set of legal configurations and can always be added back to the model after synthesis.

Lines 3–4, Implications: The second step is the construction of the implication graph, defined below:

> **Definition 5.4.** The *implication graph* (V, E) of a formula φ is a directed graph where the vertices V are features and an edge $(v_1, v_2) \in E$ exists if $\varphi \wedge v_1 \rightarrow v_2$.

Line 5, And Groups: AND-groups are features that always co-occur together in any satisfying assignment. We find AND-groups by identifying strongly connected components (SCCs) in the implication graph. A SCCs is a group of vertices such that there exists a path from each vertex to all other vertices in the group. We lift the implication graph to sets of vertices, such that each SCC becomes a single vertex. (V', E') is an implication graph such that V' is a set of features from V, and an edge $(u, v) \in E'$ exist if any feature in u implies a feature in v given φ. The resulting graph forms a DAG since implications are transitive and any cycle between features in the original implication graph is contained in a SCC. Note that from this line onward, a vertex contains a set of features.

Line 8, Mutual Exclusions: We now move to identifying mutual exclusions by computing the mutex graph, defined as:

> **Definition 5.5.** The *mutex graph* (V, E) of a formula φ is an undirected graph where vertices V are features and an edge $(u, v) \in E$ exists if $\varphi \wedge u \rightarrow \neg v$.

The mutex graph uses the SCCs as vertices and an edge (i.e., mutual exclusion) (u, v) exists between two vertices if any feature in u excludes a feature in v.

Line 9, Mutex Groups: We use the mutex graph to compute MUTEX-groups by identifying all maximal cliques in M. Each clique becomes the members of the MUTEX-group and the common ancestor in the implication graph (V', E') becomes group's parent p.

Lines 11–12, Or Groups: We now finds OR-groups by computing prime implicates for each variable p'. An *implicate* D of a propositional formula φ is a clause such that (i) D itself is not a tautology, and (ii) $\varphi \to D$ is a tautology [ACSW12]. D is a *prime implicate* iff it is minimal such that no literal can be removed from D without violating (ii).

Line 14, Xor Groups: Having computed both MUTEX-, and OR-groups in the previous steps, XOR-groups are simply the intersection of both OR-groups and MUTEX-groups. However, computing OR-groups is the most computationally complex step of the synthesis algorithm [ACSW12, SRA+13]. All other steps that involve reasoning on φ reduce to computing binary implications. We define an alternative XOR-group computation that does not depend on OR-groups next.

Alternative Xor Group Computation: An alternative method of computing XOR-groups is to check for each MUTEX-group, given the group parent, if at least one group member must be present in φ. This is equivalent to checking if $\varphi \wedge p \wedge \neg f_1 \wedge \cdots \wedge \neg f_k$ is inconsistent. Using this computation, the set of XOR-groups is alternatively defined as:

$$G'_x = \Big\{\{(f_1, p), \ldots, (f_k, p)\} \in G_m \mid \varphi \wedge p' \to f'_1 \vee \cdots \vee f'_k \\ \text{where } p' \in p \text{ and } \forall i \in \{1, \ldots, k\}, f'_i \in f_i\Big\} \quad (5.1)$$

Consistency Between the Input φ and the Produced Feature Graph FGE produces a feature graph that is weaker than φ, i.e., $\varphi \to [\![\mathsf{FG}]\!]$ given a FG produced by FGE. We define the following theorem:

Theorem 5.2. Given an input formula φ with no dead variables and equivalent variables (i.e., variables that would participate in an AND-group), FGE produces a feature graph FG that is weaker than φ in the sense that $\varphi \to [\![\mathsf{FG}]\!]$.

Proof. We assume for this proof that there are no dead variables and no equivalent variables in the input formula φ. If there exists any dead or equivalent variables, then FGE produces an isomorphic set of satisfying assignments. We prove that every element of a feature graph $\mathsf{FG} = (V', E', E_x, (G_o, G_x, G_m))$ is weaker than φ.

- The vertices of a feature graph, V', consists of the set of variables in φ (Lines 3 and 5).
- Since there are no dead or equivalent variables in φ, the set of edges, E' consists of edges from E with edges between variables in φ by construction (Line 6). It is clear that each edge $(u, v) \in E$ satisfies $\varphi \to (u \to v)$ (Line 4).
- Similarly, for the set of excludes edges E_x, $\varphi \to (u \to \neg v)$ for each undirected edge $\{u, v\} \in E_x$ by construction (Line 8).
- For each OR-group $\{(f_1, p), \ldots, (f_k, p)\} \in G_o$, $f_1' \vee \ldots \vee f_k'$ is a prime implicante of $\varphi \wedge p'$ where $\forall_{i \in 1..k} f_i' \in f_i$ (Line 11). This can be rewritten as $\varphi \to (p' \to f_1' \vee \ldots \vee f_k')$.
- For each MUTEX-group $\{(f_1, p, \ldots, (f_k, p)\} \in G_x$, $\forall_{i,j \in \{f_1, \ldots f_k\}; f_i \neq f_j} \varphi \to (f_i \to \neg f_j)$ holds since each MUTEX-group is a clique in the mutex graph that consists of excludes edges from E_x (Line 9). We showed earlier than the set of excludes edges is entailed by φ.
- Finally, XOR-groups consists of feature groups that are both OR- and MUTEX-groups (Line 14). We proved earlier that both the set of OR-groups and MUTEX-groups are entailed by φ.

Since φ entails the semantics of every element of the feature graph FG, then $\varphi \to [\![\mathsf{FG}]\!]$. □

Furthermore, the feature graph produced by FGE is a maximal, transitively closed feature graph. We define a maximal feature graph as the following:

Definition 5.6. A feature graph FG over F is maximal such that (i) no element can be added to the sets of child-parent edges E, excludes edges E_m, and OR-, XOR-, and MUTEX- groups (G_o, G_x, G_m) without violating $\varphi \to [\![\mathsf{FG}]\!]$, and (ii) no group can be moved from $G_o \cup G_m$ to G_x without violating the above implication.

Note that the above definition of maximality requires that the child-parent edges be transitively closed by condition (i). We can now state the following theorem:

Theorem 5.3. FGE produces a maximal feature graph.

Proof sketch. We sketch a proof that the produced feature graph is maximal by a contradiction argument. Assume that the feature graph produced by FGE lacks an edge (u, v) in the set of child-parent edges. We know that φ cannot entail $u \to v$, otherwise the edge would have already been in the set of child-parent edges by Line 4. Thus, any feature graph containing more edges must not be entailed by φ. Such a feature graph would be unsound, contradicting condition (i) in Definition 5.6.

The argument for the remaining components of the feature graph showing that all sets created by FGE are maximal is structured similarly. □

5.4 Fge-CNF: CNF Formula as Input

We now describe the first implementation-specific variant of the FGE. FGE-CNF assumes that φ is a formula in conjunctive normal form (CNF). A formula in CNF consists of a conjunction of clauses. A *clause* is a disjunction of literals. FGE-CNF is effective at handling input in the form of dependencies. For example, FGE-CNF is ideal for Scenarios 1 and 3 that has variable artifacts as input and provide dependencies that can be naturally represented as a CNF formula. We describe FGE-CNF next based on to the generic algorithm structure in Figure 5.2. FGE-CNF assumes that φ is a formula in CNF and uses a SAT solver as its reasoner.

Lines 1–2, Dead Features: In FGE-CNF, we detected if a feature f is dead by checking if $\varphi \wedge f$ is consistent. $\varphi \wedge f$ is a CNF formula, a single SAT call can determine consistency. If consistent, we can use the model solution provided by the SAT solver to provide liveness (i.e., not dead) of variables assigned *true*. No further SAT calls are needed for these variables proved live. In the worst-case, detecting dead features performs $O(|\mathcal{F}|)$ SAT calls.

Lines 3–4, Implications: A binary implication between two variables f_i and f_j, is detected by proving the consistency of formula of $\varphi \wedge f_i \to f_j$, or, equivalently, checking if $\varphi \wedge f_i \wedge \neg f_j$ is inconsistent. Thus, one implication edge is detected by one SAT call. Detecting all implications requires $O(|\mathcal{F}|^2)$ calls. In practice, like detecting dead features, a consistent model can be used to disprove all implications between variables f_l and f_k, whenever f_l is assigned *true* and f_k is assigned *false* in the model.

Line 8, Mutual Exclusions: Detecting mutual exclusions is done by checking if $\varphi \wedge f_i \wedge f_j$ is inconsistent. Like above, finding all exclusions requires at most $O(|\mathcal{F}|^2)$ SAT checks.

The number of SAT checks can also be decreased by learning about more than one pair of features from a consistent model.

Lines 11–12, Or Groups: To identify OR-groups we need to find prime implicates of $\varphi \wedge p$ for each $p \in \mathcal{F}$. We use the PIG algorithm [JP90, Jac92] to identify prime implicates. Applying PIG to $\varphi \wedge p$ will identify all prime implicates in the formula assuming the parent p; however, not all are valid OR-groups. We are only interested in implicates that consists of variables that imply the parent p in the implication graph since feature groups can only be between a parent and its children, and implicates that consists of only positive literals.

We can rewrite the formula $\varphi \wedge p$ to eliminate unneeded variables by applying the VER algorithm [SP04]. Given a formula φ, the output of $\text{VER}(\varphi, x)$ is a CNF formula ψ not containing the variable x, but is *equisatisfiable* to φ—ψ is satisfiable whenever φ is satisfiable. In addition to ψ being equisatisfiable to φ, the prime implicates of ψ are the same as the prime implicates of φ over the set of kept variables.

Lines 11–12, Incremental Or Group Computation: The approach outlined above performs redundant work across each $p \in \mathcal{F}$. Given variable p, prime implicates of $\varphi \wedge p$ are found, followed by prime implicates of $\varphi \wedge p'$ for a different variable p'. We can reduce redundant work by first computing the prime implicates of the formula φ, then reuse the results to compute $\varphi \wedge f$ and $\varphi \wedge f'$. [ACSW12, SRA+13] describes the algorithm for computing OR-groups incrementally in detail. The process is inspired by the PIGLET algorithm that computes prime implicates of $\varphi \wedge \psi$ assuming the prime implicates of φ are known [Jac92].

Line 14, Xor Groups: XOR-groups could be computed just as in Figure 5.2 by checking set intersection between OR- and MUTEX-groups. For the alternative XOR-group computation, the satisfiability check on a formula $\varphi \wedge p \wedge \neg f_1 \wedge \cdots \wedge \neg f_k$ is naturally performed by a single SAT call. This check is performed on each MUTEX-group, making the alternative XOR-group computation require $O(|G_m|)$ calls to the SAT solver.

5.5 Fge-DNF: DNF Formula as Input

FGE-DNF assumes that the input, φ, is a formula in *disjunctive normal form* (DNF). A formula is in DNF iff it is a disjunction of terms where a *term* is a conjunction of literals. FGE-DNF is a variant of FGE assuming a DNF formula as input. FGE-DNF is applicable to

Scenario 2 in Chapter 3 where a FM is to be synthesized from a list of existing variants of a product.

We describe FGE-DNF according to the pseudocode in Figure 5.2. When describing runtime, we assume that the DNF formula only contains satisfiable terms. A term is satisfiable if it does not contain a literal and its negation. Unsatisfiable terms can be removed in linear time. $|\varphi|$ is the size of the input and refers to the number of terms in the DNF formula.

Lines 1–2, Dead Features: A variable f is dead iff it is negated in every term of φ. This can be checked in linear time in $|\varphi|$ making this step run in $O(|\varphi||F|)$ time.

Lines 3–4, Implications: Implications are detected by checking if $\varphi \land f_i \rightarrow f_j$ is consistent. Since φ is in DNF, these implications are found by checking if every term in φ contains either $\neg f_i$ or f_j. Detecting all implications takes $O(|\varphi||\mathcal{F}|^2)$ time.

Line 8, Mutual Exclusions: Mutual exclusions are found by checking if $\varphi \land f_i \rightarrow \neg f_j$, or equivalently, $\varphi \land f_i \land f_j$ is consistent. Similar to detecting implications, the satisfiability can be computed in linear time. This process is repeated for every feature pair, so detection of exclusions also takes $O(|\varphi||\mathcal{F}|^2)$ time.

Line 11–12, Or Groups: Identifying OR-groups involves identifying prime implicates of φ. Detecting prime implicates in φ is equivalent to detecting prime implicants of $\neg\varphi$. An *implicant* C of a propositional formula φ is a term such that C is consistent and $C \rightarrow \varphi$ is a tautology. C is a *prime implicant* if it is minimal. If π is a prime implicates of φ, then $\neg\pi$ is a prime implicant of $\neg\varphi$. Using this property, OR-groups can be computed by using a SAT solver to find prime implicants of $\neg\varphi$.

We use a procedure based on *Binary Integer Programming (BIP)* [Sil97, MFSO97], a special case of Integer Linear Programming (ILP) that assumes binary domain for variables (i.e., a variable can be 0 or 1), to compute OR-groups. First, given a feature p, we construct a new formula $\neg\varphi'_p$ with the desired properties to identify OR-groups with parent p:

1. First, negate φ to create $\neg\varphi$. Since φ is in DNF, $\neg\varphi$ becomes a CNF formula.
2. Detecting OR-groups for parent p involves detecting prime implicates of $\varphi \land p$. Since we are working with $\neg\varphi$, we detect prime implicants of $\neg\varphi \lor \neg p$. We retain only the clauses in $\neg\varphi$ that contain $\neg p$.
3. The members of a valid OR-group must share a common parent, namely p. Using the implication graph (V', E'), retain only variables that imply the node containing

p. In other words, retain $\{f \mid f \in v' \wedge (v', p') \in E'$ where $v' \in V'$ and $\exists p' \in V.p \in p'\}$.

4. A valid OR-group has the constraint $p \rightarrow f_1 \vee \cdots \vee f_k$ where the variables $\{f_1, \ldots, f_k\}$ are positive. Since we are working with $\neg\varphi$, this constraints translates to prime implicants containing only negative literals. For each clause C, remove all variables $v \in C$ that are positive.

Now we can use the formula $\neg\varphi'_p$ to build the BIP problem used to identify OR-groups for a parent p:

1. Let L be the set of literals in $\neg\varphi'_p$. For each literal $l \in L$, introduce a Boolean variable x_l to the BIP problem. Note that $\neg\varphi'$ only contains negative literals.

2. For each clause $l_1 \vee \cdots \vee l_m$ in $\neg\varphi'_p$ add the linear inequality $x_{l_1} + \cdots + x_{l_m} \geq 1$ to the set of constraints.

3. Since a literal l and its negation $\neg l$ cannot both be *true* in a satisfying assignment, a constraint of the form $x_l + x_{\neg l} \leq 1$ is added for each literal in L. These constraints are not needed if the process for constructing $\neg\varphi'_p$ outlined above is used. $\neg\varphi'_p$ has removed all negative variables, and only positive variables remain in the formula.

4. The objective function is to minimize the number of positive literals: $\sum_{l \in L} x_l$. The optimal solutions to this BIP correspond to the prime implicants of $\neg\varphi \vee \neg p$.

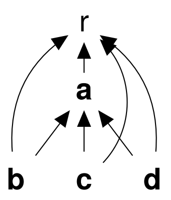
We demonstrate the BIP translation using the input DNF formula φ in Figure 5.3a. Computing the feature groups for feature a, we first construct the CNF formula $\neg\varphi'_{\mathsf{a}}$. The formula $\neg\varphi'_{\mathsf{a}}$ only includes negated variables in $\neg\varphi$ that imply a based on the implication graph in Figure 5.3b. The resulting formula is shown in Figure 5.3c. Finally, the resulting BIP translation is in Figure 5.3d. The optimal solution to this BIP problem has $l_{\neg\mathsf{b}}$ and $l_{\neg\mathsf{c}}$ set to 1 and $l_{\neg\mathsf{d}}$ set to 0. As a result, $\{\mathsf{b}, \mathsf{c}\}$ are a feature group with parent a.

5.6 Fge-BDD: Binary Decision Diagrams as Input

The third FGE variant assumes that the input φ is a binary decision diagram (BDD). BDDs are compact directed acyclic graphs of a Boolean function that provide constant time SAT and tautology checks. The size of a BDD is dependent on the Boolean function being represented and the variable ordering. Computing an optimal variable ordering is

1. $(\mathsf{r} \wedge \mathsf{a} \wedge \mathsf{b} \wedge \neg \mathsf{c} \wedge \neg \mathsf{d}) \vee$
2. $(\mathsf{r} \wedge \mathsf{a} \wedge \mathsf{b} \wedge \neg \mathsf{c} \wedge \mathsf{d}) \vee$
3. $(\mathsf{r} \wedge \mathsf{a} \wedge \neg \mathsf{b} \wedge \mathsf{c} \wedge \neg \mathsf{d}) \vee$
4. $(\mathsf{r} \wedge \mathsf{a} \wedge \neg \mathsf{b} \wedge \mathsf{c} \wedge \mathsf{d}) \vee$
5. $(\mathsf{r} \wedge \neg \mathsf{a} \wedge \neg \mathsf{b} \wedge \neg \mathsf{c} \wedge \neg \mathsf{d})$

(a) DNF formula φ



(b) Implication graph

$$\begin{aligned} (\neg b) \wedge \\ (\neg b \vee \neg d) \wedge \\ (\neg c) \wedge \\ (\neg c \vee \neg d) \wedge \end{aligned}$$

(c) CNF formula $\neg\varphi'_{\mathsf{a}}$

$$\begin{aligned} l_{\neg \mathsf{b}} &\geq 1 \\ l_{\neg \mathsf{b}} + l_{\neg \mathsf{d}} &\geq 1 \\ l_{\neg \mathsf{c}} &\geq 1 \\ l_{\neg \mathsf{c}} + l_{\neg \mathsf{d}} &\geq 1 \end{aligned}$$

(d) BIP problem for $\neg\varphi'_{\mathsf{a}}$

Figure 5.3: Translation of a DNF formula to a BIP problem for identifying OR-groups

a NP-hard problem, however, efficient heuristics for computing the variable ordering in feature models exists [MWCC08]. BDDs support logical operations such as conjunction, disjunction and existential quantification. The BDD-based feature graph extraction algorithm was introduced by Czarnecki and Wąsowski in [CW07]. We present the algorithm here since we use it as a baseline comparison for our evaluation of FGE-CNF in Section 5.9.

This description FGE-BDD uses an optimization for computing dead features, implication and mutex graphs by first computing the valid domains of variables in a BDD. A *valid domain* for a variable f given a formula φ is the set of valid values for f in any satisfiable assignment of φ [CW07]. For example, a dead feature is a variable with a valid domain of only *false*. The valid domains can be computed in linear time with respect to the size of the BDD [HJA07]. We now describe FGE-BDD according to the generic algorithm in Figure 5.2:

Lines 1–2 Dead Features: Dead features are variables that are always *false* in all satisfying assignments of φ, and are equivalent to variables with valid domains of only *false*. Dead features are removed from the input BDD by applying existential quantification to φ. Dead feature computation takes $O(|\varphi| + |F|)$ time where $|\varphi|$ is the size of the BDD.

Lines 3–4 Implications: An implication $f_i \rightarrow f_j$ exists if $\varphi \wedge f_i \wedge \neg f_j$ is inconsistent.

For each variable f_i, we construct a BDD for $\varphi \wedge f_i$, then identify all variables f_j that are *true* in all satisfying assignments of $\varphi \wedge f_i$ by computing valid domains for all variables. By using the valid domains computation, computing the implication graph takes $O(|F| \cdot |\varphi| + |F|^2)$ time.

Line 8 Mutual Exclusions: A mutex is found if $\varphi \wedge f_i \wedge f_j$ is inconsistent. Computing mutual exclusions is similar to implications, where we first construct BDDs for $\varphi \wedge f_i$. However, mutual exclusions are detected by identifying f_j's that are *false* in all satisfying assignments. This computation can also be done by computing the valid domains for all variables resulting in a runtime of $O(|F| \cdot |\varphi| + |F|^2)$.

Line 11–12 Or Groups: OR-groups with a parent p are found by computing prime implicates of $\varphi \wedge p$ or equivalently, prime implicants of $\neg\varphi \wedge \neg p$. Coudert and Madre describe two algorithms for computing prime implicants [CM92]—IP1 and IP2. IP2 has a runtime that is polynomial to $|\varphi|$, thus making OR-group computation polynomial to $|\varphi|$. However, IP1 has a better runtime on practical applications. We use a modified version of IP1 that returns only implicants consisting of all negative literals in our implementation of FGE-BDD.

5.7 Fge-FCA: Formal Concept Analysis-Based

FGE-FCA [RPK11] is an adaptation of the algorithm by Ryssel et al. to the infrastructure of of FGE. FGE-FCA is based on based on *formal concept analysis* (FCA) and addresses the same usage scenarios as FGE-DNF where the input is a set of configurations. FGE-FCA translates a list of configurations into a feature graph. We present the core steps of the approach by relating it to the steps of the generic FGE algorithm presented in Figure 5.2. We present this algorithm here since we use it as a baseline comparison for our evaluation of FGE-DNF in Section 5.9. This section was written by Ryssel for [SRA+13] and adapted for this thesis.

We begin by summarizing the concepts of FCA. FCA is a mathematical approach for identifying a lattice of concepts based on a formal context. A *formal context* is a triple (G, M, I), where G is a set of objects, M is a set of attributes and $I \subseteq G \times M$ is a binary *has-relation* relating objects and attributes. In the context of feature models, the set of objects G represents a set of configurations, and the set of attributes M represents features. The relation I maps each configuration to the features that are contained in the configuration. In Figure 5.4a, we express a formal context as a table where the

	a	b	c	d	e
1		x	x		x
2	x	x	x	x	
3	x	x	x		x
4		x	x	x	
5			x		

(a) A formal context expressed as a table

	Attribute Concept (O,A)	
	Extent O	Intent A
a	{ 2,3 }	{ a,b,c }
b	{ 1,2,3,4 }	{ b,c }
c	{ 1,2,3,4,5 }	{ c }
d	{ 2,4 }	{ b,c,d }
e	{ 1,3 }	{ b,c,e }

(b) Attributes and their corresponding attribute concepts

$$\begin{tikzcd}
 & c & \\
 & b \arrow[u] & \\
a \arrow[ur] & d \arrow[u] & e \arrow[ul]
\end{tikzcd}$$

(c) Implication graph

Figure 5.4: Formal context and its attribute concepts

numbered rows are objects and the lettered columns are attributes. An **x** in the table indicates the object having the attribute.

FCA builds a lattice of formal concepts based on the formal context. A *formal concept* is a pair (A,B) such that $A \subseteq G$, $B \subseteq M$ and every object in A has every attribute in B, and every object not in A is missing an attribute in B. FCA defines a operator $(\cdot)'$ that, given a set of attributes $B \subseteq M$, returns the objects that B share. $(\cdot)'$ applied to a set of objects $A \subseteq G$, returns the common attributes that share the objects A.

An attribute concept is a special kind of formal concept computed for a single attribute. Given an attribute $m \in M$, the *attribute concept* is a pair (A,B) where A is the set of all objects containing m, and B is the set of all attributes having all objects in A. Formally defined, the attribute concept is $(\{m\}', \{m\}'')$. Given a concept (O,A), the *extent* of a refers to the object component, O, and the *intent* refers to the attribute component, A. The attribute concepts for the formal context in Figure 5.4a are shown in Figure 5.4b. Computing attribute concepts relies on computing the extent (i.e., objects) for each attribute. Computing the extent for one attribute takes at most $|G|$ time, so computing attribute concepts takes $O(|M||G|)$ time.

Now that we defined the basic concepts of FCA, we can describe FGE-FCA according to the structure of the generic FGE algorithm in Figure 5.2:

Lines 1–2, Dead Features: Dead features are identified by finding attributes that not contained in any object. In other words, the extent for each dead attribute concept is empty. This can be computed in $O(|M|)$ time by checking all the computed attribute concepts.

Line 5, And Groups: In FGE-FCA, AND-groups can be identified prior to computing the implication graph by computing the extents for all attribute concepts. If two attribute concepts have the same extents, then they become part of the same AND-group.

Lines 3–4, Implications: Given two attributes, $u, v \in M$, an implication $u \rightarrow v$ holds iff the extent of u is a subset of the extent of v. Formally, $\{u\}' \subseteq \{v\}'$. Subsets are checked for all pairwise combinations of attributes, thus, requiring $O(|G|M|^2)$ time. Ryssel detects implications by comparing the attribute concepts and building an *attribute concept graph* [RPK11]. The attribute concept graph is equivalent to an implication graph. The implication graph for our example is shown in Figure 5.4c.

Line 8, Mutual Exclusions: Two attributes are mutually exclusive iff their extents are disjoint, i.e., given $u, v \in M$, $\{u\}' \cap \{v\}'$ is empty. Computing mutual exclusions takes $O(|G||M|^2)$ time.

Line 11–12, Or Groups: An OR-group with parent p and members $m_1, \ldots, m_n$ is present if the extent (i.e., configurations) of the parent $\{p\}'$ is equal to the union of extents of its members $\{m_1\}' \cup \cdots \cup \{m_n\}'$. Furthermore, OR-groups must be minimal, such that no member m_i can be removed without violating the equivalence of $\{m_1\}' \cup \cdots \cup \{m_n\}' = \{p\}'$.

Identifying these minimal sets of attribute concepts is a *minimal set cover problem* and $\{m_1', \ldots, m_n'\}$ is a set cover of p'. Similar to the other FGE algorithms, FGE-FCA restricts the possible set cover candidates to only the children of the parent. Group computation in FGE-FCA translates to the following set cover problem [SRA+13]:

$$\text{Find all minimal } C \subseteq M \text{ where } \bigcup_{c \in C} \{c\}' = \{p\}' \text{ and } C \text{ are all children of } p \qquad (5.2)$$

For our example, to compute OR-groups where b is the parent, we first compute the extent of the parent b, $\{b\}'$ resulting in the set $\{1, 2, 3, 4\}$ (Figure 5.4b). The set cover problem is formulated such that we find the minimal set covers $C \subseteq M$ where C are all children of b, and $\bigcup_{c \in C}\{c\}' = \{1,2,3,4\}$. In our example, the possible candidates for C are the children of b: $\{\mathsf{a},\mathsf{d},\mathsf{e}\}$. Computing the extents for each of these attributes results in: $\{\mathsf{a}\}' = \{2,3\}$, $\{\mathsf{d}\}' = \{2,4\}$, $\{\mathsf{e}\}' = \{1,3\}$. Out of these sets, the extents of d and e are not only a set cover for the extent of the parent b, but also the minimal set cover. As a result, d and e are a feature group of b.

5.8 Selecting a Feature Diagram from a Feature Graph

The synthesized feature graph describes all feature diagrams that are complete with respect to the input dependencies. A maximal feature diagram can be extracted automatically from a feature graph using following procedure:

1. establish the hierarchy by finding a spanning tree over FG and move all the edges not in the spanning tree to cross-tree implies edges (E_i);
2. for each SCC, select a single feature to be the common ancestor of all other members of the SCC—create mandatory edges (E_m) between the features;
3. select feature groups by greedily selecting non-overlapping subsets from G_o, G_x, and G_m to form syntactically correct groups;
4. Remove from E_x all edges that participate in selected MUTEX- or XOR-groups.

While this procedure extracts a feature diagram from a feature graph automatically, the resulting diagram may not describe the expected domain semantics. In Chapter 6, we describe our semi-automated approach for selecting a hierarchy from a feature graph (Steps 1 and 2 in the above procedure). Other procedures that derive a feature diagram from a feature graph include the feature model operations by Acher [Ach11] and the model editor by Janota et al. [JKW08]. We discuss these related techniques in Chapter 7.

5.9 Experimental Evaluation

In Section 5.1, we described several synthesis scenarios that operate on large input. For example, model management operations (e.g., model merge and difference) on the Linux variability model requires a synthesis algorithm capable of reasoning about several thousand features. Another scenario involves reverse engineering a FM from source code of a large scale, variability-rich project such as FreeBSD. Synthesizing FMs from such large projects poses a significant scalability challenge for FM synthesis algorithms. FM synthesis techniques are not only restricted to large scale applications; they are also used in interactive applications such as a model builder [JKW08]. The range of application scenarios for FM synthesis requires understanding the performance of the algorithms on real world input.

We perform two experiments to evaluate the scalability of FGE-CNF and FGE-DNF against existing FM synthesis algorithms with input representative of input used in practical FM synthesis scenarios. We compare the running times of FGE-CNF and FGE-DNF against baseline implementations using input extracted from real world FMs. We describe our experiments according to the reporting style outlined by Wohlin [Woh00].

5.9.1 Goal Definition

Objects of Study Our evaluation consists of two experiments to evaluate our two algorithms: FGE-CNF and FGE-DNF. Both experiments have the same format and only differ in the comparison algorithms. The two objects of study are the two algorithms presented in this paper: FGE-CNF and FGE-DNF.

Purpose The purpose of the experiments is to evaluate the performance of FGE-CNF and FGE-DNF. Our goal is to evaluate their performance compared to related FM synthesis algorithms that produce the same output and with the similar inputs. For this purpose, we use our previous BDD-based algorithm [CW07] that we call FGE-BDD as the comparison algorithm for FGE-CNF, and the FCA-based algorithm [RPK11] that we call FGE-FCA as the comparison for FGE-DNF.

Perspective The perspective is from the point of view of a user of our synthesis algorithms in one of the FM synthesis scenarios described in Section 5.1.

Quality Focus We use computation time as the quality focus for our evaluation. While the algorithms are all implemented using a JVM language (i.e., Java or Scala), the libraries, data structures, and tools used by the algorithms are different. Computation time is a common metric that we can use to compare the different algorithms. Note that we do not evaluate the quality of the produced models since both our algorithms and the baseline comparisons constructs the same compact representation of *all possible* diagrams consistent with the input.

Context The experiment is run on input derived from generated FMs and FMs in the SPLOT model repository including the Linux variability model [SLB+10]. As we established in Thm. 5.1, the decision version of the feature model synthesis problem (FMS) is NP-hard. There exists input that would perform poorly on both FGE-CNF and FGE-DNF.

Ideally, our evaluation should use input from synthesis scenarios in real software projects. However, such data for feature model synthesis is not easily accessible. Instead, we used feature models from the SPLOT model repository. The input was selected since the input is publicly available and is representative of input used in the FM synthesis scenarios that we described in Section 5.1. We use the Linux variability model to approximate input for Scenario 1 where feature model synthesis is used for tool-assisted reverse engineering. The feature models from the SPLOT model repository are naturally representative of the feature models used for FM merge operations in Scenario 3. Feature models also describe a set of legal variants making it suitable for approximating input for Scenario 2.

Summarized Research Objective Analyze FGE-CNF and FGE-DNF for the purpose of comparing their performance against related FM synthesis algorithms with respect to their computation time from the perspective of a user of a FM synthesis scenario in the context of input derived from generated FMs and FMs from a model repository.

5.9.2 Hypothesis Formulation

Our evaluation compares the computation time for the three components in FGE that differ across the evaluation algorithms: implication graph computation, mutex graph computation, and OR-group computation. The goal of this evaluation is to show that the algorithms in this paper, FGE-CNF and FGE-DNF, are faster than their respective comparison algorithms, FGE-BDD and FGE-FCA. Our null hypothesis is as follows:

Null Hypothesis, H_0 For each component of FGE (i.e., implication graph, mutex graph, and OR-group computation), there is no difference in the mean computation times for FGE-CNF and FGE-BDD. Similarly, there is no different in the mean computation times for FGE-DNF and FGE-FCA.

5.9.3 Variable Selection

The independent variables are the size and complexity of the constraints in the models used in the evaluation. The dependent variables are the computation time for each algorithm.

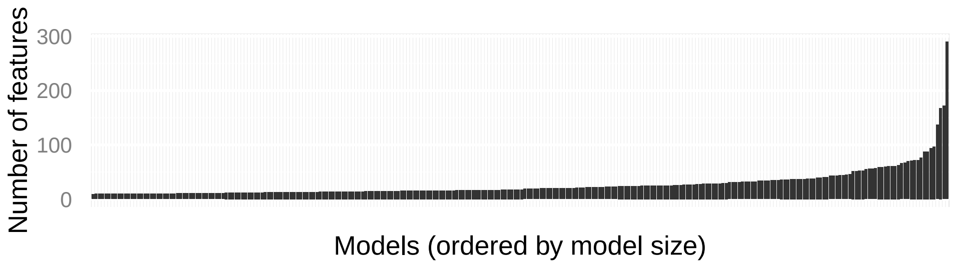
5.9.4 Selection of Subjects

Our evaluation uses two datasets: a dataset derived from real world FMs in the SPLOT model repository [MBC09] that also includes the Linux variability model [SLB+10], and a second dataset derived from randomly generated FMs.

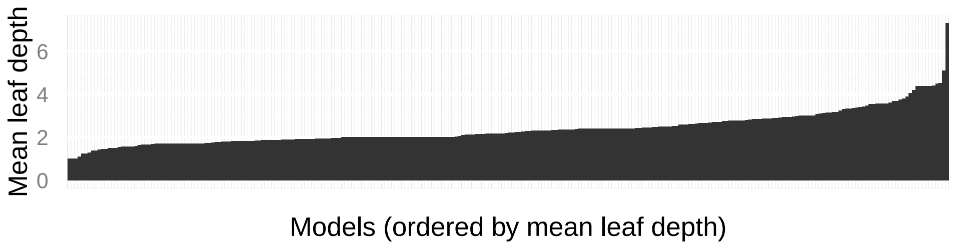
The real world dataset consists of input derived from 267 FMs and the Linux variability model.

Figure 5.5a shows the number of features over the models in the SPLOT dataset. The models have a median of 18 features and an average of 27 features. In Figure 5.5b, the average leaf depth of the individual models is shown. If we consider all features across all models, a feature has a median leaf depth of 2 and an average leaf depth of 2.3. Figure 5.5c is a plot of the mean branching factor for each individual model. Considering all features across all models, a non-leaf feature has a median of 2 children, and an average of 3 children. Finally, Figure 5.5d is a plot of the percentage of feature groups, and grouped features with respect to the total number of features including feature groups. On the left side of the plot, there are 38 models that contain no feature groups and thus, no grouped features. There are 38 models that do not contain any feature groups. The *Electronic Shopping* model has 40 feature groups—the most in the dataset. The median number of feature groups is 2 and on average a model has 4 feature groups.

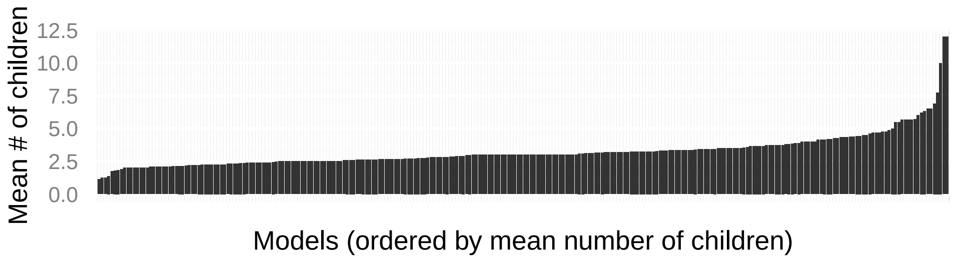
We used the variability model from the x86 architecture in the v2.6.28.6 of the Linux kernel. This version of the Linux kernel has 5426 features. A non-leaf featuer has a median of 1 child and on average, 3.65 children. There are 32 feature groups in the model.



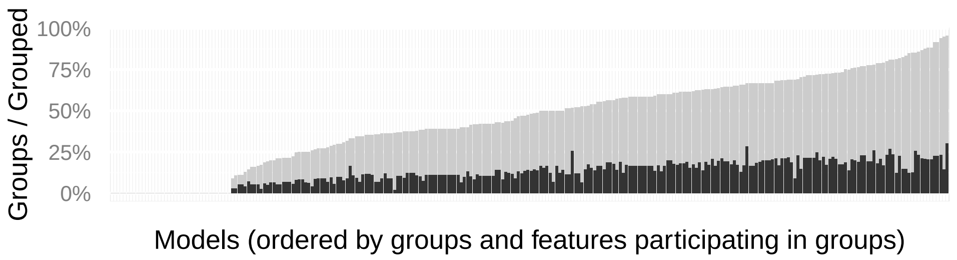
(a) Number of features per model



(b) Mean leaf depth per model



(c) Mean branching factor per model



(d) Relative number of feature groups (in black) and grouped features (in grey) per model

Figure 5.5: SPLOT model dataset properties

The generated dataset consists of 20 feature models with cross-tree constraints in 3-CNF form—clauses that have exactly three literals. Mendonca found that SAT problems derived from FMs are relatively easy, but SAT problems derived from 3-CNF FMs are typically harder than SAT problems derived from real world feature models [MWC09]. There are 10 models with roughly 100 features and another 10 models with roughly 200 features in this dataset. The number of children per parent node varied from 1 to 6. 10% of features appear in cross-tree constraints and features have an equal probability of being an optional feature, a mandatory feature, or as part of an OR-group or XOR-group.

For FGE-CNF, the CNF input is derived using the configuration semantics of the feature models [Bat05] (Formula 2.3). The configuration semantics naturally create a CNF formula.

For FGE-DNF, we create the DNF input by enumerating the legal configurations of a model. The number of legal configurations of a feature model can grow exponentially. As a result, our evaluation of FGE-DNF includes only the models with 100,000 or less configurations to limit the size of the input. The real world dataset for FGE-DNF consists of 199 models with 100,000 or less configurations and excludes the Linux variability model.

5.9.5 Experiment Design

In this section, we describe our experiment design according to the general design principles by Wholin [Woh00].

Randomization Randomization in this context refers to randomly assigning models to the evaluation subjects. We do not apply randomization to our experiments. The algorithms are evaluated on all models in their respective datasets.

Blocking Our evaluation measures computation time of the algorithms. The algorithms in the evaluation are implemented in Java or Scala: both languages that run on the Java Virtual Machine (JVM). We apply blocking to measure the construction of the implication graph, mutex graph, and OR-group computation or the alternative XOR-group computation for input derived from the Linux kernel. The blocking avoids including overhead associated with loading libraries or purely graph-based operations that are common across all the algorithms (e.g., AND-group computation).

We also separated the input into two dataset: one consisting of input derived from real world models, and the other consisting of input from generated 3-CNF models. The 3-CNF models are typically harder than SAT problems derived from real world FMs [MWC09], so we report their results separate from those of the SPLOT models.

Balancing Our datasets for the two experiments consists of all of the available SPLOT feature models as of October 18, 2012, the Linux variability model, and input from generated FMs. Properties of all FMs are not known, but the SPLOT model repository, along with the Linux variability model is the best known collection of FMs.

Design Type Both experiments are one factor with two treatment types. The factor is the input derived from the model, and the treatments are FGE-CNF and FGE-BDD for one experiment, and FGE-DNF and FGE-FCA for the other.

5.9.6 Operation

We executed our algorithms and the two baseline comparisons on an Intel Core 2 Q6600 quad core processor with 3GB of available memory. We repeated the execution of the algorithms 5 times and present the average of the computation times in our results.

Fge-CNF Evaluation Both FGE-BDD and FGE-CNF operate on input derived from dependencies—constraints expressed as propositional formulas and not as a set of configurations and constructs a feature graph. FGE-BDD was implemented in Java and uses the JavaBDD 1.0b2 library[1]. FGE-CNF was implemented using the Scala programming language [OAC+06] with the core SAT4J library[2]. SAT4J is a widely used open-source Java interface to SAT solvers that implements the initial Minisat specification [ES03]. Using Scala for FGE-CNF was a choice based on language preference instead of a performance advantage. Both Java and Scala run on the Java Virtual Machine (JVM). The performance of the PIG algorithm for computing OR-groups in FGE-CNF is heavily dependent on the expense of forward and backward subsumption and we have implemented the algorithms by Zhang [Zha05].

[1] http://javabdd.sourceforge.net/
[2] http://sat4j.org/

For input derived from the Linux variability model, both FGE-BDD and FGE-CNF were not able to compute OR-groups. We were unable to construct a BDD representing the constraints of the variability model within our memory constraints. FGE-CNF timed out when computing OR-groups. We used the alternative XOR-group computation (Section 5.3) in order to compute XOR-groups for FGE-CNF without a dependency on OR-groups.

Variable ordering affects the performance and space consumption of both SAT solvers and BDDs. We performed some limited experiments with the variable ordering heuristics for BDDs as proposed by Mendonca [Men09]. However, even with Mendonca's heuristics, we were still unable to build a BDD for the input derived from the Linux kernel. We did not use any specific variable ordering for FGE-CNF.

We measured the computation times for computing the implication graph, mutex graph, and OR-groups. We used JavaBDD 1.0b2 for the BDD-based implementation and set the timeout to 90 seconds for FGE-BDD. For any computation that timed out on FGE-BDD, we recorded a time of 90 seconds.

Fge-DNF Evaluation We evaluate FGE-DNF against the formal concept analysis (FCA)-based approach by Ryssel et al. [RPK11]. We presented the algorithm according to the structure of the FGE algorithm as FGE-FCA in Section 5.7. Both FGE-DNF and FGE-FCA constructs an identical feature graph and use a set of configurations as input. While the inputs and output is the same for FGE-DNF and FGE-FCA, the procedure for computing feature groups is vastly different. Whereas FGE-DNF identifies feature groups by translating a DNF formula to a BIP problem, FGE-FCA identifies feature groups by translating a formal context to a set cover problem. Both BIP and set cover problems are NP-hard, thus are in the same class of theoretical complexity. However, as often the case with NP-hard problems, the performance of these algorithms will differ in practical scenarios.

The approaches for identifying dead features, implications, and mutual exclusions are essentially equivalent in both FGE-DNF and FGE-FCA. For dead features, both approaches search for features that appear in no configurations. In FGE-DNF, implications and mutual exclusions are computed by checking for the presence of a pair of features across all terms in φ. FGE-FCA operates on the same principle by checking whether the configurations of one feature are a subset of another. Since both approaches are equivalent for these components, we focus the evaluation on the computation of OR-groups where the approaches are quite different. Furthermore, OR-group computation has the highest impact on the performance of the synthesis algorithm.

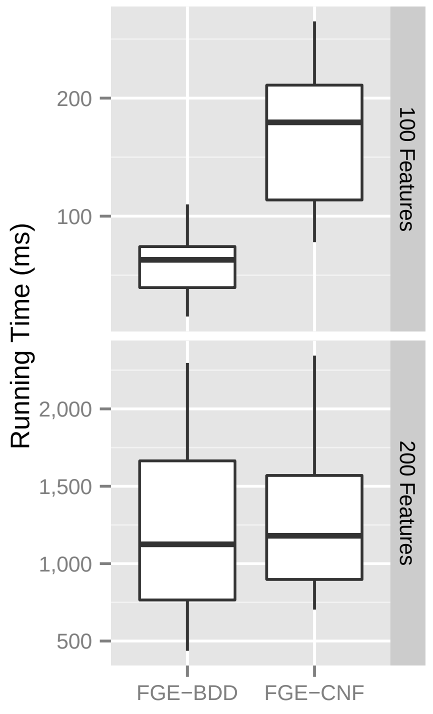
FGE-DNF was implemented in Java using the core SAT4J library. Our implementation of FGE-DNF uses the PRIME algorithm to solve the BIP problem for identifying OR-groups. PRIME is an extension by Andersen [And09] of the MIN_PRIME algorithm [EPP95]. PRIME incrementally enumerates all prime implicants of a CNF formula. PRIME first finds a prime implicant using MIN_PRIME, then adds it as a constraint to find the next prime implicant. This process repeats until all prime implicants are found. An advantage of using SAT4J for our implementation is that it natively supports the cardinality constraints needed for PRIME. We did not use a specific variable ordering for FGE-DNF. FGE-FCA was implemented by Ryssel in Java and uses the Colibri library for operations related to formal concept analysis [3] and a custom implementation for calculating minimal set covers needed for feature group computation.

5.9.7 Results: Fge-CNF Evaluation

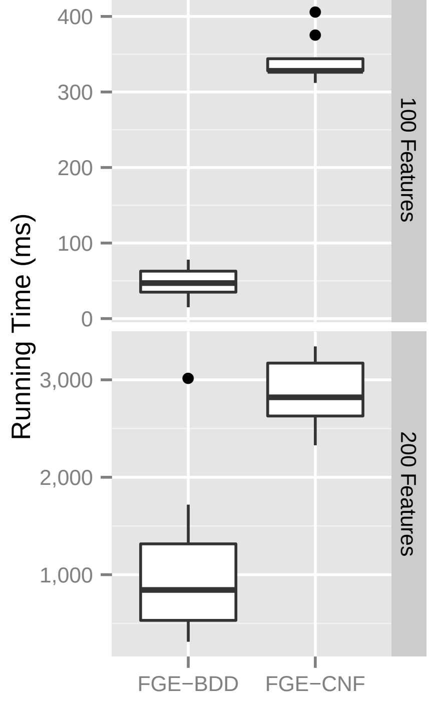
Implications Computing the implication graph was fast overall, taking less than 400ms for both FGE-BDD and FGE-CNF in 262 of the 264 models in the SPLOT dataset, leaving only the *Electronic Shopping* and *Printers* models. For the *Electronic Shopping* model, FGE-BDD took 11.7s compared to 3.3s for FGE-CNF. This model is quite large with 290 features, thus, constructing the BDD for each feature $\varphi \wedge f$ took considerable time. The situation with the *Printers* model is reversed—FGE-BDD was very fast, computing implications in 22ms while FGE-CNF took roughly 2s. The *Printers* model is smaller than *Electronic Shopping* with 172 features, however, it has many mandatory features and feature groups. As a result, FGE-BDD was extremely efficient at computing implications since a large portion of the features imply all other features requiring only a single BDD to be constructed. We see similar results in Figure 5.6a for the generated dataset. FGE-BDD was faster for the models with 100 features, however, the results were similar for the generated models with 200 features. In general, both FGE-BDD and FGE-CNF were fast at computing implications. Computing implications took 1.7 hours for the input derived from the Linux kernel variability model.

Mutual Exclusions For the SPLOT dataset, mutual exclusion computation took roughly the same amount of time as computing implications in both FGE-CNF and FGE-BDD. Mutual exclusion computation was generally faster for FGE-BDD compared to FGE-CNF for models that did not timeout. However, on input derived from Linux,

[3] http://code.google.com/p/colibri-java/



(a) Implication Graph



(b) Mutex Graph

Figure 5.6: Running times for FGE-BDD and FGE-CNF on the input derived from randomly generated models

		FGE-BDD				FGE-CNF			
id	# features	IG	MG	OR	total[1]	IG	MG	OR	total[a]
1	100	78	47	•	•	110	328	1.3 e5	1.3 e5
2	100	63	47	•	•	265	328	4875	5562
3	98	31	31	•	•	216	328	266	875
4	100	78	63	•	•	172	406	4.3 e4	4.4 e4
5	100	110	78	•	•	125	344	1.6 e6	1.7 e6
6	100	63	47	•	•	187	344	7.0 e4	7.1 e4
7	100	62	62	•	•	187	312	1.0 e5	1.0 e5
8	100	63	63	•	•	235	375	1.8 e5	1.8 e5
9	100	15	15	•	•	78	328	906	2609
10	100	32	16	•	•	110	328	1.4 e5	1.4 e5
11	200	734	531	•	•	1438	2906	1.6 e5	1.6 e5
12	200	438	328	•	•	703	2640	•	•
13	200	437	312	•	•	1593	2328	8140	1.2 e4
14	200	2297	1719	•	•	890	2625	•	•
15	200	1766	1422	•	•	1500	3343	•	•
16	200	969	734	•	•	1641	2735	•	•
17	200	2062	78	•	•	921	3235	•	•
18	200	1281	953	•	•	2344	3187	•	•
19	199	859	532	•	•	735	3125	•	•
20	200	1359	1000	•	•	922	2469	•	•
Linux	5701	○	○	○	○	1.7 h	6.1 h	•	7.8 h[b]

• timeout ○ out of memory

[a] Running times include computation of AND-groups and MUTEX-groups.
[b] Total run time for Linux includes computation of XOR-groups.

Table 5.1: Running times (in ms) of the FGE-BDD and FGE-CNF on input derived from randomly generated models and Linux

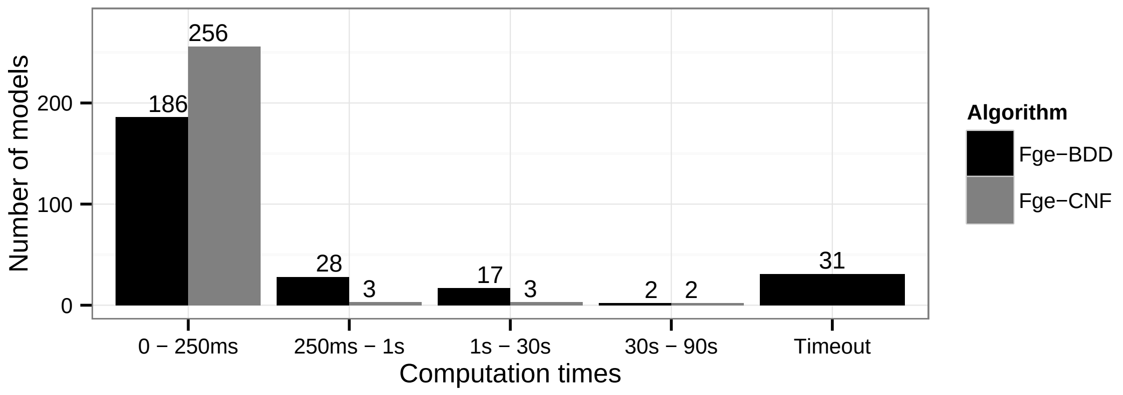


Figure 5.7: Running times for group computation in FGE-BDD and FGE-CNF on input derived from SPLOT models

mutual exclusion computation was roughly 3.5 times slower than implication computation for FGE-CNF, taking 6.1 hours to compute the entire mutual exclusion graph. Figure 5.6b shows similar results for the generated dataset where FGE-CNF was slower than FGE-BDD. One reason for the longer runtime is that our SAT solver is tuned to prefer models with positive literals when computing satisfiability. Positive literals are useful as an optimization for implication computation since it can invalidating many implications with a single model. However, such models are not as useful for mutual exclusion computation. It is likely that further tuning the SAT solver for mutual exclusion computation could significantly improve its performance. Since we were unable to create the BDD for Linux, FGE-BDD was unable to compute exclusions.

Group Computation Figure 5.7 shows a breakdown of the group computation runtimes for FGE-BDD and FGE-CNF with non-incremental OR-group computation on the SPLOT dataset. The majority of models for FGE-BDD and FGE-CNF computed groups in less than 250 milliseconds. FGE-BDD, however, had 31 models that did not complete computation by the 90 second timeout. FGE-CNF was able to compute OR-groups for all SPLOT models. Overall, FGE-CNF was on average 11 seconds faster than FGE-BDD. Furthermore, FGE-BDD was unable to compute OR-groups for any of the generated models (Table 5.1). FGE-CNF was able to compute OR-groups for 12 of the 20 generated models.

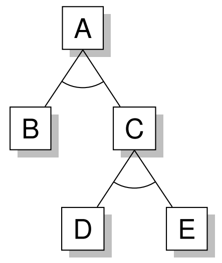
We now look at the models where FGE-BDD completed group computation, but took more than one second in Table 5.2. FGE-CNF is the clearly faster here by computing groups in less than 30ms; FGE-BDD managed a runtime of 98ms at the fastest, and the

Model	Features	FGE-BDD (ms)	FGE-CNF (ms)
SPL SimulES, PnP	32	98	0
Reference Management Software	31	640	0
Hotel Product Line	55	1113	1
SmartHome vConejero	59	1204	0
Tree Growth Simulator	32	1233	5
Reference Management Software (2)	31	1365	13
Sienna	38	1391	11
SPL SimulES, PnP	53	2115	1
Tablets	38	2343	12
SPL SimulES, PnP (2)	40	2676	3
Database Tool	59	2702	0
Plone Meeting	57	8174	1
HIS	67	9323	1
Tool Analysis	31	9681	2
SPL-Doctor Chat	34	16001	0
AndroidSPL	45	18768	3
Eclipse Platform	29	19000	1
Mobile Media 2	43	19901	1
PFTest1	56	20549	26
Electronic Drum	52	32351	2
Meshing Tool Generator	40	43434	14

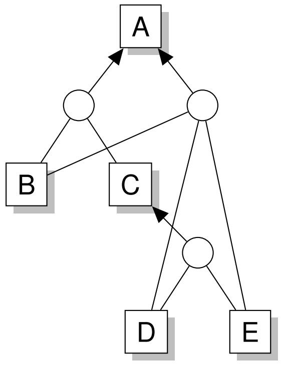
Table 5.2: Group computation runtimes for SPLOT models that took more than a second on FGE-BDD

Model	Features	Detected Groups	FGE-CNF (ms)
Agenda	34	1	1
Video Player	71	5	1
J2EE Web Architecture	77	11	2
Video Player	53	9	2
Smart Home v2.2	60	6	3
Smart Home	56	5	3
OW2-FraSCAti-1.4	63	2	3
Database Tools	70	8	3
Reuso - UFRJ - Eclipse1	72	7	4
bCMS System	66	9	4
Web Portal	43	6	5
Consolas de Videojuegos	41	20	6
DS Sample	41	6	11
Model Transformation	88	32	11
Car Selection	72	25	13
Documentation Generation	44	41	15
XText	137	0	25
Coche Ecologico	94	33	55
UP eStructural	97	34	65
Arcade Game PL	61	45	67
Letovanje	43	41	83
E-Science Application	61	48	86
E-Science Application (2)	61	48	234
Thread	44	9	310
Printers	172	46	320
FM Test	168	33	392
Experiment Environment	35	162	8967
Software Stack	37	119	12923
Software Stack (2)	37	119	12997
Electronic Shopping	290	94	42239
Dell Laptop / Notebook Computers	46	323	84939

Table 5.3: Group computation runtimes for FGE-CNF on SPLOT models that timed out for FGE-BDD (i.e., took more than 90s)



(a) Nested feature groups



(b) Detected feature groups as a feature graph. A circle represents an XOR-group where the arrowhead indicates the parent and the lines are the group members.

Figure 5.8: Detected feature groups

slowest model took 43 seconds. Table 5.3 shows the runtimes for group computation using FGE-CNF for models that resulted in a timeout on FGE-BDD (i.e., took more than 90 seconds). Here, the majority of the models took less than one second to compute feature groups. For the five models that had longer runtimes, either the number of features, or the number of detected groups—the number of groups identified by the FGE algorithm—was high. The slowest model, *Dell Laptop / Notebook Computers*, had relatively few features, but a high number of detected feature groups. The second slowest model, *Electronic Shopping*, contained a high number of features and also a relatively high number of detected feature groups.

We demonstrate how some inputs can result in a large number feature groups for FGE with the feature model in Figure 5.8a. There are two feature groups in this model: one between features B and C, and another between D and E. If we translate this model to a propositional formula and use it as input to FGE, three feature groups will be detected (Figure 5.8b). Two of these feature groups are present in the original model. FGE detects a new feature group that consists of B, D, and E. Examining the original FM (Figure 5.8a), an XOR-group between B, D, and E is actually present—selecting B excludes C and consequently D and E. Running Pearson's product-moment correlation between the number of detected feature groups and the computation time for FGE-CNF results in a coefficient of 0.85 with a p-value of $< 2.2 \times 10^{-16}$, indicating a very strong linear correlation between the two variables. FGE-BDD does not have such a strong correlation between number of detected feature groups and the computation time, with

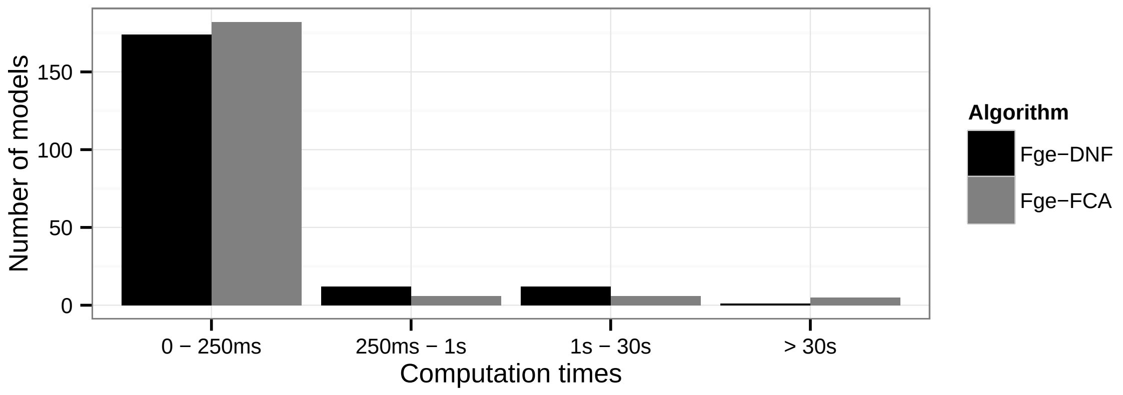


Figure 5.9: Distribution of OR-group computation times for FGE-DNF and FGE-FCA

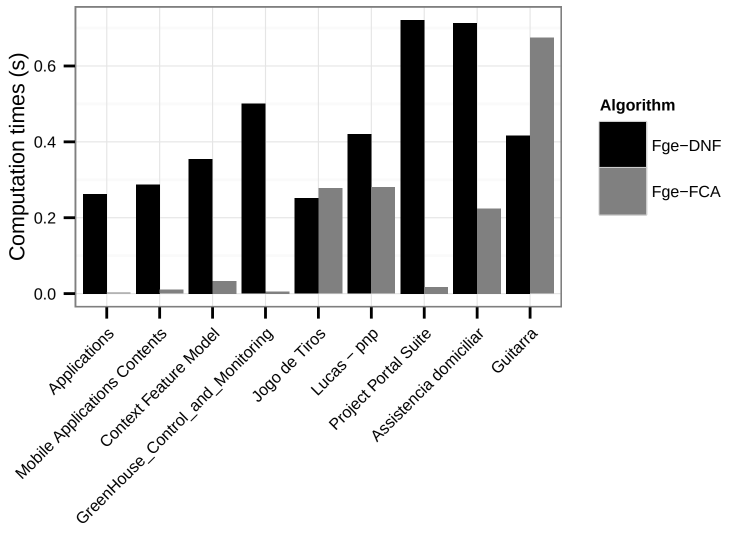
coefficient of 0.28 with a p-value of 7.4×10^{-6}.

5.9.8 Results: Fge-DNF Evaluation

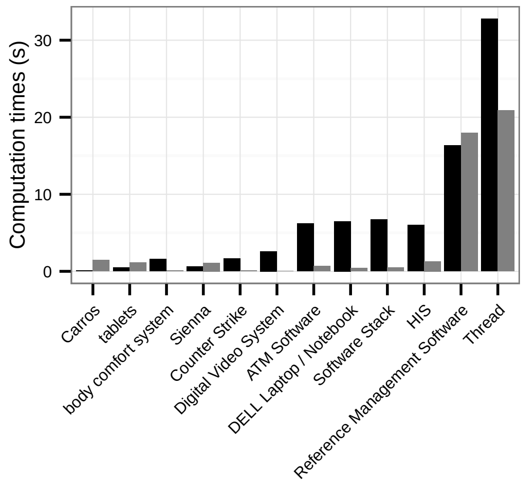
Implications and Mutual Exclusions The run times for computing implication and mutex graphs were similar in both FGE-DNF and FGE-FCA, however, FGE-FCA was slightly faster in general. The difference in computation times may be due to differences in language libraries (i.e., Scala for FGE-DNF vs. Java for FGE-FCA. As we discussed in Section 5.9.6, both approaches describe the same process for identifying implications and mutual exclusions so similar computation times was to be expected.

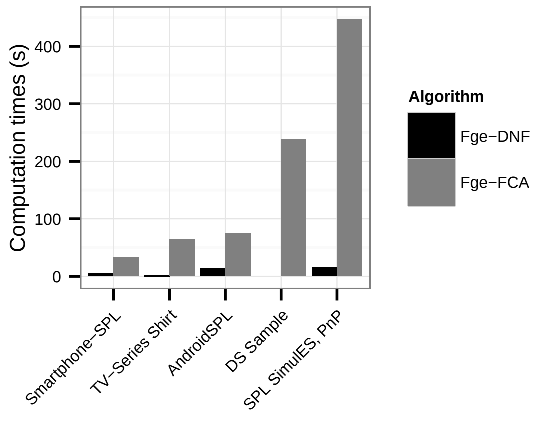
Group Computation The main difference in performance between FGE-DNF and FGE-FCA lies in the OR-group computation. Figure 5.9 shows the result grouped by their computation times into four ranges. We see that OR-group computation was fast for most models, taking less than 250 ms for both FGE-DNF and FGE-FCA.

In Figure 5.10, the results for the individual models are shown in detail. We limit models where OR-group computation took less than 250ms since the runtime was fast for both algorithms. For models that took 250ms to 1s (Figure 5.10a), the run times are similar, with FGE-FCA being slightly faster than FGE-DNF. However, the difference between the two algorithm was several hundred milliseconds and may be due to the different languages and frameworks used. For models that took 1s to 30s to compute OR-groups (Figure 5.10b), FGE-DNF was noticeably slower than FGE-FCA for most models. However, for models that took more than 30s, FGE-FCA was significantly slower than



(a) Models that took 250ms–1s





(b) Models that took 1s–30s (c) Models that took more than 30s

Figure 5.10: Group computation times for FGE-DNF and FGE-FCA

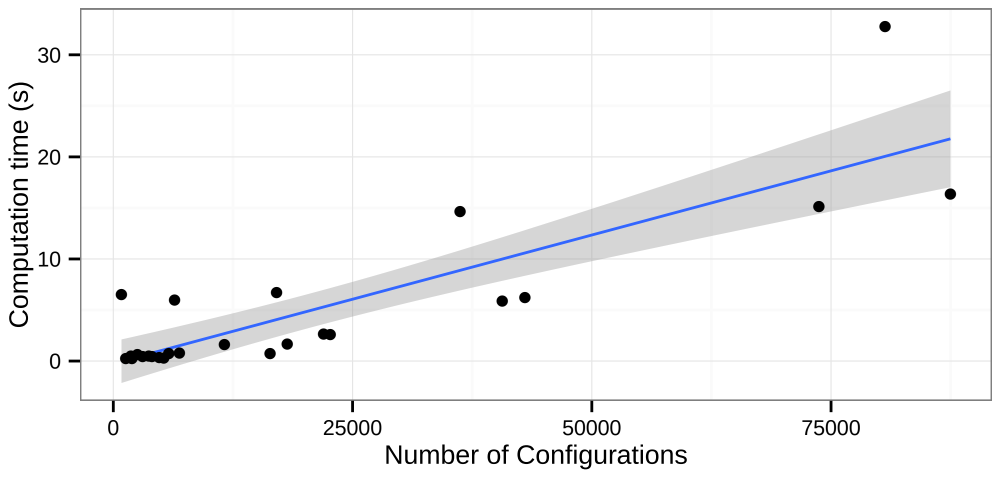


Figure 5.11: The effect of the number of configurations for OR-group computation on FGE-DNF

FGE-DNF on five models (Figure 5.10c). The models that exhibited slow performance mostly had a large number of feature groups, or many features that were independent of one another. Additionally, these groups and features may reside directly under the root features, or were mandatory with respect to the root. As a result, the input derived from such models translate to either a large BIP problem in case of FGE-DNF, or a large set cover problem for FGE-FCA.

Factors Contributing to Or-Group Computation Time

Fge-DNF We found that a high number of configurations contributed to a longer group computation time for FGE-DNF. We plot the number of configurations against the OR-group computation time for FGE-DNF in Figure 5.11. Applying Pearson's product-moment correlation on configurations and computation time for all models gives a coefficient of 0.896 with a p-value of 2.2×10^{-16}, indicating a strong linear correlation between the number of configurations and the OR-group computation time for FGE-DNF. If we just examine models that exhibit slower computation times of 250ms or more, we get a coefficient of 0.853 with 23 degrees of freedom and a p-value of 3.07×10^{-8}. indicating a strong linear correlation between the number of configurations and the OR-group computation time for FGE-DNF.

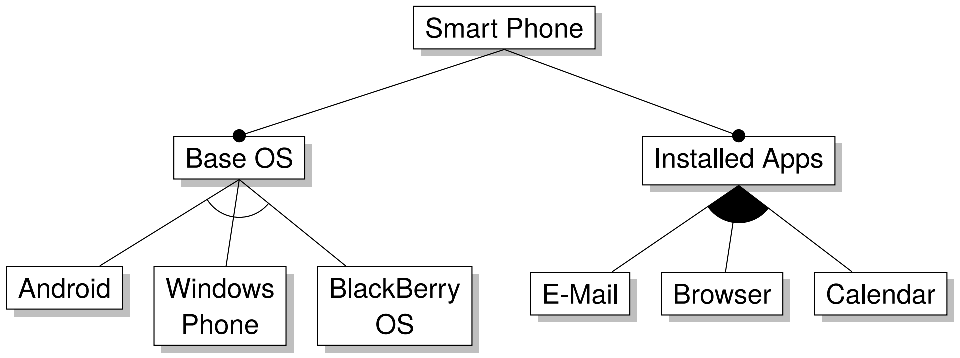


Figure 5.12: A snippet of the original Smart Phone SPL

Fge-FCA For FGE-FCA, a linear correlation between the number of configurations and computation time is not as evident. Applying Pearson's product-moment correlation on configurations and computation time for all models gives a coefficient of 0.501 with a p-value of 2.34×10^{-14}. However, if we examine just the models with computation times of 250ms or greater, the coefficient falls to 0.364 with a p-value of 0.075 and 15 degrees of freedom. This outcome on the slower models is not surprising since there are 5 models where FGE-FCA exhibits extremely slow performance (Figure 5.10c).

After manually inspecting these models, we found that all of the models that exhibited slow performance had a large number of feature groups that were mandatory with respect to root. For example, a snippet of this structure from the Smart Phone model, is shown in Figure 5.12. The input generated from this model contained a large number of features that imply the root feature and result in a large search problem for computing groups for FGE-FCA. Since group computation is potentially exponential, computation can quickly explode as the number of group members increase. The other models also shared similar properties and we describe the models below:

SmartPhone-SPL (Configuration of smart phones)

This model has three XOR-groups with three or six siblings and two OR-groups with seven siblings each. All this groups reside directly under root. The model has 23 cross-tree constraints, which amongst others reduce the three XOR-groups effectively to one independent XOR-group. Because of this, the configuration number of 40,640 is quite low (compared to 1.7×10^6 without the cross-tree constraints). However, the independent XOR-group results in a runtime of 33 s with FGE-FCA compared to 6 s with FGE-DNF.

TV-Series Shirt (Properties of shirts with TV-series logos)

This model has mainly three XOR-groups with three to four siblings, and one OR-group with seven siblings, all directly resided under root. Some cross-tree constraints reduce the number of configurations to 21,984. Computing OR-gruops on FGE-FCA results ina runtime of 64 s compared to 3 s with FGE-DNF.

DS Sample (Unknown domain)
This model has seven independent XOR-groups under root with two to 16 siblings. This results in 6912 configurations. The independent XOR-groups results in a high runtime of 238 s compared to 0.8 s with FGE-DNF.

AndroidSPL (Properties of Android-OS applications)
This model has four XOR-groups (two to four siblings in each) under root and eight partly nested features. The many options in this model increases the number of configurations to 36,240. The runtime is 74 s compared to 15 s with FGE-DNF.

SPL SimulES, PnP (Educational game for learning software project management)
This model has five XOR-groups under root with two or three siblings and nine options. All these relations are independent to each other resulting in 73,728 configurations. We found that the high number of configurations resulted in a runtime of 447 s for FGE-FCA compared to 15 s with FGE-DNF. Removing seven of the options will drop the runtime to 0.5 s for FGE-FCA.

5.9.9 Hypothesis Testing

In this section, we perform the t-test on the computation times for implications, mutual exclusions, and OR-groups on the real world dataset for the two experiments. We test our null hypothesis that there is no difference between the mean computation times of the different components between our algorithm (i.e., FGE-DNF or FGE-CNF) and their respective comparison algorithms.

Fge-CNF vs. Fge-BDD In Table 5.4a, we show the results of the t-test between FGE-CNF and FGE-BDD for computing implications and OR-groups. From the results for implication and mutual exclusion computation, we can not reject the null hypothesis for this component. The mean difference between the computation times were however, very small indicating that there is likely no difference between the two algorithms for computing these two components on the real world model dataset.

Component	Mean Difference (ms)	Degrees of Freedom	t-value	p-value
Implications	−16	263	−0.48	0.63
Mutual Exclusions	−20	263	−0.88	0.38
Or Groups	−10854	263	−6.32	1.13×10^{-9}

(a) *t*-test results for FGE-CNF vs. FGE-BDD

Component	Mean Difference (ms)	Degrees of Freedom	t-value	p-value
Implications	320	198	2.78	0.0059
Mutual Exclusions	166	198	3.28	0.0012
Or Groups	−3904	198	−3.46	0.1214

(b) *t*-test results for FGE-DNF vs. FGE-FCA

Table 5.4: *t*-test results for the real world dataset

The results for OR-group computation, however, show that we can reject the null hypothesis with a very low p-value for OR-groups. The computation time for groups is on average almost 11 seconds faster for FGE-CNF than FGE-BDD. Furthermore, for the 31 models that timed out for FGE-BDD, our results are generous towards FGE-BDD since we set a time of 90 seconds if the computation did not complete by that time. The mean difference between the algorithms is likely larger if we recorded the complete computation time for FGE-BDD.

Fge-DNF vs. Fge-FCA The results for the t-test in Table 5.4b show that we can reject the null hypothesis for computing implications and mutual exclusions. However, the difference is in favor of FGE-FCA where it is on average, it is 320ms faster than FGE-DNF in computing implications, and 166ms faster for computing mutual exclusions.

The results for group computation are not significant enough to reject the null hypothesis for this component. However, we see that the mean computation time is almost 4 seconds faster on FGE-DNF than FGE-FCA for computing OR-groups.

5.9.10 Threats to Validity

External Threats A potential threat to validity is that this input may not be representative of realistic scenarios involving feature model synthesis. In the case of feature model operations (Scenario 3), the inputs derived from existing feature models are indeed representative. We partially mitigate this threat by using both randomly generated input, as well as input derived from real world feature models from the SPLOT repository to evaluate our algorithms. While this input is not sampled from real synthesis scenarios, we believe that these inputs are sufficiently different to be representative of the input used for feature model synthesis.

Internal Threats The input formulas to our algorithms were derived from existing feature models in a repository. As a result, the constraints in the derived formulas may mainly consists of feature model constraints. Mendonca et al. found that the SAT-based analysis of feature models is generally easy, depending on the properties of the cross-tree constraints [MWC09]. This threat to validity is mitigated by the distribution of cross-tree constraints in our SPLOT model dataset. The cross-tree constraint ratio (CTCR)–i.e., the number of features participating in cross-tree constraints divided by the number of features—ranging from 0% to 100%. The dataset consists of input with varying levels of complexity.

We also used the computation time as the evaluation metric between the algorithms. A threat to validity exists where the computation time is affected by the overhead of JVM and library initialization. We mitigate this thread of validity by recording the times for the computing the individual components of the feature graph (i.e., implication graph, mutex graph, and OR-group computation). The remaining operations of FGE (e.g., AND-group computation) are independent of FGE algorithms.

5.9.11 Conclusions

We presented two efficient, scalable feature model synthesis algorithms in this chapter: FGE-CNF and FGE-DNF. We evaluated FGE-CNF against FGE-BDD—a BDD-based implementation, and evaluated FGE-DNF against FGE-FCA—a FCA-based synthesis technique by Ryssel et al. [RPK11].

We use two datasets for our evaluation: a dataset with input derived from the Linux variability model and 267 models gathered from the SPLOT model repository, and a dataset of 20 generated feature models with 3-CNF constraints. We use both datasets for our evaluation on FGE-CNF. We generated configurations for SPLOT models with less than 100,000 configurations for our evaluation on FGE-DNF. The input from the generated 3-CNF feature models were more difficult for both FGE-BDD and FGE-CNF. FGE-BDD timeouted in most cases while FGE-CNF was able to complete computation for 12 of the 20 models. Our evaluation showed that FGE-CNF was significantly faster than FGE-BDD. FGE-DNF and FGE-FCA were generally comparable for the real world model dataset. However, there were five models where FGE-DNF was significantly faster than FGE-FCA. These models had many features that were mandatory with respect to the root resulting in a large search problem for group computation.

Chapter 6

Feature Model Synthesis

In the previous chapter, we introduced the FEATURE-GRAPH-EXTRACTION algorithm for synthesizing a feature graph given input as a propositional formula. A feature graph encapsulates all feature diagrams that are entailed by the input formula. However, the feature graph is not a proper feature model—the hierarchy is a DAG instead of a tree, and feature groups can overlap.

In this chapter, we describe a semi-automated procedure, called FEATURE-TREE-SYNTHESIS, that abstractly selects the most suitable feature diagram from the feature graph. In practice, both FEATURE-TREE-SYNTHESIS and FEATURE-GRAPH-EXTRACTION are intertwined. FEATURE-TREE-SYNTHESIS supplements FEATURE-GRAPH-EXTRACTION by providing a semi-automated procedure for determining a distinct feature hierarchy. Our procedure uses the input dependencies as a guide for the configuration semantics and a textual similarity measure to approximate the domain semantics. Given a feature, the user selects its parent given a list of implied features that are ranked by their similarity to the selected feature. In a practical synthesis scenario, the input dependencies may be incomplete, i.e., constraints may be missing in the input. Our procedure deals with incomplete dependencies by providing a second list that ranks features solely using the textual similarity measure. This algorithm, combined with FEATURE-GRAPH-SYNTHESIS in Chapter 5, form a complete feature model synthesis algorithm.

Chapter Organization In Section 6.1, we introduce the motivation and context of our synthesis algorithm. Section 6.2 gives an overview of the procedure with an examples and in Section 6.3 describes our procedure. Section 6.4 describes the evaluation of our procedure on the Linux, eCos, and portions of the FreeBSD kernel.

Publications Material from this chapter was published in [SLB+11].

6.1 Introduction

Variability-rich software systems, such as FreeBSD, do not have a feature model and could benefit from having one. FreeBSD describes features and dependencies in an ad-hoc manner—features are scattered in documentation and dependencies are hidden in code. Such projects would benefit from having an explicit feature model instead. Unfortunately, constructing a feature model is both time and cost-intensive. Building the feature hierarchy in particular, requires substantial effort from a modeler. This task requires the modeler to review feature descriptions and dependencies to determine which dependencies to model in the hierarchy, and which to defer to cross-tree constraints. FreeBSD has 1203 features; constructing a feature model for a project of this size would require tremendous time and effort. Furthermore, the difficulty is compounded when the modeler lacks a complete set of dependencies. In this case, the dependencies could be uncovered by examining supplemental data such as feature names and feature descriptions. This may require the modeler to sift through the text of potentially hundreds of features in order to determine the correct placement for a single feature. Even with a complete set of dependencies, selecting the right parent for a feature is still challenging—a single feature may depend on over a hundred others as we have observed in the variability models of the Linux and eCos kernels.

We present a tool-supported approach for reverse engineering feature models called FEATURE-TREE-SYNTHESIS. The key challenge is the construction of the feature diagram. This task reduces to the selection of a parent for each feature. We present heuristics for identifying the likely parent candidates for a given feature. Our heuristics significantly decrease the number of features that a user has to consider from potentially thousands to only a handful—typically five or less, as shown by our experiments. We also provide automated procedures for finding feature groups, implies and excludes edges. If the set of input dependencies are complete, the final feature model is entailed by the input dependencies.

FEATURE-TREE-SYNTHESIS require a list of feature names, supplementary descriptions, and a propositional formula describing its dependencies. Feature names and descriptions can be extracted from documentation, preprocessor symbols or code comments. For our evaluation on the FreeBSD kernel, we extracted the input data by analyzing Makefiles, preprocessor declarations, and documentation, using a combination of generic and custom extraction tools.

Due to the complexity, size and nature of most software projects, it is likely that the extracted feature dependencies and descriptions are incomplete. Our heuristics accommodate this incompleteness by leveraging two sources of data that complement one another—when dependencies are incomplete, the feature descriptions are used to identify parent candidates and vice versa.

We evaluate the effectiveness of our procedures by comparing the results of our heuristics to the reference feature models of the Linux, eCos and FreeBSD kernels. Linux and eCos both have an existing reference feature model [BSL+10b]. The input dependencies and supplementary feature descriptions were jextracted from the reference models themselves. For FreeBSD, we manually constructed a reference feature model for a subset of features after domain analysis. The evaluations show that, for 76% of features in Linux and 79% in eCos, the correct parent is in the top five parent candidates returned by our heuristics. In contrast to Linux and eCos, the input set of dependencies for FreeBSD is incomplete, and thus, we consider two separate results for FreeBSD: (1) for 84% of the features whose parent dependency is present, the correct parent is in the top two candidates; (2) for 75% of the remaining features, the correct parent is in the top or 3% of all 1203 features. Finally, our procedure automatically recovers all feature groups, as presented in the reference models for Linux and eCos. We assume that the modeler settled on the same hierarchies as these models in our evaluation. With the incomplete dependencies of FreeBSD, we were still able to retrieve one of the three feature groups.

The contribution of this work is twofold. On the practical side, we present heuristics and procedures for reverse engineering feature models. Although reverse engineering feature models from logic formulas [CW07] and descriptions [ASB+08, NE08b] were considered before in separation, the main contribution of our approach is that it combines both sources of information together. This combination is desirable since, as our evaluation shows, the two sources are complementary. Also the procedures of [CW07] and [ASB+08, NE08b] are not complete, in the sense that the former cannot recover parents which are not direct dependencies, while the latter suggests only a single hierarchy that is unlikely the desired one. We also reverse engineering of large-scale feature models on input derived from the Linux and eCos kernel showing that our approach and procedures scale. On the theoretical front, we describe how both configuration semantics and domain semantics relate to feature hierarchy.

$$
\begin{array}{rcrcl}
 & & (\text{acpi} & \rightarrow & \text{acpi\_system} \wedge \text{pm}) \\
 & \wedge & (\text{acpi\_system} & \rightarrow & \text{acpi}) \\
 & \wedge & (\text{cpu\_freq} & \rightarrow & \text{pm}) \\
(1) & \wedge & (\text{cpu\_freq} & \rightarrow & \text{powersave} \vee \text{performance}) \\
(2) & \wedge & (\text{cpu\_hotplug} & \rightarrow & \text{powersave}) \\
(3) & \wedge & (\text{cpu\_hotplug} & \rightarrow & \neg\text{performance}) \\
 & \wedge & (\text{cpu\_hotplug} & \rightarrow & \text{acpi} \wedge \text{cpu\_freq}) \\
 & \wedge & (\text{powersave} & \rightarrow & \neg\text{performance}) \\
 & \wedge & (\text{powersave} & \rightarrow & \text{cpu\_freq}) \\
 & \wedge & (\text{performance} & \rightarrow & \text{cpu\_freq}) \\
 & \wedge & (\text{powersave} \wedge \text{acpi} & \rightarrow & \text{cpu\_hotplug})
\end{array}
$$

(a) Dependencies

pm Power management, CPU and ACPI options
acpi Advanced Configuration and Power Interface support
acpi_system Enable your system to shut down using ACPI
cpu_freq CPU frequency scaling
cpu_hotplug Allows turning CPU on and off
powersave This CPU governor uses the lowest frequency
performance This CPU governor uses the highest frequency

(b) Features and descriptions

Figure 6.1: Example input for feature model synthesis

6.2 Overview

In this section, we demonstrate how our procedures assists a user for synthesizing a feature model. Figure 6.1 shows a set of dependencies as a formula, feature names, and descriptions that we use as input data.

Our procedure reduces the synthesis process of building the feature hierarchy, finding feature groups, and inserting implies and excludes edges in a sound and complete manner, to just the first step: building the feature hierarchy. The remaining steps are automated.

Recall from Figure 3.1 that feature model synthesis takes as input, (1) a set of dependencies or configurations, (2) features, and (3) supplemental information. In the case of FEATURE-TREE-SYNTHESIS, we use feature names and feature descriptions as supplemental information.

Feature model synthesis itself can be broken down into three components: DAG hierarchy recovery, Group and CTC recovery, and tree hierarchy selection. FEATURE-TREE-SYNTHESIS is a technique that addresses the tree hierarchy selection component. We follow the early hierarchy selection workflow depicted in Figure 3.3b, where tree hierarchy selection occurs before group and CTC recovery. This workflow reduces the search space for feature groups making for a more efficient synthesis procedure. FEATURE-TREE-SYNTHESIS synthesizes a feature tree by applying heuristics based on the feature names and descriptions with the assistance of a user. The technique supports the user in building the feature tree itself by providing suggestions for parents given a feature.

The DAG hierarchy recovery is done by using the same technique as in FEATURE-GRAPH-EXTRACTION (Chapter 5). An implication graph among features is constructed forming a DAG. This DAG, along with feature descriptions and user input, is used as input to FEATURE-TREE-SYNTHESIS. The key challenge of building the feature hierarchy is the selection of a parent for each feature. This process requires an understanding of meaning of the feature and its relationships with other features. As a result, the hierarchy building process is inherently *interactive* and requires the input of a domain expert. FEATURE-TREE-SYNTHESIS presents two lists of *parent candidates* that are ordered such that the most likely parents are placed near the top of the list. The parent candidate lists reduce the amount of information that a domain expert would have to process by identifying the most likely parent candidates.

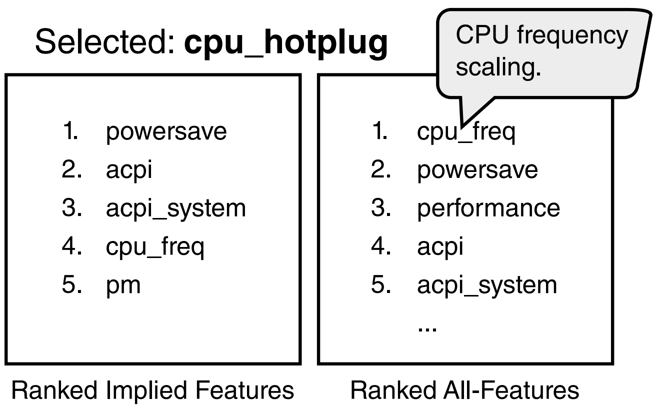


Figure 6.2: Mockup of the two parent candidate lists

The first list is the *ranked implied features* (RIFs)—a sorted list of features that a given feature implies. Implied features are the primary criteria when deciding a parent—the semantics of feature models state that a child implies its parent. However, a feature may imply more than one other feature. This is where a ranking heuristic is applied to sort the implied features by their similarity to the selected feature, placing the most likely candidates at the top of the list.

The second list is the *ranked all-features* (RAFs)—all features sorted by their similarity to a given feature. The RAFs is a complete ranking, but is typically less accurate than the RIFs. It can be reviewed in the case the input dependencies are incomplete and the user cannot find an appropriate parent in the RIFs. In this case, the RAFs is useful for identifying potential parents where an implication from the selected feature may be missing due to incompleteness of available dependency information. We describe the details of the hierarchy building procedure in Section 6.3.1.

As an example, assume that the user is selecting a parent for cpu_hotplug. We would present its parent candidates as in Figure 6.2. There are five features here that are implied features of cpu_hotplug with powersave at the top position, and the actual parent cpu_freq is at the fourth position. If the dependencies are incomplete and the user cannot find an appropriate parent in the left list, the right list can be reviewed. Later, we show that the best candidates for parents are typically highly ranked in both lists.

Once the user decides on a feature hierarchy, feature groups and cross-tree constraints are recovered. Using the input features and dependencies or configurations, a mutex graph is computed to identify MUTEX-groups. OR-groups are identified directly using

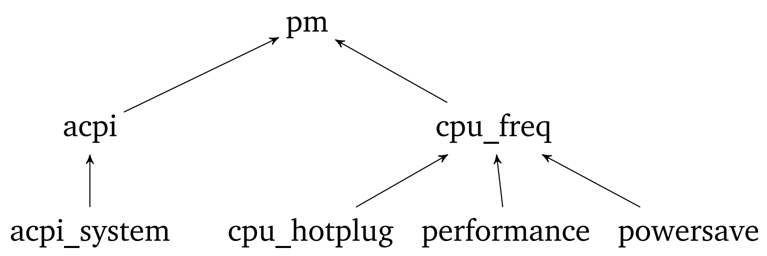


Figure 6.3: A feature hierarchy for the input in Figure 6.1

dependencies or configurations. The feature hierarchy constructed through FEATURE-TREE-SYNTHESIS is used to reduce the search space for OR-groups. XOR-groups are identified by using the identified MUTEX-groups, and OR-groups. Alternatively, like in FEATURE-GRAPH-EXTRACTION, an additional check using the input dependencies can be performed to the set of MUTEX-groups to identify XOR-groups.

The user reviews the identified feature groups and select the ones that should be retained in the feature diagram. Any feature groups not retained in the diagram are kept as part of the cross-tree formula. For example, if we assume that the hierarchy in Figure 6.3 is chosen, our tooling will detect two feature groups: a MUTEX-group between cpu_hotplug and performance and an XOR-group between performance and powersave. The user selects one of the groups to keep in the diagram, relegating the other to the cross-tree formula.

Finally, mandatory features, implies and excludes edges are automatically discovered and added to the feature diagram. For example, assume that the user decides on the hierarchy in Figure 6.3 and implication (2) is omitted from the dependencies in Figure 6.1a. Our procedure can still detect an implies edge from cpu_hotplug → powersave since it can be derived from implications (1) and (3). Now that the diagram is finished, further constraints are added to the cross-tree formula to make the resulting feature model sound—all legal configurations of the feature model are legal configurations with respect to the input dependencies. All these steps relies on BDDs or SAT-based reasoners and thus, are independent from the syntactic structure of the dependency constraint. In fact, any of the FEATURE-GRAPH-EXTRACTION variants in Chapter 5 can be used for the automated steps in the workflow. The contribution of FEATURE-TREE-SYNTHESIS is a heuristic-based technique for deriving a single feature tree.

Furthermore, if the user assumes the dependencies are complete, then only the RIFs needs to be reviewed by the user. The RAFs are no longer needed because all possible alternatives for parents are contained in the RIFs.

Our procedures can be integrated into existing feature model editors, in the style of the model builder by Janota et al. [JKW08], to equip them with reverse engineering capabilities and to allow modelers to make parent and group decisions with a graphical interface. However, we do not advocate any specific user interface design at this point and leave this to future work.

6.3 Feature Tree Synthesis

FEATURE-TREE-SYNTHESIS assumes three kinds of inputs: a set of feature names $\mathcal{F}$, feature descriptions $\mathcal{D}$, and feature dependencies φ. The ordering heuristics for parents ignore the order of words in the descriptions, so $\mathcal{D}$ is defined as a mapping assigning a multiset of words to each feature in $\mathcal{F}$. A *multiset* is a pair (X, c), where X is a set and $c : X \rightarrow \mathbb{N}_1$ is a mapping from a word to its occurrence in the feature description. Given a feature f we write $\mathcal{D}_1(f)$ to refer to the set of words in f's description and for a word $w \in \mathcal{D}_1(f)$, $\mathcal{D}_2(f)(w)$ denotes the number of occurrences of w in f's description. Finally, the input is specified as a propositional formula φ over $\mathcal{F}$.

6.3.1 Building the Feature Hierarchy

Implication Graph

The basis of the feature hierarchy is an implication graph. We use the FEATURE-GRAPH-EXTRACTION algorithm (FGE) from Chapter 5 to synthesize an implication graph given the propositional formula φ over $\mathcal{F}$. Any one of the FGE variants is capable of synthesizing an implication graph. In this description of FEATURE-TREE-SYNTHESIS, we rely on FGE-CNF since it was the most scalable of the FGE algorithms.

An implication graph is initially *transitively closed* due to the transitivity of implications. If $w \rightarrow u$ and $u \rightarrow v$ then $w \rightarrow v$. The *transitive reduction* of an implication graph is the subgraph not containing the aforementioned transitive implications, except for cliques. The transitive reduction can be computed using known algorithms [AGU72, CW07]. We denote the transitive reduction of a graph G by G_{R} and edges remaining in G_{R} as *direct implications*. Figure 6.4 shows the transitively reduced subgraph in thick edges.

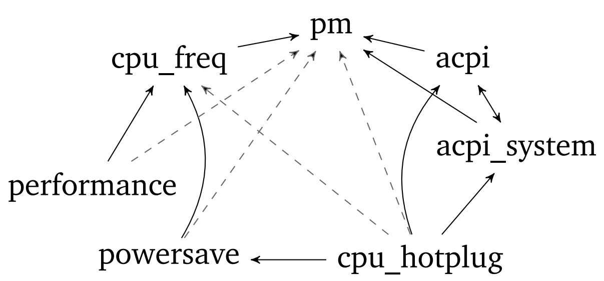


Figure 6.4: Implication graph of the dependencies in Figure 6.1 where the transitively reduced graph consists of the black edges, and the transitively closed graph consists of both the black edges and the dashed edges

Next, $\mathrm{E}(G)$ denotes the set of edges in a graph G. For a feature f write $\mathrm{I}_f(G)$ to denote features implied by f (so heads of edges outgoing from f). Then $\mathrm{I}_f(G_R)$ gives the directly implied features of f in G.

Identifying Parents

The key mechanism of Feature-Tree-Synthesis is its similarity heuristic for ranking parent candidates. The parent ranking heuristics leverages two complementing forms of data: dependencies and descriptions. The dependencies describe the configuration semantics of the resulting feature model and the feature descriptions are used to approximate its domain semantics.

Given a parent candidate p and the selected feature s, we define the similarity function $\delta(p,s)$ to return the sum of the *inverse document frequency* (IDF) of the words shared between the descriptions of p and s, weighted by the number of occurrences of each shared word in p's description (Equation 6.1):

$$\delta(p,s) = \sum_{w \in \mathcal{D}_1(p) \cap \mathcal{D}_1(s)} \mathrm{idf}(w) * \mathcal{D}_2(p)(w) \tag{6.1}$$

$$\text{where } \mathrm{idf}(w) = \log \frac{|\mathcal{F}|}{|\{f : w \in \mathcal{D}_1(f)\}|}$$

Essentially, given the selected feature s, this measure ranks features according to the number of words that they share. The measure uses the IDF to give less weight to common domain words such as ‘Linux’, ‘eCos’, ‘choose’, or ‘select’. These words are not typical stop words according to standard natural language processing tools, but they also do not contribute to identifying commonalities between two feature descriptions.

We use the similarity function δ to induce a ranking order on features. Given a selected feature s we define two strict partial orders: $>_s$ and $>_s^{\mathbf{P}}$. In the first, $>_s$, features are ranked strictly by their description similarity to s:

$$a >_s b \text{ iff } \delta(a,s) > \delta(b,s) \tag{6.2}$$

The second partial order $>_s^{\mathbf{P}}$ prioritizes directly implied features of s over all other implied features. This prioritization is based on our observation that in Linux 88% of parents in the model are directly implied features. The partial order is defined as:

$$a >_s^{\mathbf{P}} b \text{ iff } \begin{cases} a \in \mathrm{I}_s(G_R) \wedge b \notin \mathrm{I}_s(G_R) & \text{or} \\ a \in \mathrm{I}_s(G_R) \wedge b \in \mathrm{I}_s(G_R) \wedge a >_s b & \text{or} \\ a \notin \mathrm{I}_s(G_R) \wedge b \notin \mathrm{I}_s(G_R) \wedge a >_s b \end{cases} \tag{6.3}$$

Finally, we can define the two lists that make the parent candidates: The RIFs ranks only the implied features of f using the prioritizing order while the RAFs ranks all features using the non-prioritizing order.

> **Definition 6.1.** Given a feature s and an implication graph G, $\mathsf{RIF}(s)$ is the list created by sorting features in $\mathrm{I}_s(G)$ in decreasing order with respect to $>_s^{\mathbf{P}}$ (largest rank first). $\mathsf{RAF}(s)$ is the list of features in $\mathcal{F}$ sorted in decreasing order with respect to $>_s$. The orders are made total by using alphabetical ordering when two or more features have the same similarity.

Let's construct the RIFs and RAFs list for the feature in our example: cpu_hotplug. The first list, RIFs, requires the set of implied features. If we examine the implication graph for our input (Figure 6.4), we see the directly implied features of cpu_hotplug are: acpi, acpi_system, powersave. The indirectly implied features are cpu_freq and pm. Applying our ordering heuristic to these features, we see that cpu_hotplug shares the word cpu with cpu_freq in their names, and CPU appears in the descriptions of powersave and pm. Since the RIFs prioritize direct implications, the resulting RIFs ordered list is:

$$\mathsf{RIF}(\mathsf{cpu\_hotplug}) = \langle \mathsf{powersave}, \mathsf{acpi}, \mathsf{acpi\_system}, \mathsf{cpu\_freq}, \mathsf{pm} \rangle$$

The RAFs list applies the ordering heuristic to all features in the input. This list includes the feature performance that was omitted from the RIFs since it is not implied by cpu_hotplug. If we apply the feature similarity measure to this list, we get the following ordering:

$$\mathsf{RAF}(\mathsf{cpu\_hotplug}) = \langle \mathsf{cpu\_freq}, \mathsf{performance}, \mathsf{pm}, \ldots \rangle$$

The user chooses parents for every feature by examining RIFs and RAFs, forming a set of directed child-parent edges $E \subseteq \mathcal{F} \times \mathcal{F}$. These edges may not form a single tree when there is no common top-level ancestor. In this scenario, we insert an additional root feature to join together the forest to form a single tree.

6.3.2 Feature Groups and Cross-Tree Constraints

Once the hierarchy is built, feature groups and cross-tree constraints are detected with the FEATURE-GRAPH-EXTRACTION algorithm (Chapter 5). In Chapter 5, feature groups were identified among all implied features since a specific hierarchy is not assumed. We describe FEATURE-TREE-SYNTHESIS according to the early hierarchy selection workflow where a hierarchy is first constructed prior to detecting feature groups (Figure 3.3b). This workflow reduces the search space for identifying groups since only feature groups among siblings in the feature hierarchy have to be searched. If there are overlapping feature groups among sibling features, the user will have to select a non-overlapping subset of groups. We leave heuristics for ranking feature groups to future work. FEATURE-TREE-SYNTHESIS could also be applied to a feature graph constructed with a late hierarchy selection workflow (Figure 3.3a)—the disadvantage of this workflow being that unnecessary feature groups may be computed.

Incomplete Dependencies

FEATURE-TREE-SYNTHESIS considers incomplete dependencies by relying purely on the textual similarity measure in the form of the RAFs list. In this section, we examine how incomplete dependencies affect feature group detection. A MUTEX-group with members $f_1, \ldots, f_k$ is detected if there exists a clique between $f_1, \ldots, f_k$ in the mutex graph. A clique in the mutex graph requires each member to have an exclusion to every other member. If any exclusions between the members are missing in the case the dependencies are incomplete, a MUTEX-group with k members will be detected as one with less than k members.

XOR-groups, on the other hand, contain two dependencies. First requirement is that an underlying MUTEX-group exists between the group members. Second, the XOR-group requires an implication from the group's parent to its members (Equation 2.7). If either the first dependency is incomplete or if the second dependency is missing, then the XOR-group will be detected simply as a MUTEX-group of equal or lesser size.

6.4 Experimental Evaluation

We implemented FEATURE-TREE-SYNTHESIS using FGE-CNF (Chapter 5) to compute the implication graph, mutex graph, and feature groups. The natural language similarity measure uses the LingPipe toolkit[1]. We evaluate our procedures on input from Linux, eCos and FreeBSD. For Linux and eCos, we extract our input data from their existing reference feature models [BSL+10b]. This gives us two samples with complete dependencies and descriptions. FreeBSD, on the other hand, does not have a reference feature model. For this project, we extract data extracted from the FreeBSD codebase manually, giving us a sample with incomplete dependencies and partial descriptions. We believe that FreeBSD is representative of projects that will use our synthesis algorithms for reverse engineering. Since FreeBSD lacks a reference model, we created one manually by first performing domain analysis and creating an ontology for a subset of features. We then extracted a feature model from the ontology.

Our evaluation criterion is to check if for every feature, its parent in the reference feature model is one of the top parent candidates in the RIFs and RAFs. We consider the parent-child relations in the reference feature models to be the best choices possible because these models were built manually over many years by their respective development communities.

FEATURE-TREE-SYNTHESIS handles incomplete input by relying strictly on the textual similarity measure. We measure the effect of incomplete dependencies and descriptions by progressively removing dependencies and descriptions from the Linux and eCos data. Our evaluation shows that prioritizing direct implications has a significant impact on the effectiveness of our procedures with incomplete descriptions. Finally, we evaluated our procedure for recovering feature groups in the presence of complete and incomplete data.

[1] http://alias-i.com/lingpipe/

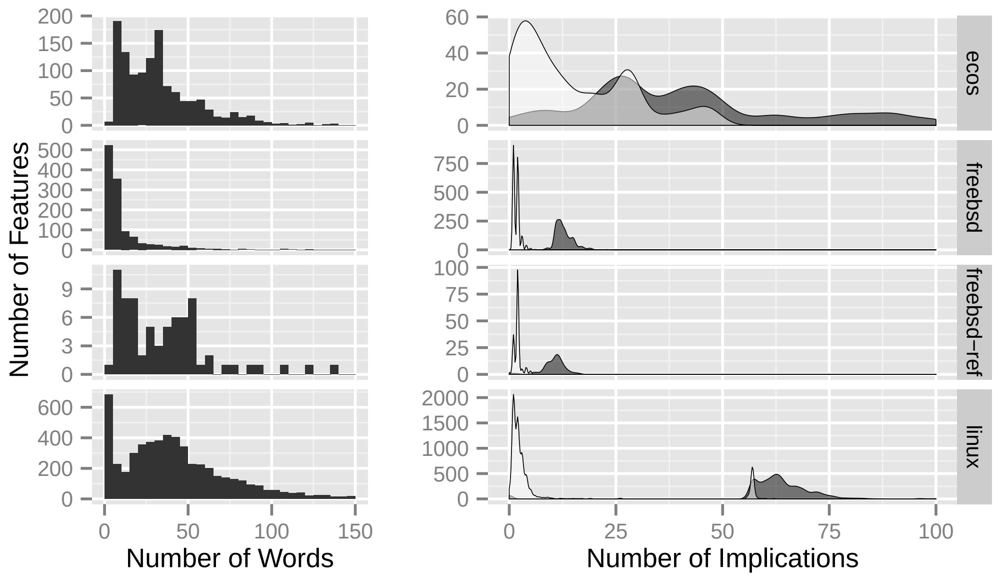


Figure 6.5: Characterization of descriptions, transitive implications (dark grey) and direct implications (white) for eCos, FreeBSD, the FreeBSD reference model, and Linux

6.4.1 Input Data Characteristics

The evaluation dataset consists of the x86 variability models of Linux v2.6.28.6 and i386pc variability model of eCos v3.0. We extract dependencies applying a translation of their formal semantics to propositional formulas [SB10, BS10]. Feature names and descriptions were extracted directly from the reference models themselves. We have placed the translation tools online as open-source projects[2,3].

Figure 6.5 characterizes the input data for the variability models of Linux, eCos, and FreeBSD. Linux has 5321 features, while eCos has 1245, and as shown in Figure 6.5, the distribution of number of words are only slightly different. The majority of features in eCos have 10 to 40 words. Linux, on the other hand, has a large number of features with no descriptions, while the rest have roughly 20 to 60 words. The distributions of direct and transitive implications are significantly different. Almost all features in Linux have from 60 to 80 transitive implications, while in eCos the variation is much

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

larger. For direct implications, Linux's features have from 0 to 10 implications, and again, eCos has a larger variety.

Unlike Linux and eCos that are configured with a graphical configurator, configuring the FreeBSD's kernel involves creating a text file with the selected features, such as devices and CPU options, and their values. Various boilerplate templates are available to the user as default configurations. There is no explicit description of legal combinations of features. For FreeBSD, we extracted features using hand-crafted parsers for analyzing configuration templates in the codebase. Feature descriptions were extracted in a semi-structured manner, with heuristics-based fuzzy parsers that identify text patterns. When possible, we would further spplement the descriptions with manual pages.

Dependencies were extracted from various sources. We manually derived around 100 dependencies that were described in feature descriptions, but the majority of dependencies were extracted from source code. We examined the codebase to find statements relevant for extracting feature dependencies. For example, dependencies between device drivers were specified by FreeBSD-specific preprocessor macros, making such dependencies easy to extract. However, most constraints had to be extracted by using a more comprehensive static analysis infrastructure. This infrastructure was built specifically for this project. The analysis infrastructure derives constraints by analyzing C source files and exploiting three types of information: `#error` preprocessor macros (*build error analysis*), the location of feature references (*feature liveness analysis*) and the definition and use of identifiers (*def/use analysis*).

From our experience with extracting dependencies from FreeBSD, we learned that part of the variability analysis can be implemented as an automatic component. However, there was significant manual work needed to capture project-specific patterns from the build system. The project-specific component once constructed, can be beneficial for other applications such as iteratively checking the dependencies of a project against its feature model as the project evolves. We estimate it took one person, one month of effort to build our project-specific variability analysis for FreeBSD.

FreeBSD Reference Model

In order to assess the quality of the model synthesized by FEATURE-TREE-SYNTHESIS, we created a reference feature model of 90 features by manually analyzing the FreeBSD kernel artifacts. We started by performing domain analysis on documentation and architecture to produce an ontology[4]. The ontology comprises of 192 features with several domain-specific relationships. The reference feature model was derived by taking the parts of the ontology that we felt were most developed and correct. The feature hierarchy was built by traversing generalization and composition relationships [CKK06]. The resulting feature model mainly covers technical aspects of the kernel, such as tracing, monitoring and debugging. Structurally, 21% of its features are mandatory; 24% participate in cross-tree constraints; and three XOR-groups that bundle 12% of the features. Creating the ontology and deriving the feature model took about two person weeks.

Figure 6.5 shows that the number of implications per feature in the FreeBSD reference model is representative of the implications of all features in FreeBSD. However, the reference model has a larger proportion of long descriptions (over 10 words) to short descriptions (less than 10 words) than all features in FreeBSD; in this aspect, it is more similar to Linux and eCos.

6.4.2 Effectiveness of Parent Heuristics

We evaluate our parent heuristics on complete input with Linux and eCos and on incomplete input with Linux, eCos and FreeBSD.

For the RIFs, our evaluation criterion is whether the reference parent as defined in the reference model, appears in the top five positions of our RIFs list. We feel the top five results are a reasonable number for a user to review and to select the correct choice. For the RAFs, we calculate the percentage of all features users will need to examine to have a 75% chance of finding the feature's parent. This measure the effectiveness of only the ranking heuristic, in the absence of any dependencies.

Complete Input For each feature, we record the position of the reference parent in the RIFs list. We found that 76% of features in Linux and 79% of features in eCos have their reference parent within the top 5 results.

These results show that the heuristics are generally successful at identifying the correct parent. However, in our evaluation subjects, a significant number of features are at the root. Since these features have no parent, the RIFs and RAFs are not applicable, thus

[4]http://code.google.com/p/variability/wiki/FreeBSDOntology

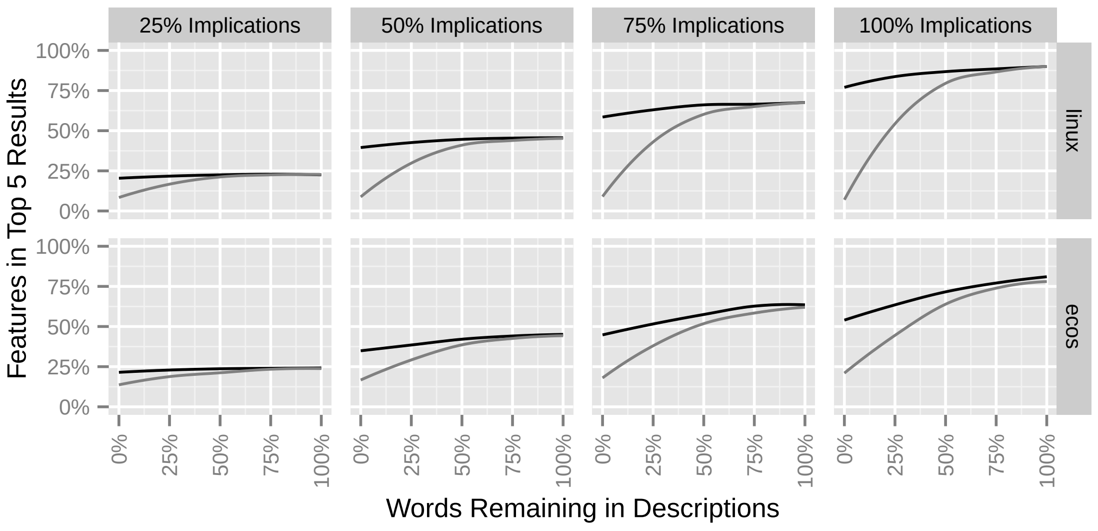


Figure 6.6: Robustness of RIFs for the *prioritizing* (black) and *non-prioritizing* (gray) orders under complete and incomplete data

skewing the statistics to our disadvantage. The lists do not contain the root because nesting under root (or on top-level) is qualitatively a different decision than nesting features anywhere else. Our tools do not guide the reader in any way to nest under root, but we consider this decision to be much easier to make then detailed nesting of small granularity features deep in the hierarchy. If we only consider features that are not children of root, we observe that as many as 90% of them in Linux and 81% in eCos have their reference parent within the top five results. For this reason, the following diagrams (Figures 6.6, 6.7) omit top-level features.

Incomplete Input The RAFs list, which ranks all features, is used to identify potential parents when the RIFs do not contain the proper parent. Common to both the RIFs and RAFs is our textual similarity heuristic that uses feature names and descriptions of features—as descriptions shrink in size, so does the effectiveness of our heuristic.

We evaluate the effects of incomplete data and the robustness of our RIFs by randomly selecting subsets of implications and words from descriptions for Linux and eCos. For RAFs, we randomly select just words from the descriptions since implications are not used.

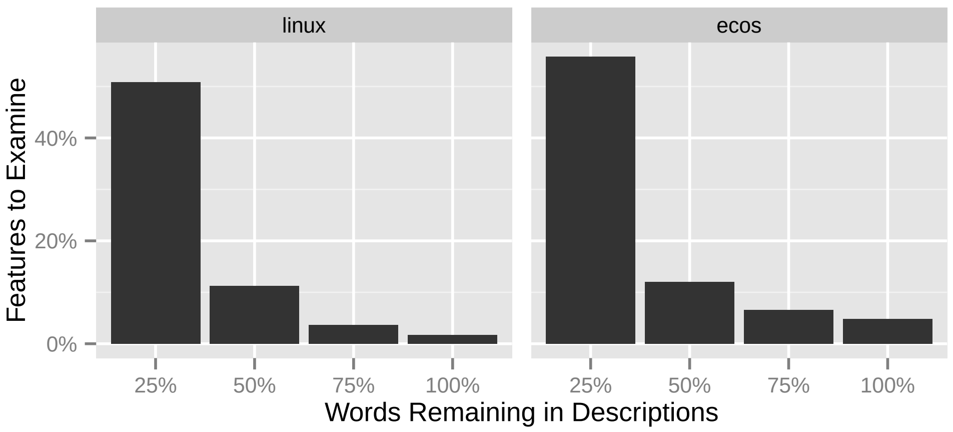


Figure 6.7: The top RAFs needed for a user to have a 75% chance of finding the reference parent under complete and incomplete descriptions

Results for the RIFs Figure 6.6 shows how the RIFs performs as we reduce the number of implications and words of descriptions for the Linux and eCos data. We created sets of samples where we progressively removed random implications and words, forming sets with 25%, 50%, 75% and 100% of the original implications and words from descriptions. We repeated the experiment 10 times for each combination to assure robustness of results.

The results linearly degrade as we remove dependencies. We observed that larger descriptions significantly improve results, particularly from 0% to 50% descriptions where the gain is most significant. The figure also shows the effect of our order where direct implications are prioritized (in black) over the non-prioritizing variant (grey). We see that the prioritizing order is more effective than the non-prioritizing variant overall. Furthermore, the prioritizing order is particularly effective on Linux where the prioritized results are still relatively high even as the descriptions approach zero. The prioritizing order is able to compensate for smaller descriptions.

On FreeBSD, we found that 35% of features did not have an implication to their reference parent in their input dependencies. As a result, these features do not have their reference parent in their RIFs. For the remaining 65%, the reference parent is contained in the top 5 results of the RIFs for all features.

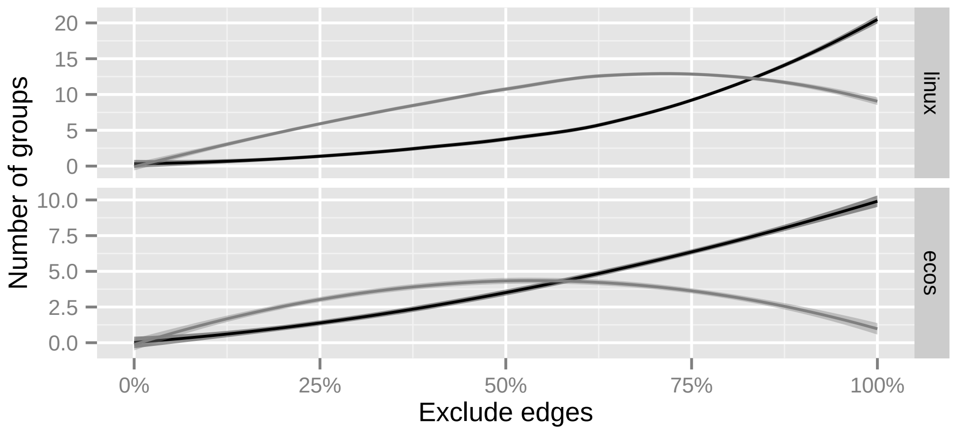


Figure 6.8: Reference groups detected as *xor*- (black) or *mutex*-groups (grey) under complete and incomplete exclusions

Results for the RAFs Figure 6.7 shows the amount of a project's total features a user would need to examine to have a 75% likelihood of finding the reference parent. For this experiment, we formed datasets containing no descriptions, 25%, 50%, 75% and 100% of randomly selected words from descriptions. We measure the robustness of our ranking heuristic with respect to the size of descriptions.

With 50% of words remaining in descriptions, the user needs to examine approximately 10% of all features in Linux and eCos to find the correct parent. As descriptions increase in size, we see that the user needs to examine fewer and fewer features, reaching only 3% of all features in Linux and 6% in eCos when we have complete descriptions (100%). FreeBSD has results similar to Linux, with the user needing to examine 3% of all features to attain a 75% likelihood of finding the reference parent.

6.4.3 Feature Groups

Our technique was able to identify all feature groups except one. One XOR-group was not found due to the presence of a dead feature in its members. We were unable to find the OR-group since our procedure lacks support for finding such groups.

The reference feature model of eCos had only a single MUTEX-group, 11 XOR-groups and no OR-groups, with 44 features participating in groups. The MUTEX-group and 10 of the

XOR-groups were discovered by our procedure. The XOR-group that was not detected was in fact, dead. The group required a package that was not present in the eCos i386 model that we analyzed.

The reference FreeBSD feature model had three XOR-groups. We correctly identified one group in its entirety. Parts of a second group were detected as MUTEX-groups and a third group was not detected at all due to the incompleteness of our dependency data. The FreeBSD data was incomplete as we saw in the evaluation for the hierarchy building—roughly a third of features did not imply their parents of the reference model.

Incomplete Input Feature groups rely on exclusions in the mutex graph to determine its type, size and members. Incomplete dependencies affects the detection of feature groups. Here, we evaluate the effect of incomplete exclusions on the Linux and eCos data. Figure 6.8 shows the number of feature groups from the respective reference models that are detected as an XOR-group, or as a MUTEX-group of equal or smaller size depending on the completeness of exclusions. We randomly selected exclusions ranging from 0 to 100%, with 100% being all present. Each point is an individual sample and the lines indicate the fitted curve across all the sample points.

As the number of exclusions increase in the input, the number of MUTEX and XOR-groups detected also increase, with the former growing at a much faster rate. In the presence of incomplete dependencies, an XOR-group may be detected as a MUTEX-group of equal or smaller size. An XOR-group is only detected when all exclusions among its members are present. In this sense, the XOR-group detection is very susceptible to incomplete input. We observe this effect at roughly 70% of exclusions in Linux and 50% in eCos—the number of MUTEX-groups starts a downward trend and the XOR-groups begin to grow at an increasing rate. This means that the input for our procedure should contain at least about 50% of exclusion dependencies to warrant that group recovery reasonably distinguishes between the two kinds of groups.

We also observe different curves between the two systems due to their characteristics—Linux has 30 groups over 526 features while eCos has 12 groups over 44 features. With the larger number of features participating in groups in Linux, we see it takes more exclusions in Linux before XOR-groups are detected when compared to eCos.

6.4.4 Threats to Validity

External threats

Our procedures assume that feature names, descriptions, and dependencies can be extracted from a project. Our evaluation on FreeBSD shows that it is possible to extract such input from existing software projects. Crafting the project-specific component for FreeBSD required significant work. However, the effort required for this component may vary depending on the project.

We evaluated our procedures on the Linux, eCos and FreeBSD projects. While all three of these projects are in the operating systems domain, the features and their variability models are of considerable complexity and size, and differ significantly in their input characteristics. Furthermore, the input to our algorithm is a set of features, dependencies, and feature descriptions; we believe that with these general inputs, the procedures can be applied to projects in other domains.

As stated before, feature names in Linux and eCos often reflect the hierarchy of the reference models. Thus, a potential threat to validity is that we obtain good results for these systems because the reference hierarchy is re-discovered by our similarity metric from the feature names. However, we found that our similarity measure performs similarly well on FreeBSD, even though the system does not come with a feature model and their feature names follow a different convention.

Internal threats

We evaluate our procedure on FreeBSD using a reference feature model that a colleague, Thorsten Berger, created. This creates a threat of potential bias, since the author knew the procedures that were to be evaluated against this model. Also, it is possible that the reference model is different from what a domain expert would create. To address these problems, we used an entirely different approach to build the feature model [CKK06], that required building an ontology first. The ontology represents the domain in more detail than a feature model would represent, effectively forcing the modeler to become an expert in the modeled fragment of the domain. Another threat is that the subset of features in the reference model may not be representative of the entire system. We compared both in Figure 6.5, observing that the subset of features had a similar distribution in its number of implications to the distribution for all features. However, the features in the subset tend to have longer descriptions than the rest of FreeBSD's features; still, the distribution is similar to that of Linux and eCos. Thus, while applying our procedures on all features of FreeBSD would likely produce worse results than for the subset, the results for the subset—in particular, the need to review 3% of RAFs to have a 75% chance of finding the right parent—are consistent with those for Linux.

We have not run user experiments to evaluate the effort saved by our procedures. Such experiments involve usability while we focus on the reverse engineering algorithms in this paper. These evaluations and incorporating our procedures into modeling tools is future work.

6.5 Conclusions

In this chapter, we introduced the semi-automated FEATURE-TREE-SYNTHESIS algorithm for constructing a feature hierarchy using both logical dependencies and textual similarity. FEATURE-TREE-SYNTHESIS presents two lists: the Ranked Implied Features (RIFs) and the Ranked All Features (RAFs). Given a feature, the RIFs ranks its implied features based on their textual similarity to the selected feature. The RAFs list is used in the presence of incomplete dependencies by ranking all features in the input by their textual similarity to the selected feature.

Our evaluation shows that the procedures work well for Linux and eCos where the input data is complete, and also for FreeBSD, a system with both incomplete descriptions and dependencies. By leveraging dependencies and descriptions, the RIFs list contains the correct parent for 76% of features in Linux and 79% in eCos in its first five positions. If we omit dependencies and rely strictly on our ranking heuristic (RAFs), the user only needs to examine 3% to 6% of all features to find the parent in most cases. Assuming the reference hierarchy is selected, all all MUTEX and XOR-groups for the Linux and eCos input are recovered.

Chapter 7

Related Synthesis Techniques

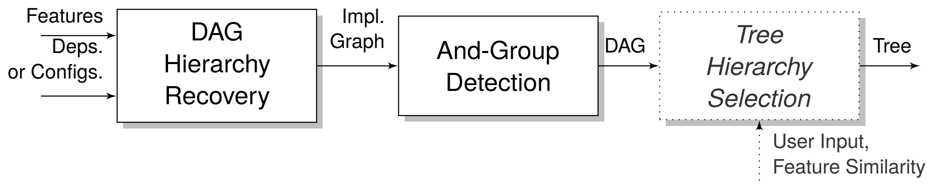
This thesis builds on the synthesis work by Czarnecki and Wąsowski [CW07]. Czarnecki and Wąsowski was the first to introduce a synthesis algorithm that uses dependencies as input and binary decision diagrams (BDDs) for reasoning. We later extended this work to develop the FEATURE-GRAPH-EXTRACTION framework that allows both dependencies and configurations as input and uses SAT solvers for reasoning [ACSW12, SRA+13]. Ryssel's synthesis technique is based on formal concept analysis and translates feature group recovery to minimal set cover problems [RPK11]. Other synthesis techniques operate on natural language text and apply clustering such as the techniques by Alves et al. [ASB+08] and Niu et al. [NE08b]. In Chapter 6, we described a heuristic-based, semi-automated procedure for building a feature hierarchy. Related heuristic-based techniques include the guided model building technique by Janota et al. [JKW08], model management operations by Acher [Ach11], and the automatic synthesis algorithm by Haslinger et al. [HLHE13]. We discuss and compare these techniques with our own in this chapter.

Chapter Organization For the tree recovery stage, some techniques first recover a directed acyclic graph (DAG) that represents all possible tree hierarchies given the input; we discuss these techniques in Section 7.1. Section 7.2 discusses techniques that recover a tree hierarchy directly. A third class of tree recovery techniques, applies heuristics and user input to select a tree given a DAG recovered using a DAG recovery technique. We discuss these techniques in Section 7.3.

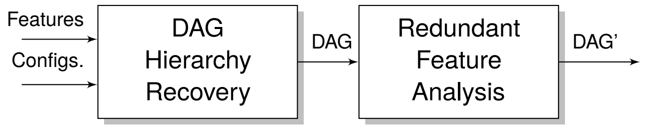
	Input[1]	Sound[2]	Complete[2]	Technique
(a) DAG Recovery Techniques				
Czarnecki (BDD) [CW07]	D	✓	✓	Binary decision diagrams
Andersen (CNF) [ACSW12],[SRA+13]	D	✓	✓	SAT solver, prime implicate computation
Andersen (DNF) [ACSW12],[SRA+13]	C	✓	✓	SAT solver, prime implicant computation
Ryssel [RPK11]	C	✓	✓	Formal concept analysis, minimal set covers
Snelting [Sne96]	C	✓	✓	Formal concept analysis
Czarnecki (Prob.) [CSW08], Dumitru [DGH+11]	C	✓	✓	Association rule mining
(b) Clustering-based Tree Recovery Techniques				
Alves [ASB+08], Weston [WCR09]	C	N/A	N/A	Hierarchical agglomerative clustering
Niu [NE08b]	C	N/A	N/A	Overlapping clustering
Chen [CZZM05]	C	N/A	N/A	Weighted graph clustering
(c) Heuristics-based Tree Selection Techniques				
She [SLB+11]	G	✕	✓	Text similarity
Janota [JKW08]	G	✕	✓	Guided model editing
Acher (FMs) [Ach11]	G	✕	✓	Input FM hierarchies
Acher (Products) [ACP+12]	G	✕	✓	User input and heuristics
Haslinger [HLHE13]	C	✓[3]	✓[3]	Automatic heuristic-based

[1] Required form of input. **C** is *configurations*, **D** is *dependencies*, and **G** is a *DAG*.
[2] Soundness and completeness is defined with respect to φ_G—the maximal DAG consisting of all binary implications in the input.
[3] Haslinger's technique is sound and complete if the input contains only feature diagram constraints (e.g., binary implications and exclusions, OR- and XOR-group constraints).

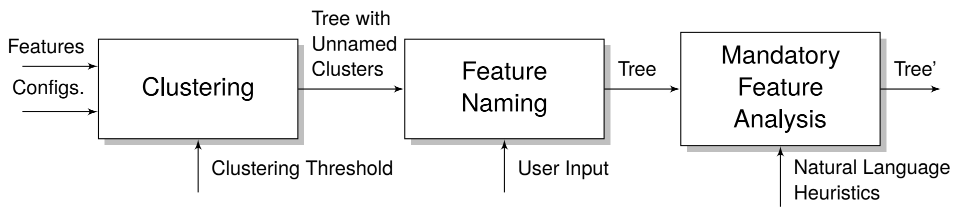
Table 7.1: Summary of related feature model synthesis techniques



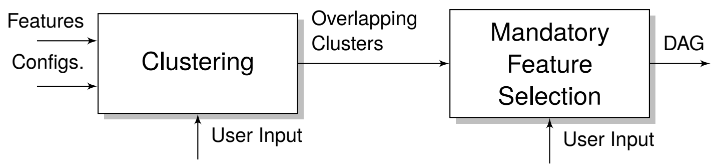
(a) Czarnecki [CW07], Andersen [ACSW12], She [SLB+11], Janota [JKW08]



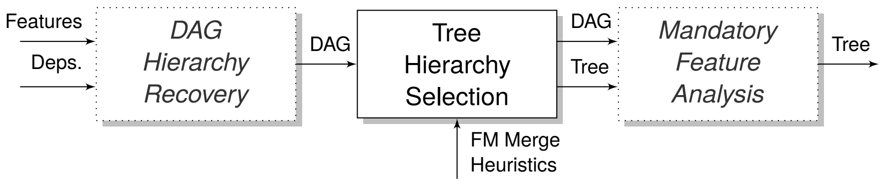
(b) Ryssel [RPK11]


Features
Configs.
Clustering
Tree with Unnamed Clusters
Feature Naming
Tree
Mandatory Feature Analysis
Tree'
Clustering Threshold
User Input
Natural Language Heuristics

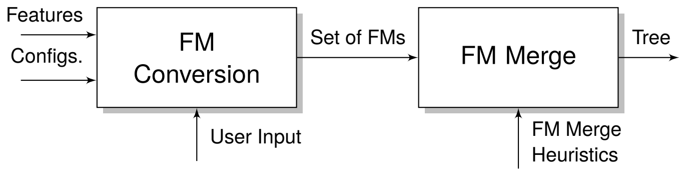
(c) Alves [ASB+08], Weston [WCR09]



(d) Niu [NE08b]



(e) Acher (FMs) [Ach11]



(f) Acher et al. (Products) [ACP+12]

Figure 7.1: Workflows for DAG and Tree Recovery Techniques

7.1 DAG Recovery Techniques

One of the key challenges of feature model synthesis is selecting a feature tree from the many possible feature trees that describe the input configurations, φ. Our FEATURE-GRAPH-EXTRACTION algorithm recovered a directed acyclic graph (DAG) that describes all trees derivable from φ. The maximal DAG is equivalent to the transitively reduced implication graph of φ (Definition 5.4). Given the set of features $\mathcal{F}$, we define φ_G as the formula created by conjoining all binary implications found in φ, or equivalently, conjoining all implications in the implication graph of φ:

$$\varphi_G = \bigwedge_{\{(u,v)\in\mathcal{F}\times\mathcal{F} \mid \varphi\wedge u\to v\}} (u \to v) \tag{7.1}$$

Given a DAG or a tree, $G(V,E)$, where V is a set of features and $E \subseteq V \times V$, its semantics are the conjunction of all of its edges as implications, $[\![G]\!] = \bigwedge_{(u,v)\in E} u \to v$. A technique is *sound* if its recovered DAG, G, satisfies the property $[\![G]\!] \to \varphi_G$. A technique is *complete* if its DAG, G, satisfies the property $\varphi_G \to [\![G]\!]$. All of the DAG recovery techniques in Table 7.1a are sound and complete. These techniques recover a DAG that contains all binary implications in the input, φ. We discuss each technique below.

Czarnecki and Wąsowski [CW07]

Overview Czarnecki and Wąsowski developed the first technique for synthesizing a DAG from a propositional formula represented as a binary decision diagram. A *binary decision diagrams (BDD)* is a data structure that efficiently stores, analyzes, and manipulates propositional formulas [And97]. BDDs can represent arbitrary propositional formula, i.e., dependencies. We adapt their algorithm to the FEATURE-GRAPH-EXTRACTION framework as the FGE-BDD variant in Chapter 5. Figure 7.1a shows the workflow for this algorithm. Mandatory features are identified by detecting strongly connected components (SCCs) in the recovered DAG, then merged into a single node called an AND-group. For each feature p, OR-groups are found by identifying prime implicants of $\neg\varphi \wedge \neg p$ using the algorithm by Coudert and Madre [CM92].

Comparison The synthesis algorithms in this thesis are extensions of the algorithms in [CW07]. The original algorithm by Czarnecki and Wąsowski assumes propositional formulas are stored as BDDs. The scalability of a BDD is determined by the number of variables, the variable ordering, and the propositional formula that the BDD represents. In the worst case, BDDs can grow exponentially with respect to the number of variables. In our experience, we encountered scalability issues when constructing the formula for the Linux variability kernel where BDD construction used up availablem memory (4GB). The algorithms in this thesis adapted the original algorithms to use SAT solvers and extend support for MUTEX-groups.

Ryssel et al. [RPK11]

Overview The synthesis algorithm by Ryssel et al. was the subject of our evaluation with FGE- DNF (Section 5.9). We described Ryssel's technique using the FEATURE-GRAPH-EXTRACTION framework in Section 5.5 as FGE-FCA. In Ryssel's technique, the DAG is recovered as a *attribute concept graph* by checking subsets of features between configurations. Ryssel's DAG recovery is similar to the approach in FGE-DNF. Feature groups are found by solving minimal set cover problems. The possible set cover candidates consists of only immediate children of a given parent. Ryssel also addresses the recovery of *complex implications*—implications with the form $x_1 \wedge \cdots \wedge x_n \rightarrow y_1$. Ryssel first builds an extended attribute concept graph that includes negated features as nodes. The complex implications are identified by solving a minimal set cover problem using the extended attribute concept graph to limit the problem size.

Comparison Ryssel's technique is comparable to our FGE-DNF algorithm. The process for building the implication and mutex graphs are identical between Ryssel's technique and our own. However, Ryssel's technique supports the extraction of complex implications, whereas FGE-DNF does not. Ryssel's technique for extracting complex implications can be applied to FGE-DNF since both techniques use input as a set of configuration.

Snelting [Sne96]

Overview Similar to Ryssel's technique [RPK11], Snelting uses formal concept analysis to build concept lattices. However, Snelting applies his technique to C code annotated with dependencies using #ifdef statements. The concept lattice is equivalent to the reduced implication graph of our FGE algorithms (Definition 5.4). Snelting derives the formal context (i.e., configuration table) by making each conditional fragment of C code that's enclosed in a #ifdef statement into a configuration. A C preprocessor symbol is a feature in the table and a conditional fragment contains a feature if the symbol appears in its #ifdef condition. Negated preprocessor symbols are treated as a separate feature. Since Snelting's work was not directed towards feature model synthesis, his technique does not address feature group or cross-tree constraint recovery.

Comparison Snelting uses the same approach as Ryssel et al. [RPK11] to construct the concept lattice used to derive the feature hierarchy. Their approach is conceptually the same as the approach we use to derive the implication graph in FGE-DNF.

Probabilistic techniques—Czarnecki et al. [CSW08], Dumitru [DGH+11]

Overview In our previous work, we proposed a technique for synthesizing *probabilistic feature models*—feature models with support for suggestions via soft constraints [CSW08]. The algorithms use a *sample set* as input. Unlike a set of configurations, a sample set allows configurations to be repeated multiple times. We detect association rules in the sample set. An *association rule* $A \Rightarrow B$ is an expression between two boolean formulas, A and B with an associated *interestingness*—a measure of the quality and strength of the association rule in the input sample set [She08]. The interestingness measures used by our algorithm was *support* and *confidence* [She08]. Support is a measure of statistical significance, and confidence is a measure of the association rule strength. An association rule with 100% confidence is equivalent to a Boolean implication. The DAG extracted by the association rule mining consists of only association rules with 100% confidence, making it equivalent to the maximal DAG. Feature groups are found by identifying disjunctive association rules, that are in turn, identified by mining for minimal OR-clauses using the algorithm by Zhao et al. [ZZR06]. Zhang and Becker use association rule mining to recover complex association rules with multiple features in the antecedent and consequent [ZB13]. These complex rules can be used as cross-tree constraints of a probabilistic feature model and help guide product configuration.

Dumitru et al. propose a system for recommending product features extracted from openly available product descriptions [DGH+11]. In their paper, they state that they construct a probabilistic feature model using association rule mining. However, details on the hierarchy and feature groups recovery are missing in the paper.

Comparison Unlike a set of configurations, sample sets allow repeated configurations. Identifying association rules requires first identifying *frequent itemsets*—collections of items that appear more than a minimum specified support threshold. Frequent itemsets are similar to binary implications, with the added constraint that the items must appear together a minimum number of times.

7.2 Clustering-Based Tree Recovery Techniques

Clustering is used to identify similar features by comparing the associated feature text (i.e., names and descriptions) using a measure such as Latent Semantic Analysis (LSA) [DDF+90] or the Vector Space Model (VSM) [SWY75]. The clustering approaches identify groups of similar features by comparing the common words between feature text. We experimented with LSA and VSM as a similarity measure between features, but found that these techniques were not as efficient as TF-IDF (term frequency-inverse document frequency) for the short feature names and descriptions in the Linux and eCos kernels. Unlike using a set of configurations or dependencies as input, there are many different kinds of relations between features in natural language text, with not all relations being relevant to a feature model.

Unlike the DAG recovery techniques, clustering-based techniques form a feature hierarchy by creating *abstract features* that group features that are part of the same cluster. Abstract features are features that are not in the input feature set, and are synthesized to group related features. Abstract features are particularly useful for domain analysis since they identify feature groupings that may not have been obvious to the modeler.

Alves et al. [ASB+08] and Weston et al. [WCR09]

Overview Alves and Weston use hierarchical agglomerative clustering to construct the feature tree from textual requirements (Figure 7.1c). In hierarchical agglomerative clustering, the feature hierarchy is formed by clustering with progressively lower similarity thresholds until only a single cluster remains. The parent-child relations are determined by subset inclusion between the clusters' features. Once the feature tree is built, mandatory features are identified by searching for specific keywords, such as "optional", "alternatively", or "at least one of" [ASB+08].

Comparison Hierarchical agglomerative clustering creates abstract features for each cluster of features. Our techniques can use clustering to create abstract features that group related sibling features. A significant difference between our techniques is that Alves and Weston use the textual content as a source of variability. Our approach does not extract variability from the textual descriptions; instead, we rely on a separate logical formula as the source of variability. FEATURE-TREE-SYNTHESIS uses the textual content to provide recommendations for building the feature hierarchy.

Niu et al. [NE08b]

Overview Niu describes a user-driven approach based on overlapping clustering for synthesizing feature models (Figure 7.1d). Niu identifies overlapping clusters between functional requirements to produce a set of interconnected, crosscutting feature trees. Their similarity measure is over the set of attributes of abstract requirements called functional requirements profiles (FRPs). The FRPs and their attributes need to first be extracted manually from the requirements. The connections between the feature trees created through overlapping clustering form a DAG, but does not contain all possible edges to make it complete. Niu relies on manually marking features as mandatory, otherwise they are considered optional.

Comparison Building functional requirement profiles can be interpreted as part of the variability analysis stage. Once FRPs are identified, we can translate the FRPs to sets of configurations and use the FGE-DNF algorithm to synthesize a feature model. FGE-DNF can automatically identify whether a feature is mandatory or optional.

Chen et al. [CZZM05]

Overview The synthesis technique by Chen et al. first requires that weighted graphs describing relations between requirements be built by manually analyzing requirements documents. The weighted graphs are then clustered to identify similar features and construct the feature hierarchy. However, unlike the previous two techniques, Chen's clustering is based on edge weights instead of textual similarity. Their algorithm progressively decreases the similarity threshold and the hierarchy is constructed by checking subset inclusion between the identified clusters. Chen also relies on the user for marking features as mandatory.

Comparison Chen's technique defers part of the hierarchy selection to the variability analysis stage by relying on manually constructed weighted graphs where the edge weights determine the hierarchy of the resulting feature model. As a result, their synthesis stage is fully automatic. Similarly, applying an early hierarchy selection to our techniques can make the synthesis stage fully automatic. For future work, we can use concepts from Chen's approach (e.g., weighted graphs, spanning trees) to automatically derive a feature hierarchy.

7.3 Tree Selection Techniques

Feature models consists of a feature tree and a set of cross-tree constraints, however, more than one feature tree can model the same set of configurations. In Section 5.8, we described an automated algorithm for extracting a single feature tree from a DAG. However, feature models are artifacts designed to be read and interpreted by humans and such an automatically selected feature tree is likely to be nonsensical to the modeler. The tree selection techniques in this section use heuristics and user input to select a distinct feature tree from an input DAG. User input is needed so the resulting feature tree matches the expectations of the modeler. Heuristics reduce the effort of the modeler by identifying likely candidates. The input DAG to these techniques can be recovered using any of the DAG recovery techniques in Section 7.1. All of these tree selection techniques are *complete*—the recovered tree contains a subset of the edges in the implication graph of φ.

We compare each technique to FEATURE-TREE-SYNTHESIS that we described on Chapter 6. FEATURE-TREE-SYNTHESIS is a technique for selecting a feature hierarchy by ranking potential parents for a feature by a textual similarity measure. Given a feature, the modeler is presented with two lists of features ranked by their similarity to the selected feature. The first list rank features that imply the selected feature. By selecting only features in this list, the resulting feature model is guaranteed to be complete with respect to the input dependencies. The second list ranks all features in the input by the similarity. This second list addresses practical settings where the set of dependencies may be incomplete. For example, static analysis tools may only recover dependencies that are can be guaranteed to be sound. The extracted dependencies may be missing valid dependencies that the static analysis tool could not identify.

Janota et al. [JKW08]

Overview Janota et al. described an interactive modeling editing tool that produces only feature models that are complete with respect to a set of features and a set of input configurations or dependencies. Their tool uses a feature graph (Section 5.2) as input. Their tool allows model edit operations that are consistent with the constraints contained in the feature graph. The set of cross-tree constraints are derived by performing a logical difference between the constraints in the feature diagram and the input configurations or dependencies.

Comparison Janota proposed a model builder that relies on FEATURE-GRAPH-EXTRACTION to construct a feature graph. In his work, hierarchy selection is manual with the constraint that the model is implied by the input formula. FEATURE-TREE-SYNTHESIS can supplement Janota's model builder by providing a ranking of potential parents.

Acher (FM) [Ach11]

Overview This technique by Acher describes automated model management operations on feature models that include model merges. Acher describes heuristics for selecting the structure of the resulting merged model based on the structures of the input models. These heuristics enable the automatic selection of a distinct feature tree that matches the expectation of the user based on the selected feature model merge operation. Acher first translate the input models to their propositional formulas [Bat05] then perform merges the formula with a propositional operation. The resulting formula is used as input to recover a feature graph by using Czarnecki and Wąsowski's BBD-based implementation [CW07]. Acher's heuristics are applied on the feature graph to automatically select a feature hierarchy and feature groups. Acher reported that the synthesis algorithm scaled up to propositional formulas with 2000 features.

Comparison Acher's technique exploits the structure of the input models to automatically select a hierarchy. While Acher's algorithm is automatic, it may not always select the hierarchy that is expected by the user. For future work, we can use FEATURE-TREE-SYNTHESIS to supplement his technique by suggesting alternative hierarchies based on the textual similarity between features.

Acher et al. (Products) [ACP+12]

Overview Acher et al. described an approach for synthesizing feature models by using heuristics based on semi-structured data (i.e., tabular data). Acher's technique exploits structurally similar product descriptions and uses a conversion specification to describe the transformation of a single semi-structured product description to a feature model. The individual feature models are then merged to form a final feature model describing all products in the dataset. The conversion specification is tailored to parse and interpret the specific structure of the input data. The individual feature models share the same hierarchy making the final merge relatively simple. Acher et al. rely on their FAMILIAR domain specific language for performing the merge operations [ACLF11].

Comparison FEATURE-TREE-SYNTHESIS relies on textual similarity to suggest a hierarchy to the user. Similar to the previous algorithm, Acher's heuristics exploits the structure of the data to automatically select a hierarchy. We can use a similar approach of exploiting the structure of the input to improve our ranking mechanism. For example, for object-oriented source code, our algorithm can use the fact that methods are nested within classes, and classes are within packages. We leave this to future work.

Haslinger et al. [HLHE13]

Overview Haslinger et al. describe an automated algorithm that synthesizes a feature model from a set of configurations. Haslinger's computation of implication and mutex graph is similar to that of FGE-DNF (Chapter 5). Haslinger proposes an automated algorithm for extracting a distinct feature tree given the input configurations, implication graph, and mutex graph.

Their algorithm performs a traversal across all features starting at the root—they call this a bottom-to-top traversal of the implication graph. Given a feature, XOR-groups are identified by identifying cliques in the mutex graph between direct children of the given feature. For the remaining children not in an XOR-group, Haslinger detects whether a feature is a so-called *true optional feature*—a feature such that there exists valid configurations in the input where the true optional feature can be selected or deselected, ignoring any features that imply it. In other words, a true optional feature is one that does not have a constraint other than the parent-child constraint. At this point of the algorithm, each child feature is classified into one of three cases: (1) it is a true optional—the feature is marked as optional in the hierarchy, (2) it is a possible optional, but is involved in some other global constraints—in this case, the feature is moved up a level in the hierarchy, or (3) it is involved in constraints among its siblings—it becomes part of an OR-group in the following step of the algorithm. Haslinger's algorithm is only capable of handling input with XOR- and OR-groups, and implies and excludes edges. In their algorithm, mandatory features are detected as *atomic sets* that are equivalent to strongly connected components (SCCs) in our algorithm. The authors do not discuss how a hierarchy is determined among features in an atomic set.

Comparison Unlike FEATURE-TREE-SYNTHESIS, the algorithm by Haslinger et al. is fully automatic. However, the trade-off is that the resulting feature model may differ from the expectation of the user. In their evaluation, the authors state that the synthesized model may not match the input model, however, the models are equivalent in terms of their configurations. Furthermore, their technique only works on input that contain only the identified feature diagram components. In essence, Haslinger's algorithm can be seen as heuristics for automatically deriving a specific feature model from a feature graph. Their algorithm prioritizes the detection of XOR-groups, then a special class of optional features—the true optional features. The remaining features are moved up in the hierarchy until they are made into a OR-group.

Chapter 8

Conclusions

We summarize the contributions of this thesis below:

- Chapter 3 describes several feature model synthesis scenarios and their workflows. We extracted these from literature and industry experience reports. We found that the scenarios varied widely in their intended use of feature model synthesis and the variability was extracted from a wide range of input artifacts. We derived requirements for synthesis techniques that we address in the algorithms presented in Chapters 5 and 6. We use these requirements to classify related synthesis techniques in Chapter 7.

- Chapter 4 describes the Kconfig variability modeling language. This language is used to specify the variability models of the Linux kernel and other open-source projects. The features, descriptions, and propositional formulas extracted from the Kconfig models, including the Linux variability model, make excellent benchmarks for feature model analysis and synthesis tools. Our work on Kconfig addresses the need for input derived from large, realistic variability models [BSRC10].

- Chapter 5 describes the automated FEATURE-GRAPH-EXTRACTION algorithm that recovers a feature graph given features and a propositional formula in conjunctive normal form (CNF) or disjunctive normal form (DNF). The CNF algorithm is able to handle much larger input—such as input derived from the Linux variability model—when compared to the BDD-based synthesis technique by Czarnecki and Wąsowski [CW07]. Our evaluation showed a 10 to 1000 time improvement over BDD-based implementation when the BDD-based method did not timeout. We found that the DNF algorithm was comparable in runtime to the formal concept analysis-based algorithm by Ryssel et al. [RPK11]. However, there were several models that timed out with Ryssel's algorithm, but completed with our algorithm.

- Chapter 6 describes the semi-automated FEATURE-TREE-SYNTHESIS algorithm. This algorithm is used to construct a feature hierarchy given features, valid configurations, and supplemental feature descriptions. The algorithm presents two lists: the *Ranked Implied Features* (RIFs) ranks the implied features by their textual similarity to the selected feature. A second list, the *Ranked All Features* (RAFs), accounts for incomplete input dependencies by ranking all features irrespective of their implications by the feature similarity measure. FEATURE-TREE-SYNTHESIS was the first synthesis algorithm to combine logical dependencies with heuristics applied to natural language text. Our evaluation showed that the correct parent was in the top 5 positions of the RIFs list for 76% of features in the input derived from the Linux variability model, and 79% of features from the eCos variability model. For the RAFs list, users would have to examine 3% to 6% of all features to identify the correct parent.

8.1 Future Work

Bottom-Up vs. Top-Down Synthesis The synthesis procedure in Chapter 6 is a bottom-up procedure. The procedure starts with the individual features and forms a forest of trees as the procedure progresses. The procedure is completed when a single tree is formed. However, a bottom-up procedure may not be not the most natural, or effective way of constructing a feature model. Jepsen et al. reported that developers applied both a top-down and a bottom-up approach in parallel to re-engineering a software system to a SPL at Danfoss Drives [JDB07]. The top-down approach involved making a feature model for two products as a form of domain analysis. Requirements, documentation, VPs, and VP configs were used as input for the top-down approach.

For the bottom-up approach, the developers performed a difference between the codebases of the two products and inserted `#ifdef` statements to identify VPs and their conditions between the products. The authors report that the bottom-up approach was more effective than the top-down approach. While the feature model constructed using the top-down approach was correct in terms of the domain, the developers struggled to define features at the right level of granularity. For future work, we can integrate our synthesis algorithms to support both at top-down and bottom-up procedure.

Abstract Features Abstract features are used for grouping related sub-features. The feature model synthesis procedures that we describe do not provide any assistance with creating abstract features. In feature modeling, abstract features are simply represented as a mandatory feature. Whether a feature is abstract or concrete can be inferred from the feature name and description, and is determined from whether the feature appears in the problem-to-solution space mapping.

In the Kconfig language, abstract features take the form of menus. For example, the menu "CPU Scheduler" in the Linux Kconfig model is an abstract feature that does not declare a build symbol and is not referenced in the source code. The sub-features of "CPU Scheduler" are concrete features that implement the actual CPU schedulers.

The synthesis techniques that we describe synthesize feature models that contain exactly the set of input features. In a software re-engineering scenario where automatic static analysis is used, these input features are only concrete features that are detected in the analyzed source code. In other scenarios, any abstract features in the synthesized feature model will have to be determined in advance to our synthesis procedures, or added in after synthesis.

We can draw inspiration from the the natural language clustering-based techniques described in Chapter 7. Alves and Weston use hierarchical agglomerative clustering to synthesize a hierarchy from a set of input requirements [ASB+08, WCR09]. The input requirements become the leaves of the model and the clustering algorithm is used to successively group similar requirements by introducing abstract features.

Incremental Procedure The procedures presented in Chapters 5 and 6 assume a one-way, batch process for synthesizing a feature model. In a practical setting, synthesizing a feature model is likely an iterative, and incremental procedure. Beuche's workflow for a feature-oriented software re-engineering depicts the variability analysis and model building stages as incremental processes (Figure 1.4) [Beu06]. An incremental procedure is also useful for comparing and updating feature models with merging from other software artifacts (e.g., code changes). An incremental synthesis procedure would involve incremental analysis and editing tools as well.

Handling Unsound or Incomplete Input Feature model synthesis is one part of a larger re-engineering workflow. The correctness of a synthesized feature model is dependent upon the correctness of the tools used to extract the needed input. For example, when we synthesized a feature model for a portion of the FreeBSD kernel

(Chapter 6), the set of input features and dependencies were extracted from source code, the build system, and documentation using a custom-built tool for analysis. The correctness of the synthesized feature model is dependent upon the correctness of the tools used to extract the synthesis input. We addressed incomplete dependencies in our synthesis technique addressed incomplete dependencies by relying strictly on the feature similarity measure. As part of future work, we could investigate other techniques for handling incomplete or unsound input. Developing these tools would likely involve developing an incremental synthesis procedure as well.

8.2 Summary

Variability is a software system's ability to adapt and customize for a particular context [vGBS01]. Unfortunately, variability is often tangled within an artifact and scattered over multiple artifacts. Furthermore, ad-hoc techniques for handling variability can lead to build-time errors or undetected run-time errors. Variability can be systematically handled with product line engineering—a development methodology that strives to exploit commonalities for a problem domain while managing variability in a systematic way [Cza04, WL99]. Software product lines, a form of systems family engineering, systematically enables the reuse of code across a family of related products with command and variable product characteristics [CN01]. Software product lines advocate the use of explicitly defined variability models, such as feature models. Feature models provide an explicit, centralized source for representing variability in a software system. However, current techniques for building feature models rely entirely on manual analysis and construction by a domain modeler. This process is difficult and time-consuming. Automated tooling for feature models synthesis would greatly reduce the effort and time required to construct a feature model.

Our work on feature model synthesis addresses the need for automated tooling with two algorithms. The first, FEATURE-GRAPH-EXTRACTION, is an automated procedure for synthesizing a feature graph that represents all complete feature diagrams for a given input as a propositional formula in CNF or DNF. The configuration semantics of all feature diagrams that are derivable from the extracted feature graph are implied by the input formula. The CNF-based algorithm was 10 to 1000 times faster than the previous BDD-based algorithm and could handle input that was significantly larger, such as the Linux variability model. The DNF-based algorithm was comparable to the FCA-based algorithm by Ryssel et al. [RPK11].

The second algorithm, FEATURE-TREE-SYNTHESIS, is a semi-automated procedure for selecting a feature hierarchy given a feature graph and supplemental descriptive text. Our evaluation showed that our algorithm works well for input derived from Linux and eCos where the data is complete, and also for FreeBSD, a system with incomplete descriptions and dependencies.

Our third contribution is the analysis of the Kconfig language. While large feature models have been reportedly used in industry, the research community has suffered from a lack of large, realistic benchmarks for feature model analysis tools [MBC09, BSRC10]. We built the *Linux Variability Analysis Tools (LVAT)* to analyze Kconfig models and translate the models to a propositional formula for analysis and synthesis tools. We used LVAT to extract a propositional formula for the Linux Kconfig model with over 6000 features. This formula represents the largest realistic variability model available to the research community to-date and we used it to evaluate both FEATURE-GRAPH-EXTRACTION and FEATURE-TREE-SYNTHESIS algorithms.

Bibliography

[AC04] M. Antkiewicz and K. Czarnecki, "FeaturePlugin: feature modeling plug-in for Eclipse," in *Proceedings of the 2nd OOPSLA workshop on Eclipse Technology eXchange (ETX 2004)*. ACM, 2004.

[Ach11] M. Acher, "Managing multiple feature models: Foundations, language, and applications," Ph.D. dissertation, University of Nice Sophia Antipolis, France, 2011.

[ACLF11] M. Acher, P. Collet, P. Lahire, and R. B. France, "A domain-specific language for managing feature models," in *Proceedings of the 26th Symposium on Applied Computing (SAC 2011)*. ACM, 2011.

[ACP$^+$12] M. Acher, A. Cleve, G. Perrouin, P. Heymans, P. Collet, and C. V. Lahire, Philippe, "On extracting feature models from product descriptions," in *Proceedings of the 6th International Workshop on Variability Modelling of Software Intensive Systems (VaMoS 2012)*. ACM, 2012.

[ACSW12] N. Andersen, K. Czarnecki, S. She, and A. Wąsowski, "Efficient synthesis of feature models," in *Proceedings of the 16th International Software Product Line Conference (SPLC 2012)*. ACM, 2012.

[AGU72] A. V. Aho, M. R. Garey, and J. D. Ullman, "The transitive reduction of a directed graph," *SIAM Journal on Computing*, vol. 1, no. 2, 1972.

[AK09] S. Apel and C. Kästner, "An overview of feature-oriented software development," *Journal of Object Technology*, Jul. 2009.

[And97] H. R. Andersen, "Binary decision diagrams," Department of Information Technology, Technical University of Denmark, 1997, lecture notes for 49285 Advanced Algorithms E99. Available: http://www.itu.dk/people/hra/notes-index.html

[And09] N. Andersen, "Automatic synthesis of feature models based on satisfiability checking," Master's thesis, IT University of Copenhagen, Denmark, 2009.

[ASB+08] V. Alves, C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl, and A. Rummler, "An exploratory study of information retrieval techniques in domain analysis," in *Proceedings of the 12th International Software Product Line Conference (SPLC 2008)*. IEEE Computer Society, 2008.

[Bat04] D. Batory, "Feature-oriented programming and the AHEAD tool suite," in *26th International Conference on Software Engineering (ICSE 2004)*. IEEE Computer Society, 2004.

[Bat05] D. Batory, "Feature models, grammars, and propositional formulas," in *Proceedings of the 9th International Software Product Line Conference (SPLC 2005)*. Springer, 2005.

[BCFH10] Q. Boucher, A. Classen, P. Faber, and P. Heymans, "Introducing TVL, a text-based feature modelling language," in *Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010)*, ser. ICB-Research Report. Universität Duisburg-Essen, 2010.

[BCW10] K. Bak, K. Czarnecki, and A. Wąsowski, "Feature and meta-models in clafer: mixed, specialized, and coupled," in *Proceedings of the 3rd International Conference of Software Language Engineering (SLE 2010)*. Springer-Verlag, 2010.

[BEGB11] E. Bagheri, F. Ensan, D. Gaševic, and M. Boškovic, "Modular feature models: Representation and configuration," *Journal of Research and Practice in Information Technology*, vol. 43, no. 2, 2011.

[Ber12] T. Berger, "Variability modeling in the real: An empirical journey from software product lines to software ecosystems," Ph.D. dissertation, University of Leipzig, Germany, 2012.

[Beu06] D. Beuche, "Migration of legacy systems to software product lines," pure-systems GmbH, 2006. Available: http://www.mddpl.uni-leipzig.de/archiv/mddpl06/slides/beuche.pdf

[BMAC05] D. Benavides, P. T. Martín-Arroyo, and A. R. Cortés, "Automated reasoning on feature models," in *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE 2005)*. Springer, 2005.

[BMB+10] M. Bošković, G. Mussbacher, E. Bagheri, D. Amyot, D. Gašević, and M. Hatala, "Aspect-oriented feature models," in *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010)*. Springer, 2010.

[BPSP04] D. Beuche, H. Papajewski, and W. Schröder-Preikschat, "Variability management with feature models," *Sci. Comput. Program.*, vol. 53, no. 3, Dec. 2004.

[BS10] T. Berger and S. She, "Formal semantics of the CDL language," 2010, technical note.

[BSL+10a] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wąsowski, "Feature-to-code mapping in two large product lines," in *14th International Software Product Line Conference, Software Product Lines: Going Beyond (SPLC 2010)*. Springer, 2010.

[BSL+10b] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki, "Variability modeling in the real: a perspective from the operating systems domain," in *25th International Conference on Automated Software Engineering (ASE 2010)*. IEEE/ACM, 2010.

[BSL+12] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki, "Variability modeling in the systems software domain," Generative Software Development Lab, University of Waterloo, Tech. Rep. GSDLAB-TR 2012-07-06, 2012. Available: http://gsd.uwaterloo.ca/sites/default/files/vm-2012-berger.pdf

[BSRC10] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: a literature review," *Information Systems*, vol. 35, no. 6, 2010.

[BV00] J. D. Bart Veer, "The eCos component writer's guide," 2000, retrieved December 11, 2012. Available: http://ecos.sourceware.org/docs-2.0/cdl-guide/cdl-guide.html

[CE00] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CGR+12] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wąsowski, "Cool features and tough decisions: A comparison of variability modeling approaches," in *6th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2012)*. ACM, 2012.

[CHE05a] K. Czarnecki, S. Helsen, and U. Eisenecker, “Formalizing cardinality-based feature models and their specialization,” *Software Process Improvement and Practice*, vol. 10, no. 1, 2005.

[CHE05b] K. Czarnecki, S. Helsen, and U. Eisenecker, “Staged configuration through specialization and multi-level configuration of feature models,” in *Software Process Improvement and Practice*, 2005.

[CKK06] K. Czarnecki, C. H. P. Kim, and K. Kalleberg., “Feature models are views on ontologies,” in *Proceedings of the 10th International Software Product Line Conference (SPLC 2006)*. Springer, 2006.

[CM92] O. Coudert and J. C. Madre, “Implicit and incremental computation of primes and essential primes of boolean functions,” in *Proceedings of the 29th ACM/IEEE Conference on Design Automation*. IEEE Computer Society, 1992.

[CN01] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[CSW08] K. Czarnecki, S. She, and A. Wąsowski, “Sample spaces and feature models: There and back again,” in *Proceedings of the 12th International Software Product Line Conference (SPLC 2008)*. IEEE, 2008.

[CW07] K. Czarnecki and A. Wąsowski, “Feature models and logics: There and back again,” in *Proceedings of the 11th International Software Product Line Conference (SPLC 2007)*. IEEE Computer Society, 2007.

[Cza04] K. Czarnecki, *Overview of Generative Software Development*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2004, vol. 3566.

[CZZM05] K. Chen, W. Zhang, H. Zhao, and H. Mei, “An approach to constructing feature models based on requirements clustering,” in *Proceedings of the 13th International Conference on Requirements Engineering (RE 2005)*. IEEE Computer Society, 2005.

[DDF$^+$90] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, “Indexing by latent semantic analysis,” *Journal of the American society for information science*, vol. 41, no. 6, 1990.

[DGH$^+$11] H. Dumitru, M. Gibiec, N. Hariri, J. Cleland-Huang, B. Mobasher, C. Castro-Herrera, and M. Mirakhorli, “On-demand feature recommendations derived from mining public product descriptions,” in *Proceedings of the 33rd*

International Conference on Software Engineering (ICSE 2011). ACM, 2011.

[DGR07] D. Dhungana, P. Grünbacher, and R. Rabiser, "Domain-specific adaptations of product line variability modeling," in *Proceedings of the IFIP WG 8.1 Working Conference on Situational Method Engineering: Fundamentals and Experiences*. Springer, 2007.

[DGR11] D. Dhungana, P. Grünbacher, and R. Rabiser, "The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study," *Automated Software Engineering*, vol. 18, no. 1, 2011.

[DGRN10] D. Dhungana, P. Grünbacher, R. Rabiser, and T. Neumayer, "Structuring the modeling space and supporting evolution in software product line engineering," *Journal of Systems and Software*, vol. 83, no. 7, Jul. 2010.

[DHR10] D. Dhungana, P. Heymans, and R. Rabiser, "A formal semantics for decision-oriented variability modeling with DOPLER," in *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2010)*, ser. ICB-Research Report. Universität Duisburg-Essen, 2010.

[DRGP11] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 25, no. 1, 2011.

[EPP95] B. Errico, F. Pirri, and C. Pizzuti, "Finding prime implicants by minimizing integer programming problems," in *Australian Joint Conference on AI*, 1995.

[ES03] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*. Springer, 2003.

[FAY11] T. Fukuda, Y. Atarashi, and K. Yoshimura, "An approach to evaluate time-dependent changes in feature constraints," in *Proceedings of the 15th International Software Product Line Conference, Volume 2 (SPLC 2011)*. ACM, 2011.

[FLW11] U. Fahrenberg, A. Legay, and A. Wąsowski, "Vision paper: Make a difference! (semantically)," in *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MODELS 2011)*. Springer, 2011.

[HJA07] T. Hadzic, R. Jensen, and H. Andersen, "Calculating valid domains for bdd-based interactive configuration," *arXiv preprint arXiv:0704.1394*, 2007. Available: http://www.itu.dk/~tarik/cvd/cvd.pdf

[HLHE13] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed, "On extracting feature models from sets of valid feature combinations," in *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE 2013)*. Springer, 2013.

[HTM09] H. Hartmann, T. Trew, and A. Matsinger, "Supplier independent feature modelling," in *Proceedings of the 13th International Software Product Lines Conference (SPLC 2009)*. ACM, 2009.

[Hub12] A. Hubaux, "Feature-based configuration: Collaborative, dependable, and controlled," Ph.D. dissertation, University of Namur, Belgium, 2012.

[Jac92] P. Jackson, "Computing prime implicates incrementally," in *Proceedings of the 12th International Conference on Automated Deduction (CADE 1992)*. Springer, 1992.

[JB08] M. Janota and G. Botterweck, "Formal approach to integrating feature and architecture models," in *Proceedings of the 11th International Conference Fundamental Approaches to Software Engineering (FASE 2008)*. Springer, 2008.

[JDB07] H. P. Jepsen, J. G. Dall, and D. Beuche, "Minimally invasive migration to software product lines," in *Proceedings of the 11th International Software Product Lines Conference (SPLC 2007)*. IEEE Computer Society, 2007.

[JKW08] M. Janota, V. Kuzina, and A. Wąsowski, "Model construction with external constraints: An interactive journey from semantics to syntax," in *Proceedings of the 11th International Conference Model Driven Engineering Languages and Systems (MODELS 2008)*. Springer, 2008.

[JP90] P. Jackson and J. Pais, "Computing prime implicants," in *Proceedings of the 10th International Conference on Automated Deduction (CADE 1990)*. Springer, 1990.

[KAK08] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*. ACM, 2008.

[Kan10] K. Kang, “FODA: Twenty years of perspective on feature modeling,” 2010, keynote presentation. Available: http://www.sse.uni-due.de/vamos/2010/files/VaMoS2010_Keynote_Kang_FODA.pdf

[Käs10] C. Kästner, “Virtual separation of concerns: Toward preprocessors 2.0,” Ph.D. dissertation, University of Magdeburg, Germany, 2010.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, “Feature-oriented domain analysis (FODA) feasibility study,” Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[LSB+10] R. Lotufo, S. She, T. Berger, A. Wasowski, and K. Czarnecki, “Evolution of the linux kernel variability model,” in *Proceedings of the 14th International Software Product Lines Conference (SPLC 2010)*. Springer, 2010.

[MBC09] M. Mendonca, M. Branco, and D. Cowan, “S.P.L.O.T.: software product lines online tools,” in *Companion to the 24th Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA 2009)*. ACM, 2009.

[Men09] M. Mendonca, “Efficient reasoning techniques for large scale feature models,” Ph.D. dissertation, School of Computer Science, University of Waterloo, Jan 2009.

[MFSO97] V. M. Manquinho, P. F. Flores, J. P. M. Silva, and A. L. Oliveira, “Prime implicant computation using satisfiability algorithms,” in *Proceedings of the 9th International Conference on Tools with Artificial Intelligence (ICTAI 1997)*. IEEE Computer Society, 1997.

[MWC09] M. Mendonca, A. Wąsowski, and K. Czarnecki, “SAT-based analysis of feature models is easy,” in *Proceedings of the 13th International Software Product Lines Conference (SPLC 2009)*. ACM, 2009.

[MWCC08] M. Mendonca, A. Wąsowski, K. Czarnecki, and D. Cowan, “Efficient compilation techniques for large scale feature models,” in *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE 2008)*. ACM, 2008.

[NE08a] N. Niu and S. Easterbrook, “Extracting and modeling product line functional requirements,” in *Proceedings of the 16th International Requirements Engineering Conference (RE 2008)*. IEEE Computer Society, 2008.

[NE08b] N. Niu and S. Easterbrook, "On-demand cluster analysis for product line functional requirements," in *Proceedings of the 12th International Software Product Line Conference (SPLC 2008)*. IEEE Computer Society, 2008.

[NE10] A. Nöhrer and A. Egyed, "Conflict resolution strategies during product configuration," in *Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010)*, ser. ICB-Research Report. Universität Duisburg-Essen, 2010.

[OAC$^+$06] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, L. Spoon, E. Stenman, and M. Zenger, "An overview of the Scala Programming Language (2. edition)," EPFL, Tech. Rep. LAMP-REPORT-2006-001, 2006.

[Obj06] Object Management Group, "Meta object facility (MOF) core specification," 2006, document formal/2006-01-01. Available: http://www.omg.org/spec/MOF/2.0/

[Obj12] Object Management Group, "Common variability language (CVL)—OMG revised submission," 2012, document ad/2012-08-05. Available: http://www.omgwiki.org/variability/lib/exe/fetch.php?id=start&cache=cache&media=cvl-revised-submission.pdf

[pur12] pure-systems GmbH, "Screenshots," 2012, retrieved September 14, 2012. Available: http://www.pure-systems.com/Screenshots.51.0.html

[RBSP02] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow, "Extending Feature Diagrams with UML Multiplicities," in *Proceedings of the 6th World Conference on Integrated Design & Process Technology (IDPT 2002)*, Jun. 2002.

[RC10] J. Rubin and M. Chechik, "From products to product lines using model matching and refactoring," in *Proceedings of the 14th International Software Product Lines Conference (SPLC 2010), Volume 2 (Workshops, Industrial Track, Doctoral Symposium, Demonstrations and Tools)*. Springer, 2010.

[RC12] J. Rubin and M. Chechik, "Combining related products into product lines," in *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE 2012)*. Springer, 2012.

[RPK11] U. Ryssel, J. Ploennigs, and K. Kabitzsch, "Extraction of feature models from formal contexts," in *3rd International Workshop on Feature-Oriented*

Software Development (FOSD 2011) in Proceedings of the 15th International Software Product Lines Conference (SPLC 2011), Volume 2, 2011.

[RPK12] U. Ryssel, J. Ploennigs, and K. Kabitzsch, "Automatic library migration for the generation of hardware-in-the-loop models," *Science of Computer Programming*, vol. 77, no. 2, 2012.

[SB10] S. She and T. Berger, "Formal semantics of the Kconfig language," Generative Software Development Lab, University of Waterloo, Technical Note, 2010. Available: http://gsd.uwaterloo.ca/sites/default/files/kconfig_semantics.pdf

[Sch86] D. A. Schmidt, *Denotational semantics: a methodology for language development*. William C. Brown Publishers, 1986.

[SCW12] S. She, K. Czarnecki, and A. Wąsowski, "Usage scenarios for feature model synthesis," in *Proceedings of the Variability for You Workshop—variability made useful for everyone (VARY 2012)*, 2012. Available: http://vary2012.irisa.fr/VARY2012Proceedings.pdf

[She08] S. She, "Feature model mining," Master's thesis, University of Waterloo, Canada, 2008.

[SHTB07] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps, "Generic semantics of feature diagrams," *Computer Networks*, vol. 51, no. 2, 2007.

[Sil97] J. P. M. Silva, "On computing minimum size prime implicants," in *Proceedings of the International Workshop in Logic Synthesis (IWLS 1997)*, 1997.

[SJ04] K. Schmid and I. John, "A customizable approach to full lifecycle variability management," *Science of Computer Programming*, vol. 53, no. 3, 2004.

[SLB$^+$10] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki, "The variability model of the linux kernel," in *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2010)*, ser. ICB-Research Report. Universität Duisburg-Essen, 2010.

[SLB$^+$11] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki, "Reverse engineering feature models," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*. ACM, 2011.

[Sne96] G. Snelting, "Reengineering of configurations based on mathematical concept analysis," *Transactions On Software Engineering And Methodology*, vol. 5, no. 2, 1996.

[Sof93] Software Productivity Consortium Services Corporation, "Reuse-driven software processes guidebook, version 02.00.03," Tech. Rep. SPC-92019-CM, 1993.

[SP04] S. Subbarayan and D. Pradhan, "NiVER: Non-increasing variable elimination resolution for preprocessing sat instances," in *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*. Online Proceedings, 2004.

[SRA$^+$13] S. She, U. Ryssel, N. Andersen, A. Wąsowski, and K. Czarnecki, "Efficient synthesis of feature models," *Information and Software Technology*, 2013, submitted for review.

[SRC09] S. Segura and A. Ruiz-Cortés, "Benchmarking on the automated analyses of feature models: A preliminary roadmap," in *Proceedings of the 2nd International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2009)*, ser. ICB-Research Report. Universität Duisburg-Essen, 2009.

[SRG11] K. Schmid, R. Rabiser, and P. Gruenbacher, "A comparison of decision modeling approaches in product lines," in *Proceedings of the 5th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2011)*. ACM, 2011.

[SSSPS07] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk, "Is The Linux Kernel a Software Product Line?" in *Workshop on Open Source Software and Product Lines (OSSPL 2007) in Proceedings of the 11th International Software Product Lines Conference (SPLC 2007), Volume 2*. IEEE Computer Society, 2007.

[SV06] T. Stahl and M. Völter, *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.

[SWY75] G. Salton, A. Wong, and C.-S. Yang, "A vector space model for automatic indexing," *Communications of the ACM*, vol. 18, no. 11, 1975.

[TBK09] T. Thüm, D. Batory, and C. Kastner, "Reasoning about edits to feature models," in *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*. IEEE Computer Society, 2009.

[TBRC$^+$08] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and A. Jimenez, "FAMA framework," in *Proceedings of the 12th International Software Product Lines Conference (SPLC 2008)*. IEEE Computer Society, 2008.

[TJ09] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*. Springer, 2009.

[Tor09] E. Torlak, "A constraint solver for software engineering: finding models and cores of large relational specifications," Ph.D. dissertation, USA, 2009.

[vGBS01] J. van Gurp, J. Bosch, and M. Svahnberg, "On the notion of variability in software product lines," in *Proceedings of the 2nd Working IEEE / IFIP Conference on Software Architecture (WICSA 2001)*. IEEE Computer Society, 2001.

[WCR09] N. Weston, R. Chitchyan, and A. Rashid, "A framework for constructing semantically composable feature models from natural language requirements," in *Proceedings of the 13th International Software Product Lines Conference (SPLC 2009)*. ACM, 2009.

[Wik12] Wikipedia, "Configuration menu language — wikipedia, the free encyclopedia," 2012. Available: http://en.wikipedia.org/w/index.php?title=Configuration_Menu_Language&oldid=509241113

[WL99] D. M. Weiss and C. T. R. Lai, *Software product-line engineering: a family-based software development process*. Addison-Wesley, 1999.

[Woh00] C. Wohlin, *Experimentation in Software Engineering: An Introduction*, ser. The Kluwer International Series in Software Engineering. Kluwer Academic, 2000.

[WSB$^+$08] J. White, D. Schmidt, D. Benavides, P. Trinidad, and A. Cortés, "Automated diagnosis of product-line configuration errors in feature models," in *SPLC*, 2008.

[XHSC12] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki, "Generating range fixes for software configuration," in *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*. IEEE, 2012.

[ZB13] B. Zhang and M. Becker, "Mining complex feature correlations from software product line configurations," in *Proceedings of the 7th International*

Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2013). ACM, 2013.

[Zc] R. Zippel and contributors, "kconfig-language.txt," available in the kernel tree at kernel.org, retrieved December 11, 2012.

[Zha05] L. Zhang, "On subsumption removal and on-the-fly CNF simplification," in *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*. Springer, 2005.

[ZZR06] L. Zhao, M. J. Zaki, and N. Ramakrishnan, "BLOSOM: A framework for mining arbitrary boolean expressions over attribute sets," Tech. Rep. 06-05, 2006, available from http://www.cs.rpi.edu/research/pdf/06-05.pdf.

Appendix A

Kconfig Semantics

In this appendix, we present the formal semantics of Kconfig using denotational semantics [Sch86]. We first define an abstract syntax for Kconfig, then it's semantics domain, followed by valuation functions. The *semantic domains* the sets that are used as value spaces in the programming language. *Valuation functions* map a specification to directly to its meaning. This work was published as a technical note [SB10].

The semantics are modeled after the Kconfig specification and when the specification is unclear, the semantics are derived from the behaviour of the configurators such as *xconfig* and *menuconfig*. However, there are specifications that cause unwanted side effects in the configurator. These corner cases are not modeled in our semantics. For example, in case of the *reverse dependency* the documentation explicitly states the following shortcoming of the configurators:

> "*select* should be used with care. *select* will force a symbol to a value without visiting the dependencies. By abusing *select* you are able to select a symbol foo even if foo depends on bar that is not set." [Zc]

A.1 Semantic Domain

A configuration of a Kconfig model is an assignment of values $v \in \mathsf{Const}$ to config elements. Thus, the set of all possible configurations is defined as:

$$\mathsf{Confs} = \mathsf{Id} \rightarrow \lfloor \mathsf{Const} \rfloor \tag{A.1}$$

If $c \in \mathsf{Confs}$ and $x \in \mathsf{Id}$, we write $c(x)$ to refer to the value of identifier x under the configuration c. Now, we define the semantics of a Kconfig model in terms of sets of configurations. Thus, $\mathcal{P}(\mathsf{Confs})$ is our semantic domain. We define $[\![\cdot]\!]_{\mathsf{kconfig}}$ as the function that evaluates a Kconfig model and returns a set of valid configurations:

$$[\![\cdot]\!]_{\mathsf{kconfig}} \colon \mathsf{Kconfig} \to \mathcal{P}(\mathsf{Confs}) \tag{A.2}$$

A.2 Global Functions

We start with the definition of some functions used throughout the semantics. First, we define an interpretation of tristate values in boolean logic with function bool: $\mathsf{Tri} \to \mathsf{Bool}$ where $\mathsf{Bool} = \{T, F\}$:

$$\text{bool}(v) = \begin{cases} F & \text{iff } v = 0_t \\ T & \text{iff } v = 1_t \vee v = 2_t \end{cases} \tag{A.3}$$

Moreover, we define a function *access* : $(\mathsf{Id} \cup \mathsf{Const}) \times \mathsf{Confs} \to \mathsf{Const}$ that retrieves the value of either a constant or a symbol. When an identifier has the value of $\perp$ (to be defined in Equation A.7), then the *access* function returns the identifier itself in the form of a string:

$$\text{access}(iv, c) = \begin{cases} iv & \text{iff } iv \in \mathsf{Const} \vee (iv \in \mathsf{Id} \wedge c(iv) = \perp) \\ c(iv) & \text{otherwise} \end{cases} \tag{A.4}$$

Next, we define the function *toStr*: $\mathsf{Const} \to \mathsf{String}$ that models the translation of a constant to a string representation. Let $i \in \mathsf{Int}$, $h \in \mathsf{Hex}$ and $s \in \mathsf{String}$, in the following definition of *toStr*:

$$\begin{array}{ccc} \mathit{toStr}(0_\mathrm{t}) = \text{“n”} & \mathit{toStr}(1_\mathrm{t}) = \text{“m”} & \mathit{toStr}(2_\mathrm{t}) = \text{“y”} \\ \mathit{toStr}(i) = \text{“”} + i & \mathit{toStr}(h) = \text{“0x”} + h & \mathit{toStr}(s) = s \end{array} \tag{A.5}$$

where the + operator is string concatenation.

Finally, the function $eval \colon KExpr(\mathsf{Id}) \to \mathsf{Tri}$ describes the evaluation of a $KExpr$ in the Kconfig language. We define $eval$ recursively with $e_1, e_2 \in KExpr(\mathsf{Id})$ and

$iv, iv_x, iv_y \in \mathsf{Id} \cup \mathsf{Const}$:

$$\begin{aligned}
eval(iv_x = iv_y, c) &= \begin{cases} 2_t & \text{iff } toStr(\text{access}(iv_x, c)) = toStr(\text{access}(iv_y, c)) \\ 0_t & \text{otherwise} \end{cases} \\
eval(iv_x \neq iv_y, c) &= 2_t - eval(iv_x = iv_y, c) \\
eval(\text{not}\, e_1, c) &= 2_t - eval(e_1, c) \\
eval(e_1 \,\text{and}\, e_2, c) &= \min(eval(e_1, c), eval(e_2, c)) \\
eval(e_1 \,\text{or}\, e_2, c) &= \max(eval(e_1, c), eval(e_2, c)) \\
eval(iv, c) &= \begin{cases} v_{iv} & \text{iff } v_{iv} = \text{access}(iv, c) \wedge v_{iv} \in \mathsf{Tri} \\ 0_t & \text{otherwise} \end{cases}
\end{aligned} \tag{A.6}$$

A.3 Valuation Functions

Kconfig model. We begin by defining the $[[\cdot]]_{\mathsf{kconfig}}$. Given a Kconfig model $m \in \mathsf{Kconfig}$, the semantics of a model is the intersection of all denotations across the model, configs and choices. In other words, the set of valid configurations for a Kconfig model is those configurations that satisfy all denotations. $[[\cdot]]_{\mathsf{kconfig}} : \mathsf{Kconfig} \rightarrow \mathsf{Confs}$ is defined:

$$\begin{aligned}
[[m]]_{\mathsf{kconfig}} = \left(\bigcap_{n \in m_{\mathsf{config}}} [[n]]_{\mathsf{type}} \cap [[n]]_{\mathsf{bounds}} \cap [[n]]_{\mathsf{default}} \cap [[n]]_{\mathsf{range}} \right) \cap \left(\bigcap_{n \in m_{\mathsf{choice}}} [[n]]_{\mathsf{choice}} \right) \\
\cap\, [[m]]_{\mathsf{module}} \\
\cap\, [[m]]_{\mathsf{undeclared}}
\end{aligned} \tag{A.7}$$

Type. The first denotation pertains to the constraints imposed by a config's type. The type of a config restricts its valid values to those in its respective domain. $[[\cdot]]_{\mathsf{type}} : \mathsf{Configs} \rightarrow$

Confs is defined:

$$[[(n,t,\_,\_,\_,\_)]]_{\text{type}} = \begin{cases} \{c \in \text{Confs} \mid c(n) \in \text{Tri} \setminus \{1_t\}\} & \text{iff } t = \text{boolean} \\ \{c \in \text{Confs} \mid c(n) \in \text{Tri}\} & \text{iff } t = \text{tristate} \\ \{c \in \text{Confs} \mid c(n) \in \text{String}\} & \text{iff } t = \text{string} \\ \{c \in \text{Confs} \mid c(n) \in \text{Hex} \cup \{\text{“”}\}\} & \text{iff } t = \text{hex} \\ \{c \in \text{Confs} \mid c(n) \in \text{Int} \cup \{\text{“”}\}\} & \text{iff } t = \text{int} \end{cases} \tag{A.8}$$

Upper and lower bounds. Next, the bounds denotation models the lower and upper bounds of a config. The lower bound is determined by the evaluation of a config's reverse dependency. Recall that the reverse dependency models the behaviour of the *select* statement in the concrete syntax. The upper bound is defined by a config's prompt condition. This denotation has no effect on configs of type int, hex, or string since the the reverse dependency that determines a lower bound is 0_t by our well-formedness rules, and the $eval$ function returns 0_t when evaluating a value not in Tri. $[[\cdot]]_{\text{bounds}}$: Configs $\rightarrow$ Confs is defined:

$$[[(n,\_,pro,\_,rev,\_)]]_{\text{bounds}} = \\ \{c \in \text{Confs} \mid eval(c(n),c) \geq \text{Lower}(c) \wedge (\text{Upper}(c) < \text{Lower}(c) \vee eval(c(n),c) \leq \text{Upper})\} \tag{A.9}$$

where Lower(c) = $eval$(rev,c) and Upper(c) = $eval$(pro,c).

Defaults. Kconfig has support for setting a default expression for a config. The default expression interacts with the prompt condition that determines when the config is user-changeable. When the prompt condition is satisfied, then the user is free to set a value. However, when the prompt condition is not satisfied, the default determine the config's value. $[[\cdot]]_{\text{default}}$: Configs $\rightarrow$ Confs is defined:

$$[[(n,\_,\_,defs,rev,\_)]]_{\text{default}} = \\ \{c \in \text{Confs} \mid \text{bool}(eval(pro,c)) \vee c(n) = \max(eval(\text{default}(defs,c)), eval(rev,c))\} \tag{A.10}$$

where $default$: $\mathcal{P}$(Default) × Type × Confs → Const is a function that models the retrieval of a default. Recall that $defs$ is a list of defaults (and thus ordered). The effect of a default's value depends on the type of its defining config. If the config is $boolean$

or $tristate$, then the default value is evaluated to a value in Tri. Otherwise, the default value must be either an element of Const or Id. Let Nil be the empty list and :: be the list $cons$ operator. Let $t_{\mathsf{Tri}} \in \{\text{boolean}, \text{tristate}\}$ and $t_{\mathsf{Entry}} \in \{\text{int}, \text{hex}, \text{string}\}$. The $default$ function is defined recursively, so we begin by defining its base cases:

$$\begin{aligned} \text{default}(Nil, t_{\mathsf{Tri}}, c) &= 0_{\mathsf{t}} \\ \text{default}(Nil, t_{\mathsf{Entry}}, c) &= \text{``''} \end{aligned} \tag{A.11}$$

Equation A.11 states that given an empty list of defaults, we return 0_{t} if the type is either boolean or tristate, or the empty string for types int, hex or string. Next, we define the recursive rule. In the following equation, we decompose the list into its head and tail components. First, we describe the function for boolean and tristate type (recall that $\mathsf{Bool} \in \mathsf{Tri}$):

$$\text{default}(\,(e, cond) :: rest, t_{\mathsf{Tri}}, c) = \begin{cases} eval(e, c) & \text{if bool}(eval(cond, c)) \\ \text{default}(rest, t_{\mathsf{Tri}}, c) & \text{otherwise} \end{cases} \tag{A.12}$$

Now for the remaining types:

$$\text{default}(\,(e, cond) :: rest, t_{\mathsf{Entry}}, c) = \begin{cases} \text{access}(e, c) & \text{if bool}(eval(cond, c)) \\ \text{default}(rest, t_{\mathsf{Entry}}, c) & \text{otherwise} \end{cases} \tag{A.13}$$

Ranges. Ranges impose a lower and upper bound on the value of int or hex configs. $[\![\cdot]\!]_{\mathsf{range}} : \mathsf{Configs} \rightarrow \mathsf{Confs}$ is defined as:

$$\begin{aligned} [\![n, \_, \_, \_, \_, rngs)]\!]_{\mathsf{range}} = \{c \in \mathsf{Confs} \mid \ \forall (l, u, cond) \in rngs. \\ \text{bool}(eval(cond, c)) \rightarrow c(n) \geq \text{access}(l, c) \wedge c(n) \leq \text{access}(u, c)\} \end{aligned} \tag{A.14}$$

Choices. A choice restricts the number of members that can be selected (i.e. have a value greater than 0_{t}). The choice denotation, $[\![\cdot]\!]_{\mathsf{choice}} : \mathsf{Choices} \rightarrow \mathsf{Confs}$ is defined:

$$[\![(boolOrTri, isMand, prompt, mems)]\!]_{\mathsf{choice}} = \{c \in \mathsf{Confs} \mid \mathsf{Xor} \wedge \mathsf{BChoice} \wedge \mathsf{Mandatory}\} \tag{A.15}$$

where Xor defines the condition that one and only one member may be set to 2_{t}:

$$\mathsf{Xor} = \exists m_1 \in mems.\ (m_1 = 2_{\mathsf{t}}) \rightarrow (\forall m_2 \in mems \setminus \{m_1\}.\ m_2 = 0_{\mathsf{t}}) \tag{A.16}$$

If the choice is a boolean choice, then the only valid value for its members is 2_t. In combination with Xor, this defines that a boolean choice may have at most one member with a value not equal to 0_t and that member must be set to 2_t:

$$\mathsf{BChoice} = bool(eval(prompt, c)) \wedge (boolOrTri = \text{boolean}) \rightarrow \exists m \in mems.\ c(m) = 2_\text{t} \tag{A.17}$$

Finally, if the choice is *mandatory* and the prompt condition is satisfied, then one Boolean member can be selected:

$$\mathsf{Mandatory} = (boolOrTri = \text{boolean}) \rightarrow (\exists m_1 \in mems.\ (m_1 = 2_\text{t})) \tag{A.18}$$

Modules. A special MODULES config is used to specify support for modules in the kernel. Disabling MODULES disallows the 1_t state for configs and effectively turns all tristate configs into boolean configs. A special symbol m is used in expressions to identify a dependency on the MODULES feature in the concrete syntax. Configs with a dependency on m cannot be selected (i.e. must be set to 0_t) if MODULES is not selected. We assume that the special m identifier has been expanded to MODULES in the abstract syntax.

$$[[m]]_{\mathsf{module}} = \{c \in \mathsf{Confs} \mid (c(\textsc{modules}) = 0_t) \rightarrow \forall i \in \mathsf{Id}.\ c(i) \neq 1_t\} \tag{A.19}$$

Undeclared symbols. The Kconfig language supports references to symbols that are not declared in constraints. These undeclared symbols are assigned the special symbol $\perp$ in our semantics. The use of this symbol will become apparent in the definition of the *eval* function in Section A.2. The $[[\cdot]]_{\mathsf{undeclared}} : \mathsf{Kconfig} \rightarrow \mathcal{P}(\mathsf{Confs})$ denotation is defined as:

$$[[m]]_{\mathsf{undeclared}} = \{c \in \mathsf{Confs} \mid \forall x \in \mathsf{Id} \setminus \mathsf{Id}(m).\ c(x) = \perp\} \tag{A.20}$$

Appendix B

SPLOT Models

The following models from the SPLOT model repository were used in the comparison with Kconfig models in Chapter 4 and in the evaluation of FGE-DNF in Chapter 5:

a
Adams Car
Agencia de Propaganda
Agenda
Agile Simules
Aircraft PL
AllSports Feature Model
AndroidSPL
Applications
Asghar
Assistencia domiciliar
ATM Software
AudioPlayer
AvionFEatures
Basic Text Editor
Bicis_Feature_Model
Bicycle feature model
body comfort system
Car
Car demo
Card Product
Car PL
carpl
Carro
Carro Model
Carros
Car Software System
CCCMS
CD OD Semantic Variability
MKT
Mobile Applications Contents
Mobile Game
Mobile Games
Mobile Media Adriano Lages
MobileMedia_BrunoAquino
MobileMedia_BrunoMartins
Mobile Media Bruno R
MobileMedia_Conejero
Mobile Media
MobileMedia
MobileMedia_Gustavo
MobileMedia_HudsonSilvaBorges
MobileMedia_Marcelo
MobileMedia_Pedro_Pires
mobile_media_thom
Mobile Media until relase 7
MobileMedia_VitorSales
Mobile Phone
Mobile-Phone
Mobile Phone GF
Mobile phone (lenita)
Mobile Phone, until relase 5
Modelo de Caracteristicas de um Carro
Monitor Engine
Monitor_Engine_System
mon model
MoviesApp PL
MyFeatureModel

Cell Phone
Cellphone
CFDP Library
CIMS PL
Coff Feature Model
Connector PL
Construtora
Context Feature Model
Counter Strike Simple Feature Model
CTOS MODEL
daniel souza
DELL Laptop/Notebook Computers
desordem
Diagrama de Experimento
Digital_Video_System
Disciplina
Doorlopende reisverzekering
DS Sample
Dummy Car
EC2
Editor de imagem
Ejercicio
ELEC5619 - End User Model
ELEC5619 - Tootawl's Model
ELPS - ISE
E-Portal
E-Shop
Eshop-ufrj
ETRobocon
Example Mobile Phone
Example Product Line Course Using SPLOT
Fabrica de software
FAME-DBMS
FeatureModel1
FIRE-ALARM
FM1
G10-music player
GMF_Eclipse_Reuso_UFRJ
GoPhone Inc
Graph
Graph Product Line
GreenHouse_Control_and_Monitoring
GuideAgent
Guitarra
HDD Seagate
Henrique Gomes Nunes
MyFM
NCV_Simulation
Ökonomische Merkmale
ordem
Parking lot Manager
PFTest1
PFTest
Phone
Phone sple
PLeTs
Project Portal Suite
prueba
RE
Reference Management Softwares Feature Model
Respaldo de base de datos (Poda)
Rhiscom Process Model
RhiscomProcessModel
rtrt
SAL
Scheduler
SD Voter pattern feature model
Search_Engine_PL
sftcam
Sienna
Simaldores de Engenharia de Software
Simnumero
Simple Drawing Tools
Sistema de Informes por movil
Small Graph Product Line
SmartHome- Simplified Sample
SmartHome_vConejero_r1
Smartphone-SPL
SMS a Cobrar _ Resposta paga
SoftwareClassification
Software Engineering Course
Software Stack
SPL SimulES, PnP
Stacja Pogodowa
Stack PL
stcam
StoreFrontSystem
tablets
Tata Cars
Technical Solution Process
Telecommunication_System
Test

HIS
Homepage based on wiki
i18n
IDE4OCL
iNavBasic
Insurance Policy
Insurance_Product
IntelligentTutoringSystem
ISA11_12_pgr
isolation
ISPW-6
James
JCalc_LPS
Jogo de Engenharia de Software
Jogo de Tiros
JPlug
Key_Word_In_Context_index_systems
KIModel
Linea de Experimentos
Linha de produtos - SimulES e PnP
Loja Virtual
Lucas - pnp
Mauricio AlfÃĭrez
MiniRPG
MiniTV
Test Env
test_fm
testISA2012
Text Editor
Text_Editor
Thread
Toyota Feature Model
Transmission
TrialADC
TV-Series Shirt
UML-IDE-TOOL
UPL
VeÃculo
Venza
Virtual_Office_of_the_Future
VMS
VOD feature model
VODPlayer
VOTE4ME Serveur
WB2
WB
Weather Station
Web Content Delivery
Web Game