



Technische  
Universität  
Braunschweig



# The Role of Complex Constraints in Feature Modeling

Master's Thesis

Alexander Knüppel

July 5, 2016

Institute of Software Engineering and Automotive Informatics  
Prof. Dr.-Ing. Ina Schaefer

Supervision  
Dr.-Ing. Thomas Thüm

at

Technische Universität Carolo-Wilhelmina zu Braunschweig





Technische  
Universität  
Braunschweig



# The Role of Complex Constraints in Feature Modeling

Masterarbeit

Zur Erlangung des akademischen Grades Master of  
Science

Alexander Knüppel

5. Juli 2016

Institut für Softwaretechnik und Fahrzeuginformatik  
Prof. Dr.-Ing. Ina Schaefer

Betreuung  
Dr.-Ing. Thomas Thüm

Technische Universität Carolo-Wilhelmina zu Braunschweig



# Abstract

Feature modeling is a method to compactly capture commonality and variability of a software product line. Multiple feature modeling languages have been proposed that evolved over the last decades to become more expressive in syntax and semantics. Most of today's languages are capable of utilizing arbitrary propositional formulas in cross-tree constraints, denoted as complex constraints, a mechanism enabling complete expressiveness. However, many of today's publications and feature model applications are targeting older, less expressive languages, due to their history and long domination in the product-line community. We present a study on the importance of complex constraints in feature modeling. Furthermore, to build a bridge between feature models using complex constraints and methods lacking support for complex constraints, we present a sound refactoring of complex constraints, discuss preconditions that must be met, and conduct empirical experiments on real-world feature models to evaluate its usefulness and scalability.



# Zusammenfassung

Feature-Modellierung ist eine Methode, um Gemeinsamkeiten und Variabilität einer Produktlinie in der Softwareentwicklung kompakt darzustellen. Über die letzten Jahrzehnte wurden verschiedene Sprachen für die Feature-Modellierung vorgestellt, die sich sowohl syntaktisch als auch semantisch voneinander unterscheiden. Viele der heute eingesetzten Sprachen unterstützen die Angabe beliebiger logischer Ausdrücke, so genannte komplexe Constraints, um orthogonale Beziehungen zwischen Features festzulegen. Komplexe Constraints geben einer Feature-Modellierungssprache volle Ausdrucksmächtigkeit. Allerdings werden heutzutage immer noch eine große Menge an Methoden und Applikationen publiziert, die auf bekanntere Sprachen mit weniger Ausdrucksmächtigkeit aufbauen. In dieser Arbeit untersuchen wir die Notwendigkeit von komplexen Constraints in der Feature Modellierung. Zudem überbrücken wir die Problematik zwischen neueren Sprachen mit komplexen Constraints und Methoden und Tools, die auf älteren Sprachen aufbauen, indem wir einen Ansatz präsentieren, um komplexe Constraints in Feature Modellen zu eliminieren. Wir diskutieren Vorbedingungen und evaluieren unseren Ansatz hinsichtlich Nutzen und Skalierbarkeit an Feature Modellen aus der realen Welt.





# Acknowledgement

I would first like to thank my thesis advisor Dr.-Ing. Thomas Thüm. The door was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this thesis to be my own work, but steered me in the right direction whenever he thought I needed it. His careful reviews of my drafts and impeccable remarks allowed me to constantly improve my research skills during the creation of this thesis.

I would also like to thank Prof. Dr. Ina Schaefer and her team at the institute of software engineering and automotive informatics. Their input and substantial feedback guided me in the right direction and broadened my view on the thesis' topic.

A very special thanks goes to Jens Meinicke, Malte Lochau, Stephan Mennicke, and Reimar Schröter, whose general ideas about complex constraints and the basic principle of refactoring them laid the foundation for this thesis.

Finally, I must express my very profound gratitude towards my family and friends for their continuous support.



# Contents

<b>List of Figures</b>	<b>iii</b>
<b>List of Tables</b>	<b>v</b>
<b>List of Code Listings</b>	<b>vii</b>
<b>List of Algorithms</b>	<b>ix</b>
<b>1. Introduction</b>	<b>1</b>
<b>2. Constraints in Feature Modeling</b>	<b>5</b>
2.1. Software Product Lines . . . . .	5
2.1.1. Preprocessor-Based Variability . . . . .	6
2.1.2. Feature Modeling . . . . .	7
2.1.3. Domain Engineering . . . . .	8
2.2. A Survey of Feature Modeling Languages . . . . .	10
2.2.1. Graphical Representations of Feature Models . . . . .	11
2.2.2. Textual Representations of Feature Models . . . . .	14
2.2.3. Comparison of Feature Model Representations . . . . .	18
2.3. Applications of Feature Models . . . . .	20
2.4. Summary . . . . .	26
<b>3. Formal Foundations of Feature Models</b>	<b>27</b>
3.1. Motivation for a Formal System . . . . .	27
3.2. A Formal Semantics for Feature Modeling Languages . . . . .	28
3.2.1. Defining an Abstract Syntax . . . . .	28
3.2.2. Semantic Domain: Giving Meaning to Syntax . . . . .	32
3.2.3. Capturing Feature Model Extensions . . . . .	34
3.2.4. Mapping Feature Models to Propositional Logic . . . . .	36
3.3. Expressive Power of Feature Models . . . . .	37
3.4. Summary . . . . .	42
<b>4. Eliminating Complex Constraints</b>	<b>45</b>
4.1. General Refactoring of Feature Models . . . . .	45
4.2. Refactoring Group Cardinality . . . . .	51
4.3. Refactoring Complex Constraints . . . . .	52
4.3.1. Pseudo-Complex Constraints and Trivial Simplifications . . . . .	53
4.3.2. Refactoring Using Negation Normal Form . . . . .	54

4.3.3. Refactoring Using Conjunctive Normal Form . . . . .	59
4.3.4. One-to-One Correspondence of Configurations . . . . .	61
4.4. Summary . . . . .	62
<b>5. Eliminating Complex Constraints with FeatureIDE</b>	<b>65</b>
5.1. Overview . . . . .	65
5.2. Preprocessing Phase . . . . .	67
5.3. Choosing a Conversion Strategy . . . . .	68
5.4. Implementing an Exporter for the FAMA File Format . . . . .	72
5.5. Summary . . . . .	73
<b>6. Evaluation</b>	<b>75</b>
6.1. Methodology . . . . .	75
6.2. Experimental Results . . . . .	77
6.2.1. Constraint Classification . . . . .	78
6.2.2. Performance Analysis . . . . .	80
6.2.3. Scalability . . . . .	86
6.3. Threats to Validity . . . . .	89
6.4. Summary . . . . .	90
<b>7. Related Work</b>	<b>91</b>
<b>8. Conclusion</b>	<b>93</b>
<b>9. Future Work</b>	<b>95</b>
<b>Appendix A. Evaluation Results</b>	<b>97</b>
<b>Bibliography</b>	<b>107</b>

# List of Figures

2.1. Stack Example Using the C Preprocessor to Implement Variability . . . . .	7
2.2. Example Feature Diagram of a Mobile Phone Product Line . . . . .	8
2.3. Overview of the Domain and Application Engineering Process . . . . .	9
2.4. Syntax of a FODA Feature Diagram (Adopted from Kang et al. (1990)) . . . . .	11
2.5. Syntax of a FeatuRSEB Feature Diagram . . . . .	12
2.6. Syntax of a Feature Diagram Proposed by Riebisch et al. (2002) . . . . .	13
2.7. Syntax of a Feature Model Proposed by Van Deursen and Klint (2002) . . . . .	14
2.8. Syntax of a GUIDSL Feature Model . . . . .	15
2.9. Syntax of a Feature Model in SXFM . . . . .	15
2.10. Syntax of a FAMA Feature Model . . . . .	16
2.11. Syntax of a Feature Model in TVL . . . . .	16
2.12. Syntax of a VELVET Feature Model . . . . .	17
2.13. Syntax of a FAMILIAR Feature Model . . . . .	17
2.14. Syntax of a CLAFTER Feature Model . . . . .	18
2.15. Frequency of Complex Constraints in Feature Model Applications . . . . .	24
2.16. Frequency of Complex Constraints Over Time . . . . .	26
3.1. Three Syntactically Different Yet Equivalent Feature Models . . . . .	34
3.2. Shortened Version of the Mobile Phone Product Line . . . . .	37
3.3. Feature Model with two Complex Constraints . . . . .	39
3.4. Difference in Expressive Power of Feature Models . . . . .	41
3.5. Example of a Potential Feature Model in $\mathcal{L}_{RBFM}$ . . . . .	42
3.6. Semantic Domain of Feature Modeling Languages . . . . .	43
4.1. Three Semantically Equivalent Feature Models . . . . .	46
4.2. Refactoring of a Feature Model $m$ . . . . .	47
4.3. Example of a Feature Model Change . . . . .	48
4.4. Example of an Original Feature Model and an Abstract Subtree . . . . .	49
4.5. Transformation of a Propositional Formula to an Abstract Subtree . . . . .	50
4.6. Replacing a Complex Constraint by an Equivalent Abstract Subtree. . . . .	51
4.7. Problem of Refactoring Group Cardinality . . . . .	51
4.8. Eliminating Group Cardinality . . . . .	52
4.9. Resulting Abstract Subtree Based on Conjunctive Normal Form . . . . .	60
4.10. Resulting Feature Model when a Coherent Refactoring is Performed . . . . .	62
5.1. Feature Modeling Project in FeatureIDE with Context Menu . . . . .	66
5.2. Activity Diagram of the Refactoring Process . . . . .	66

5.3. Identifying Redundant Constraints . . . . .	68
5.4. A Wizard for the Conversion Process . . . . .	69
5.5. Relationships Between Conversion Strategies . . . . .	71
6.1. Constraint Classification for each Feature Model . . . . .	78
6.3. Number in Literals of Strict-Complex Constraints for each Feature Model . . . . .	79
6.4. Total Time Measured for Eliminating Complex Constraints (Incoherent Refactoring)	81
6.5. Total Time Measured for Eliminating Complex Constraints (Coherent Refactoring)	82
6.6. Time Measured to Identify Redundant and Tautological Constraints. . . . .	82
6.7. Time Measured to Process Pseudo-Complex Constraints . . . . .	83
6.8. Time Measured to Process Strict-Complex Constraints . . . . .	84
6.9. Time Measured to Construct and Compose all Abstract Subtrees . . . . .	84
6.10. Average Heap Allocation for the Linux Kernel for each Conversion Strategy . . . . .	85
6.11. Heap Allocation of a Refactoring of the Linux Kernel Using Conjunctive Normal Form	86
6.12. Increase in Features after Refactoring using the Combined Method . . . . .	87
6.13. Increase in Constraints after Refactoring using the Combined Method . . . . .	87

# List of Tables

2.1. Comparison of Feature Modeling Languages . . . . .	19
2.2. Summary of Reviewed Publications for Feature Model Applications . . . . .	25
3.1. Translation Between Group Cardinality and Concrete Syntax in a Feature Diagram	29
3.2. Abstract Representation of the Mobile Phone Product Line . . . . .	31
3.3. Comparison of Program Variants and Configurations . . . . .	34
3.4. Mapping of Feature Models to Propositional Logic . . . . .	36
4.1. Overview of Defined Refactoring Algorithms . . . . .	63
5.1. Publicly Available Methods of Class ComplexConstraintConverter . . . . .	67
5.2. Exporting a Feature Model into the FAMA File Format . . . . .	73
6.1. Notation Used for Evaluation . . . . .	76
6.2. Representative Sample Set of Evaluated Feature Models . . . . .	77
A.1. Notation Used for Evaluation . . . . .	98
A.2. Statistical Properties of Evaluated Feature Models . . . . .	99
A.3. Time Measurements for Incoherent Refactoring . . . . .	100
A.4. Time Measurements for Coherent Refactoring . . . . .	101
A.5. Increase in Features and Constraints After Refactoring . . . . .	102
A.6. Average Statistical Properties of Generated Feature Models . . . . .	103
A.7. Summary of Computed Times for Generated Feature Models . . . . .	104
A.8. Summary of Increase in Features and Constraints for generated Feature Models . .	105





# List of Code Listings

5.1. IConverterStrategy Interface . . . . .	68
5.2. Preprocessing Method of NNFCConverter . . . . .	69
5.3. Preprocessing Method of CombinedConverter . . . . .	71



# List of Algorithms

4.1. Converting a Propositional Formula to Negation Normal Form . . . . .	55
4.2. Converting a Propositional Formula to Conjunctive Normal Form . . . . .	60



# 1 Introduction

A major challenge in today's software engineering process is the cost-efficient reuse of software artifacts. Obviously, reuse not only helps in drastically decreasing the cost of software development and maintenance, but also helps in increasing the software's quality. *Software product-line engineering* has evolved to a new software paradigm embracing the idea of mass customization and software reuse (Pohl et al., 2005). A software product line is a family of related software products that share a common code base and are each composed by a legal combination of software artifacts (Czarnecki and Eisenecker, 2000). Those software artifacts are termed *features*. Kang et al. (1990) define a feature as a "prominent or distinctive user-visible aspect, quality, or characteristic of a software or system that is relevant to some stakeholder".

Currently, the most frequently used family of approaches in industrial practice to represent a software product line is *feature modeling*, originally introduced in the Feature-Oriented Domain Analysis (FODA) method by Kang et al. (1990). *Feature modeling* captures valid configuration options of a software product line in a hierarchically arranged set of features. Feature models allow practitioners to describe commonality (i.e., characteristics that all products share) and variability (i.e., the differences between products) of a software product line in terms of features.

In its *basic* form (Benavides et al., 2010) there are two types of relationships among features in a feature model. The first type is a parental relationship (i.e., decomposition) between a feature and its child features and consists of either an *optional/mandatory relation*, *alternative relation*, or *or relation*. The second type, namely *cross-tree constraints*, symbols a dependency of two features in the final product beyond decomposition. There are two kinds of cross-tree constraints in a basic feature model: either the existence of one feature requires the existence of another feature (*requires relation*) or a mutual exclusion between two features (*excludes relation*) (Kang et al., 1990). We call *requires* and *excludes* relations *simple constraints*. More *complex constraints* have later been proposed in literature (Batory, 2005) and allow the existence of arbitrary propositional formulas among features in feature models.

In this thesis, we focus on the distinction between simple and complex constraints. Many of today's feature models consist of at least several hundreds of features. We argue that the use of only simple constraints may complicate the modeling process. Moreover, complex constraints were introduced with the clear objective to deal with the enormous and ever-increasing complexity of feature models. However, over the last decades, several new or extended feature modeling languages, both graphical and textual, have been proposed, which do not consider complex constraints at all (Kang et al., 1990; Griss et al., 1998; Kang et al., 1998; Czarnecki and Eisenecker, 2000; Gorp et al., 2001; Riebisch et al., 2002; Eriksson et al., 2005; Benavides et al., 2005). Benavides et al. (2010) even proclaim that cross-tree constraints "are typically inclusion or exclusion statements", i.e., simple constraints, but as far as we know, this has not been studied empirically yet.

Nevertheless, there are also numerous languages that increase the power of feature models through complex constraints (Batory, 2005; Kästner et al., 2009; Mendonça et al., 2009). Starting

from the distinction between simple and complex constraints, we found one particular question of interest to be left unanswered by today's research: how important is the role of complex constraints in feature modeling?

We identified that several publications on feature modeling applications only consider languages close to basic feature models, i.e., without complex constraints. These applications include the generation of feature model configurations (White et al., 2014), the generation of feature models that are particular hard to analyze (Segura et al., 2014), extracting models from configurations (Al-Msie 'deen et al., 2014), and optimal product derivation based on non-functional properties (Guo et al., 2011). Another challenge is the automated analysis of feature models (Benavides et al., 2010). BETTY, for example, is a framework that aims at "[enabling] the automated detection of faults in feature model analysis tools" (Segura et al., 2012). However, BETTY can only work with feature models using simple constraints. Feature modeling practitioners working with complex constraints may not be able to make use of one of these approaches and tools, or the results and insights BETTY offers.

We think that a starting point to this problem is automated elimination of complex constraints and the accompanied feature model refactoring to a basic feature model. However, Schobbens et al. (2007) argue that the language of basic feature models is not expressive complete (i.e., there exists a product line for which no feature model can be found), which prevents a transformation from feature models with complex constraints to feature models with only simple constraints. We argue that with the concept of abstract features (i.e., features with no code mapping used for decomposition and as part of cross-tree constraints), feature models with simple constraints can become expressive complete.

Nevertheless, it is unclear if an efficient refactoring can be constructed and if these refactorings are still useful in practice. If such a refactoring does not scale well, we have indeed highlighted the expressive power of complex constraints, hence we are motivated to conduct empirical experiments to examine the role of complex constraints in feature modeling.

## Problem Statement

The BETTY example shows that even modern tooling can rely on a basic feature modeling language. The following four research questions, left mostly unanswered by related work, motivate us to address this problem further.

**Research Question 1 (RQ1).** *Which feature modeling languages, methods, and tools do support or do not support complex constraints, and how valuable are complex constraints in general?*

**Research Question 2 (RQ2).** *Under which circumstances is it possible to eliminate complex constraints?*

**Research Question 3 (RQ3).** *To what extent are complex constraints used in real-world feature models?*

**Research Question 4 (RQ4).** *What are the costs of eliminating complex constraints?*

## Research Goals

The main goal of this thesis is a sound refactoring of feature models containing complex constraints to feature models containing only simple constraints. We will see that the concept of abstract features will play a big role in the refactoring. The second big major goal is an investigation of the portion of real-world feature models, containing arbitrary constraints, and to what extent approaches, such as feature model analysis or optimal product derivation, can only deal with requires and excludes constraints. Namely, this thesis consists of the following goals.

**Research Goal 1 (RG1).** *Evidence that real-world feature models rely on complex cross-tree constraints but many published methods for feature models do not.* Whether the problem is worth studying remains. We will collect evidence that many feature models in practice already use arbitrary cross-tree constraints, yet a big portion of modern tools and research contributions in this domain assumes that feature models only contain requires and excludes constraints. 9pt] **Realization.** We survey several feature modeling languages and take on a short literature review on methods and tooling in the domain of feature models. Moreover, we examine the portion of complex constraints in real-world feature models in an empirical study.

**Research Goal 2 (RG2).** *A proof that feature model trees with only requires and excludes constraints but abstract features are expressive complete.* The case without explicit abstract features has already been studied by Schobbens et al. (2007). The general conclusion is that these kinds of feature models are not expressive complete. With a formal semantics and the use of abstract features we provide a formal proof that every product line can be modeled by a feature model language only using simple constraints.

**Realization.** We formalize a general formal semantics targeting the diversity of various feature modeling languages. This formal framework will help us to highlight the expressive value of complex constraints and abstract features, and helps us also in examining the expressive restriction for other languages (e.g., basic feature models).

**Research Goal 3 (RG3).** *An algorithm for refactoring feature models with arbitrary constraints to feature models with only requires and excludes constraints.* From the conclusion drawn above we want to present an algorithm for the refactoring process including a formal proof of the correctness of the transformation.

**Realization.** Based on our formal semantics, we are able to formulate a step-by-step refactoring algorithm and to prove its correctness formally. We introduce the concept of *abstract subtrees* that we exploit to build semantically equivalent feature model structures from cross-tree constraints.

**Research Goal 4 (RG4).** *An implementation of the refactoring algorithm in FEATUREIDE and an evaluation of its scalability.* To evaluate our approach in terms of speed, memory usage, and scalability with real-world examples, we want to implement our algorithm in FEATUREIDE, an Eclipse-based framework for the whole process of software product-line engineering. Moreover, it is important to empirically evaluate whether our algorithm is useful for large feature models.

**Realization.** The implementation will consist of a basic implementation eliminating complex constraints of feature models in FEATUREIDE. Moreover, we use this implementation to extend FEATUREIDE with a real application scenario: an exporter for the FAMA file format, a format for basic feature models. Our implementation is also eventually used to conduct experiments to empirically evaluate our algorithm and to answer part of the research questions formulated above. We therefore use a representative sample set of large and small feature models, and artificial generated feature models of different sizes.

## Reader's Guide

Chapter 2 provides an introduction to feature models and software product lines, and a survey on the evolution of feature modeling languages. We also provide evidence from the real-world that the underlying problem of missing support for complex constraints needs to be investigated. In Chapter 3, we develop a general formal semantics for feature models and explore the expressiveness of different feature modeling characteristics. Chapter 4 covers our formulation of a refactoring algorithm to eliminate complex constraints, including a formal proof of its correctness. The implementation in FEATUREIDE is described in Chapter 5. We empirically evaluate the usefulness and scalability of our algorithm in Chapter 6. Related work is discussed in Chapter 7. Finally, we conclude this thesis and give an outlook to future work in Chapter 8 and Chapter 9, respectively, and present the collected results of our empirical evaluation in Appendix A.



# 2 Constraints in Feature Modeling

The subject of this thesis are complex constraints in feature models. This chapter's objective is twofold. First, in Section 2.1, we provide the necessary background on software product lines, feature modeling, and domain engineering. Second, we are motivated by *RQ1* to analyze which feature modeling languages, methods, and tools use complex constraint and which use only simple constraints to reason about the acceptance of complex constraints in feature modeling. We survey different feature modeling languages in Section 2.2, and collect information about the use of complex constraints in the literature in Section 2.3.

## 2.1. Software Product Lines

In today's software engineering process, more and more companies are targeting the demands of their prospective customers by creating a software product line (i.e., a set of software products as variations of a common code base (Clements and Northrop, 2001)) rather than a single software product.

For example, a company developing an infotainment system in the automotive marketplace will not be competitive if it sells its software product to only one client tailored to only one particular car model. Different automotive companies have different requirements, specifications, and needs. Creating individual products for every car model would result in massive costs. Another prominent example can be found in embedded systems, where software often has a critical responsibility. *Inboard software* is embedded software running in aircraft, spacecraft, or satellites, and thus needs to work as intended, or otherwise can lead to catastrophic behavior. The goal of software product lines is to reuse reliable and tested software artifacts in related software products.

To address these problems, cost-efficient manufacturing of software from reusable parts has emerged into a new software paradigm called *software product-line engineering*. Pohl et al. (2005, p. 14) define software product line engineering as follows.

*"Software product line engineering is a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customization."*

Mass customization in product-line engineering targets similar products in the same *domain*. A specific process called *domain engineering* (Pohl et al., 2005) helps in identifying reusable software artifact, as a first step to implement a product line, and also to define commonalities and variability of the underlying domain. The result is then captured in a *variability model* that offers *variation points*. Variation points enable the adjustment of similar software assets to one context to derive a

customized software product. For example, a software asset sending messages over a network can vary in its encryption method, i.e., symmetric key, public key, or other means of encryption. The type of encryption can be seen as a variation point.

Reusing software artifacts during product development is done in a process called *application engineering* (Pohl et al., 2005). In this process, selected software artifacts are merged together to generate a desired software product. The selection is called a *configuration*.

The most typical approach on representing a product line is through compiler-directives such as the well-known *#ifdef* from the C preprocessor (Svahnberg and Bosch, 2000; Gacek and Anastasopoulos, 2001), or other means of conditional compilation. A basic example of a product line using the C preprocessor is described in Section 2.1.1. In Section 2.1.2, we give an informal introduction to feature models, a prominent kind of a variation model and subject of this thesis. Finally, in Section 2.1.3, we give an overview on domain engineering, the first stage in developing a software product line.

### 2.1.1. Preprocessor-Based Variability

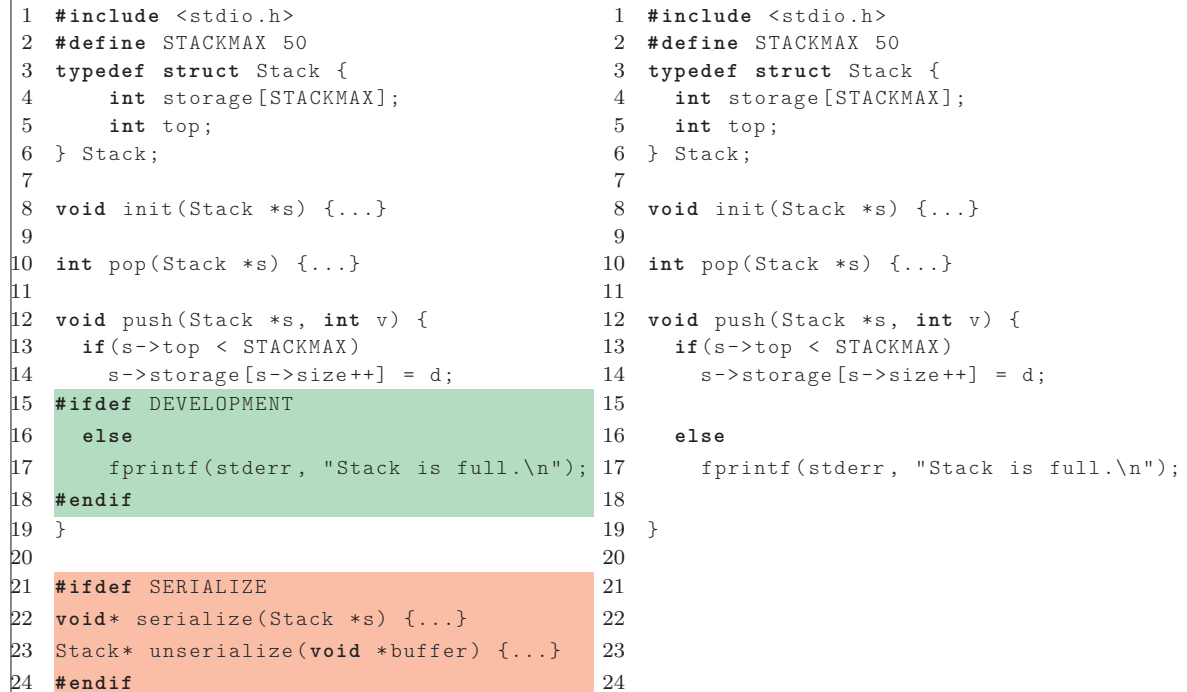
As mentioned in the previous section, a typical approach on implementing variability in source code is through preprocessor directives. A preprocessor is a tool that runs over the source code before compilation and removes parts wrapped between specific annotations (directives) (Apel et al., 2013, p. 110ff.). For example, the C preprocessor allows code parts to be wrapped between *#ifdef* and *#endif*. The code in between is removed when following symbol after *#ifdef* is not defined. Figure 2.1 exemplifies a variability through preprocessor directives of a simple *stack* implementation.

In this example, *DEVELOPMENT* is defined whereas *SERIALIZE* is not. Hence, only the part surrounded by *#ifdef DEVELOPMENT ... #endif* is retained (highlighted in green). The right-hand side depicts the generated source code after preprocessing and before compilation. This variability is offered by defining symbols either directly in the source code through *#define*, or by giving the compiler a list of defined symbols, mostly through the prefix *-D* followed by the symbols name (e.g., *-DDEVELOPMENT* in this case).

In product-line engineering, we speak of *features* instead of symbols. The exact definition of a *feature* varies in literature depending on the approach to implement variability as shown by Classen et al. (2008). For example, Batory et al. (2006) define a feature as an "*increment in program functionality*" which aligns with the perspective of feature-oriented programming; another mechanism to offer variability as opposed to a preprocessor. We stick with the notion by Kang et al. (1990) and see a feature as a specific characteristic or aspect that can be added to the software base to generate a new program variant.

One of the most prominent and complex examples for a preprocessor-based variability model with many different variants is the Linux kernel (Bovet and Cesati, 2005). In one of its larger versions, the Linux kernel comprises over 10,000 features, which can be selected to generate specific kernel images (i.e., software products).

Preprocessing is an easy way to develop a product line, yet this technique has obviously some weak points. For example, reusing and even identifying software assets is not a clear task. Besides the preprocessor, there are other common options to implement variability, i.e., *Runtime Parameters*,



```

1  #include <stdio.h>
2  #define STACKMAX 50
3  typedef struct Stack {
4      int storage[STACKMAX];
5      int top;
6  } Stack;
7
8  void init(Stack *s) {...}
9
10 int pop(Stack *s) {...}
11
12 void push(Stack *s, int v) {
13     if(s->top < STACKMAX)
14         s->storage[s->size++] = d;
15     #ifdef DEVELOPMENT
16     else
17         fprintf(stderr, "Stack is full.\n");
18     #endif
19 }
20
21 #ifdef SERIALIZE
22 void* serialize(Stack *s) {...}
23 Stack* unserialize(void *buffer) {...}
24 #endif

```

```

1  #include <stdio.h>
2  #define STACKMAX 50
3  typedef struct Stack {
4      int storage[STACKMAX];
5      int top;
6  } Stack;
7
8  void init(Stack *s) {...}
9
10 int pop(Stack *s) {...}
11
12 void push(Stack *s, int v) {
13     if(s->top < STACKMAX)
14         s->storage[s->size++] = d;
15     else
16         fprintf(stderr, "Stack is full.\n");
17 }
18
19 }
20
21
22
23
24

```

Figure 2.1.: Stack Example Using the C Preprocessor to Implement Variability

*Design Patterns, Frameworks, Components and Services, Build and Version-Control Systems, and Feature-Oriented and Aspect-Oriented Programming.* All of these are further described by [Apel et al. \(2013\)](#).

### 2.1.2. Feature Modeling

*Feature models* were first introduced in the Feature-Oriented Domain Analysis (FODA) by [Kang et al. \(1990\)](#) and became an important means for modeling variability in software product-line engineering. A feature model is a simple and hierarchically organized model that captures commonality and variability of a software product line in terms of features.

Feature models are used to represent all possible configurations in a software product line. There are several representations of feature models. The graphical representation of a feature model is called a *feature diagram* ([Kang et al., 1990](#)). An example of a feature diagram modeled in FEATUREIDE ([Kästner et al., 2009](#)) is given in Figure 2.2. A feature diagram is a tree where each feature has a parent feature except for the *root feature*. Each feature is decomposed into one or more features except for terminal features. The selection of features to derive a product underlies rules given by the feature diagram's notation. As indicated in Figure 2.2, we distinguish between four decomposition types:

- **Or:** at least one of its sub-features must be included (cf. sub-features of *Media*).
- **Alternative:** exactly one of its sub-features must be included (cf. sub-features of *Screen*).

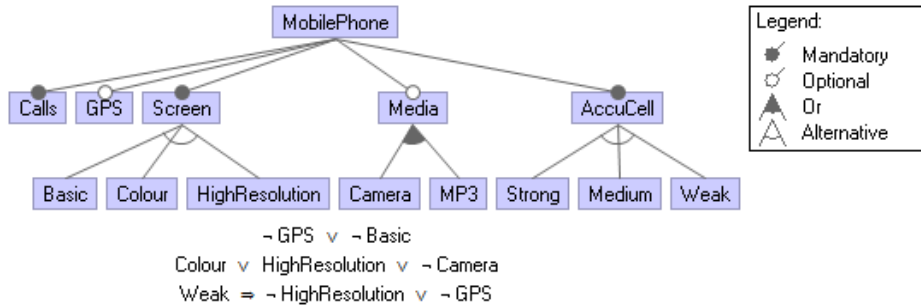


Figure 2.2.: Example Feature Diagram of a Mobile Phone Product Line

- **Mandatory and Optional:** Mandatory and optional features are typically part of an *and decomposition*. Mandatory sub-features must be included and optional features can be included (cf. features *Calls* and *GPS*).

The inclusion of any feature assumes that its parent is also included. Furthermore, *cross-tree constraints* can be specified to define further relationships between features not in parental relationship. In Figure 2.2, cross-tree constraints are arbitrary propositional formulas that must evaluate to true.

The use of arbitrary propositional formulas is not self-evident. Many notations, as investigated in Section 2.2, are restricted to two special kinds of cross-tree constraints, namely *requires* and *excludes* constraints. Given two features *A* and *B*,

- **A requires B:** if *A* is included then *B* must also be included.
- **A excludes B:** if *A* is included then *B* is not allowed to be included, and vice versa.

Hence, we make the distinction between arbitrary propositional formulas and these two special cross-tree constraints. Every constraint representing a *requires* constraint or *excludes* constraint is called a *simple constraint*. A textual constraint using arbitrary propositional formula is called a *complex constraint*. It has to be noted that this concept does not depend on the concrete syntax of a feature modeling notation. For example, consider a *requires* constraint between features *A* and *B*, then  $(A \implies B)$ ,  $(\neg A \vee B)$ , or a graphical representation (e.g., a drawn arrow from feature *A* to feature *B*) are all equivalent simple constraints. In contrast,  $(A \vee B)$  is a complex constraint.<sup>1</sup>

We also distinguish between features that influence program functionality (i.e., result in different source code if included) and features that are mainly used for modeling and decomposition (i.e., they do not occur in an `#ifdef` in preprocessor-based variability). The former features are called *concrete features*, whereas the latter features are called *abstract features* (Thüm et al., 2011).

### 2.1.3. Domain Engineering

In Section 2.1.2, we introduced feature modeling. In product-line engineering, the process eventually resulting in a feature model is called *domain engineering*. When implementing a software product

<sup>1</sup>Opposed to the definitions here, some authors use the term *basic constraint* to refer to a subset of arbitrary propositional formula, and the term *complex constraint* to refer to first order logic or to constraints including attributes (Classen et al., 2011).

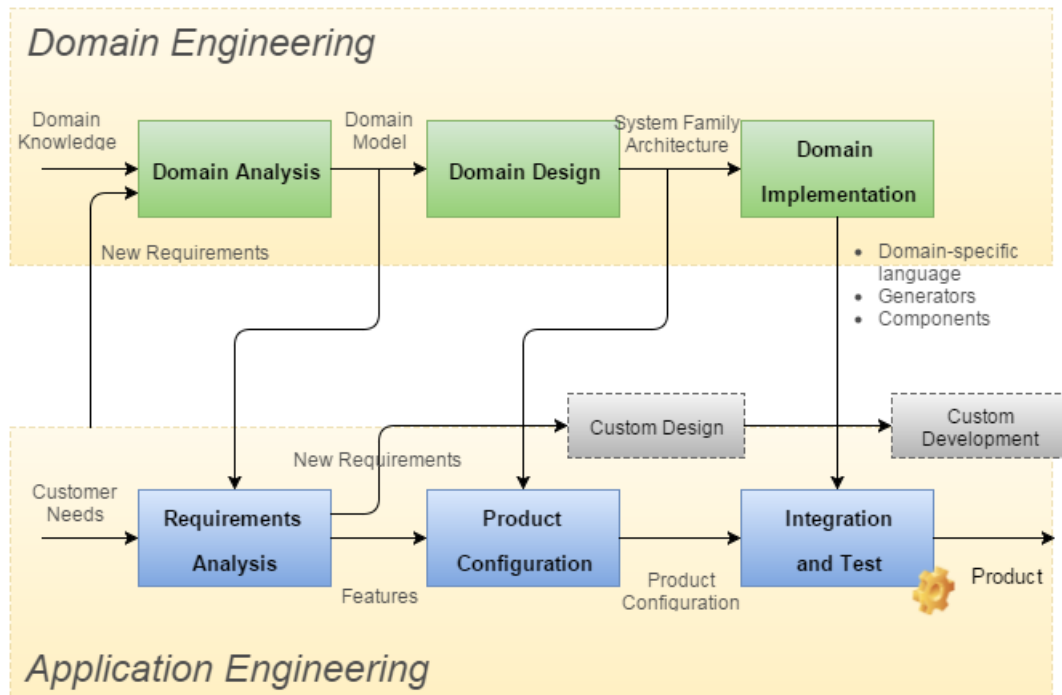


Figure 2.3.: Overview of the Domain and Application Engineering Process (Adopted from Pohl et al. (2005))

line, a common *software platform* for gathering software assets is used. The term *platform* originally arose from a customer's need to have an individualized product in the industry sector. Therefore, especially in the car industry, many companies started to introduce *common platforms* for their different types of models to allow the cost-efficient exchange of certain parts (Pohl et al., 2005). Meyer and Lehnerd (1997) define a platform as "a set of common components, modules, or parts from which a stream of derivative products can be efficiently developed and launched". For example, a different suspension system can be used in the same car model to satisfy a customer's demands.

Similar, a software platform consists of software subsystems (also called software assets) and interfaces that form a common structure and enable the derivation of a number of software products. These subsystems can be classified according to their functionality. For example, a hotel reservation system may depend on a database system, a GUI library, a numerical library for statistics, etc. We refer to these classifications as *domains*. Again, the approach of organizing and preparing software assets for reuse according to their domain is called *domain engineering*. Czarnecki and Eisenecker (2000, p. 33) define domain engineering as follows.

*"Domain engineering is the activity of collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets (i.e. reusable workproducts), as well as providing an adequate means for reusing these assets (i.e. retrieval, qualification, dissemination, adaptation, assembly, etc.) when building new systems."*

Domain engineering embodies three consecutive steps: *domain analysis*, *domain design*, and *domain implementation* (Czarnecki and Eisenecker, 2000). These steps are illustrated in Figure 2.3. In the domain analysis, a set of reusable, configurable requirements for the system in the domain

is defined. The purpose is to select and define the domain of focus and to collect relevant domain information. Sources of information are usually existing systems, experiments, publications, or domain experts. This step yields to a domain model capturing all relevant relationships among the classes, oftentimes represented through a UML class diagram. According to the mobile phone product line example indicated in Figure 2.1, the domain model states that a mobile phone needs to have exactly one type of screen (either basic, colour, or high resolution).

During domain design, a common platform for the system in the domain is established. We call such a platform an architecture for the family of systems in the domain (Pohl et al., 2005). Part of the architecture is a variability model such as the feature model presented in Figure 2.2 for the mobile phone example. The architecture along with the feature model captures commonality and variability and therefore enables a production plan. GPS for a mobile phone is optional whereas the ability for calls, a screen type, and a battery type (AccuCell) are mandatory.

Finally, during domain implementation, reusable software assets are implemented. These include domain specific languages, components, and code generators. When choosing preprocessor-based variability, features from the feature model are mapped to preprocessor directives. Hence, different selections of features (i.e., configurations) result in different source code and therefore in a different, yet similar, product.

In conclusion, domain engineering focuses on providing reusable solutions for *families* of systems contrary to conventional software engineering, which focuses only on a single system. An important part of the result is a variability model, like the feature diagram presented in Section 2.1.2.

During *application engineering*, reusable software assets developed in the domain engineering phase are received to derive applications based on configurations. First, a requirement analysis similar to traditional software engineering is performed. If necessary features are missing, they need to be implemented. Then, based on the requirements, features are selected, mapped to the corresponding software artifacts, and a software product is derived (Pohl et al., 2005; Apel et al., 2013). Nevertheless, in this thesis we are mainly concerned with feature models and domain engineering.

## 2.2. A Survey of Feature Modeling Languages

In the history of software product lines and feature modeling, many different languages have been proposed. We distinguish between two kinds of representations, graphical and textual languages. Graphical feature modeling languages (also called *feature diagrams*) based on FODA Kang et al. (1990) are by far the most common used in the literature. According to RQ1, we are interested in the role of complex constraints in different languages. This includes identifying commonalities and differences between different languages. Many of the languages have been surveyed elsewhere (Schobbens et al., 2007; Alturki and Khedri, 2010). However, they all only focus either on a specific subset (e.g., graphical or textual notations) or have a different concern than ours. Hence, we want to recall the essentials on these surveys and complete them with additional languages.

This survey is by no means complete, but it has the purpose of identifying most important characteristics of different feature modeling languages. In particular, we are interested in concepts that enriches the expressiveness of feature models. This includes supported decomposition types,

support for simple and complex constraints, whether the languages are built upon directed acyclic graphs or trees, and if abstract features are supported. It gives a sampled overview of different feature modeling dialects. In Section 3.2, we use this information to define an abstract syntax and a formal semantics to capture the most important characteristics of feature model languages.

### 2.2.1. Graphical Representations of Feature Models

In the following, we present nine different graphical feature modeling languages.

**FODA (Kang et al., 1990).** The feature diagram introduced as part of the *Feature Oriented Domain Analysis* (FODA) by Kang et al. (1990) was the first ever specified graphical feature modeling notation and as such the foundation for many other extensions. As depicted in Figure 2.4, the characteristics are as follows:

- It is a tree.
- The *root* feature (also termed *concept*) of the product line is at the top.
- Features can be *mandatory* (by default) or *optional* (visualized with a hollow circle).
- Features can be hierarchically decomposed into *and*-groups or *alternative*-groups (cf. Section 2.1.2).
- Features can have a non-hierarchically relationship with textual cross-tree constraints called *composition rules* (the same as simple constraints).
  - Features that are mutual exclusive, i.e., two features cannot coexist in the same configuration.
  - *Including* features, i.e., the presence of one feature requires the existence of another feature

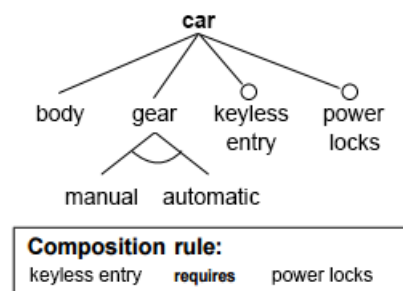


Figure 2.4.: Syntax of a FODA Feature Diagram (Adopted from Kang et al. (1990))

**FeatuRSEB (Griss et al., 1998).** Griss et al. (1998) have developed *FeatuRSEB* which is a fusion of FODA and the *Reuse-Driven Software Engineering Business* (RSEB) method. RSEB "[...] is a systematic, model-driven approach to large-scale software reuse" (Griss et al., 1998). Figure 2.5 shows an example feature diagram in FeatuRSEB. Differences and extensions to FODA are as follows.



- It is a directed acyclic graph.
- Hierarchical decomposition into *or* groups was added (cf. Section 2.1.2).
- Constraints are represented in graphical fashion.
  - Requires with a dashed arrow
  - Excludes with a dashed double-arrow

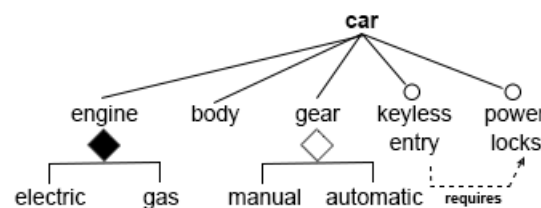


Figure 2.5.: Syntax of a FeaturSEB Feature Diagram

**FORM (Kang et al., 1998).** Kang et al. (1998) have proposed a couple of extensions under the name *Feature-Oriented Reuse Method* (FORM) to FODA.

- Feature diagrams can be directed acyclic graphs instead of trees.
- Features are now organized into a hierarchy of the following four layers. *capabilities layer* (functionality of the end user), *operating environment layer* (attributes of the environment; hardware, software), *domain technologies layer* (non-technical issues), *implementation technologies layer* (technologies not specific to a domain).

**Generative programming (Czarnecki and Eisenecker, 2000).** Czarnecki and Eisenecker (2000) extend the feature diagram of FODA and adapt it to generative programming, a programming paradigm aiming at the automated generation of source code.

- Hierarchical decomposition into *or* groups was added.
- Different decomposition types can be specified on the same level in the tree (e.g., a mandatory feature next to an *or* group).
- Group features can be labeled as mandatory (e.g., an *or* group with a mandatory feature).

**Gurp et al. (2001).** Gurp et al. (2001) only extend FeaturSEB (Griss et al., 1998) by *binding times* and *external features*, which are features referring to "technical possibilities offered by the target platform of the system" (Schobbens et al., 2007).

**Riebisch et al. (2002).** Riebisch et al. (2002) proposed the concept of *group cardinalities* in feature models. Group cardinalities are a UML-like version of multiplicities and generalize *or*, *and*, and *alternative* groups. Group cardinalities consist of two numbers. First, a lower bound indicating how many sub-features must be included. Second, an upper bound restricting the maximum number of sub-features that can be included.



- It is a directed acyclic graph.
- Group cardinalities instead of and, or, and alternative groups.
- Edges can be mandatory or optional.
- Simple constraints as in FeatuRSEB

Figure 2.6 depicts an example of the car model in the language proposed by Riebisch et al. (2002). Mandatory features are visualized with a black circle at the end of an edge. Optional features are visualized with a white circle. Moreover, group cardinalities dictate the group type. For instance, feature *gear* has group cardinality  $1..1$ , which is essentially the same as an alternative group.

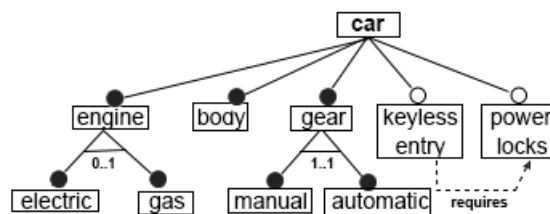


Figure 2.6.: Syntax of a Feature Diagram Proposed by Riebisch et al. (2002)

**PLUSS (Eriksson et al., 2005).** *Product Line Use case modeling for Systems and Software engineering* (PLUSS) by Eriksson et al. (2005) follows the same model driven approach as proposed by Griss et al. (1998) with FeatuRSEB. Additionally, it combines use case modeling with feature modeling in the same graphical representation. PLUSS has the following characteristics:

- PLUSS is a tree.
- There are some notational changes for decomposition. Every node has a circle which can possess a specific character. An 'S' in a circle refers to *single adaptors*, which is equivalent to be part of an *alternative-group*. An 'M' in a circle refers to *multiple adaptors*, which is equivalent to be part of an *or-group*. A white circle without a character indicates a optional feature and a black circle indicates a mandatory feature.
- PLUSS has graphical requires and excludes constraints.

**Benavides et al. (2005).** Benavides et al. (2005) provide an extension to the language proposed by Czarnecki and Eisenecker (2000). They introduced the concept of *attributed* features. Attributes in feature models are additional non-boolean information (e.g., int, string,...) directly attached onto features. Attributes can then be used for specific tools. For example, finding optimal products requires each feature to have a cost. Restricting variability by means of constraints on attributes is another common use case.

**FeatureIDE (Kästner et al., 2009).** FEATUREIDE is an Eclipse-based open-source framework supporting feature-oriented software development, a paradigm for the construction, customization and synthesis of software systems. FEATUREIDE offers a graphical feature model editor for constructing feature diagrams as well as a textual representation, which can be modified by hand. There are two main extensions to FODA.

- Arbitrary propositional formulas as cross-cutting constraints are allowed.
- FEATUREIDE explicitly distinguishes between *abstract* and *concrete* features.

Because FEATUREIDE is a framework following feature-oriented software development, it integrates a mapping from source code to features in a feature model. One distinctive characteristic of features in FEATUREIDE is that every feature can either be abstract or concrete, typically highlighted through a difference in color in the concrete syntax. We refer again to Figure 2.2 on Page 8, showing an example of a feature diagram modeled in FEATUREIDE.

### 2.2.2. Textual Representations of Feature Models

Over time, textual feature modeling notations have also been proposed, arguing that it is oftentimes difficult to analyze and interpret large visual feature diagrams. In the following, we present eight textual feature modeling languages.

**Van Deursen and Klint (2002).** The approach proposed by Van Deursen and Klint (2002) is one of the earliest attempts of defining a textual notation for feature models. Its representation is a list of *feature definitions*, where each definition consists of a unique feature name followed by ":" and a *feature expression*. The textual notation by Van Deursen and Klint (2002) has the following characteristics.

- It is a tree.
- Features can be decomposed into or, and, and alternative groups with syntactical function *more-of*, *all*, and *one-of*, respectively.
- An and group comprises mandatory and optional features (suffixed with a ?).
- Simple constraints can be specified.

Figure 2.7 shows an example of the car feature diagram depicted in Figure 2.5 on Page 12.

```

1 car: all(engine, body, gear, keyless_entry?, power_locks?)
2 engine: more-of(electric, gas)
3 gear: one-of(manual, automatic)
4 keyless_entry requires power_locks

```

Figure 2.7.: Syntax of a Feature Model Proposed by Van Deursen and Klint (2002)

**GUIDSL (Batory, 2005).** GUIDSL is part of the AHEAD tool suite and offers a grammar-like notation for specifying a feature model. It consists of the following elements:

- A feature  $s$  with mandatory children  $e_1 \dots e_n$  is represented as  $s:e_1 \dots e_n$ . If a child is optional, it is surrounded by [brackets].
- An *alternative*-group is represented as  $s:e_1|e_2|\dots|e_n$
- An *or*-group is a composition of two rules:  $s:t+$  and  $t:e_1|e_2|\dots|e_n$ .

- In addition, arbitrary propositional formulas are allowed between terminal symbols, represented through certain keywords (i.e., implies, or, and, not, brackets).

Figure 2.8 shows an example of the car feature model in GUIDSL.

```

1 car : engine body gear [keyless_entry] [power_locks];
2 engine : e+;
3 e : electric | gas;
4 gear : manual | automatic;
5 keyless_entry implies power_locks;

```

Figure 2.8.: Syntax of a GUIDSL Feature Model

**SXFM (Mendonça et al., 2009).** SXFM (*Simple XML Feature Models*) is a file format created for the software product line online tools (S.P.L.O.T.). SXFM is similar to the tree-grammar approach by Batory (2005). Moreover, SXFM uses group cardinalities instead of explicit or, and, or alternative groups (cf. language proposed by Riebisch et al. (2002)). Complex constraints are supported, but must be specified in conjunctive normal form. Each clause then represents a solitary cross-tree constraint. Figure 2.9 shows an example of the car feature model in SXFM.

```

1 <feature_model name="car">
2 <meta>
3 ...
4 </meta>
5 <feature_tree>
6 :r car(_r)
7   :m engine(_r_1)
8     :g (_r_1_6) [1,*]
9       : electric(_r_1_6_7)
10        : gas(_r_1_6_8)
11   :m body(_r_2)
12   :m gear(_r_3)
13     :g (_r_3_9) [1,1]
14       : manual(_r_3_9_10)
15       : automatic(_r_3_9_11)
16   :o keyless_entry(_r_4)
17   :o power_locks(_r_5)
18 </feature_tree>
19 <constraints>constraint_1:~_r_4 or _r_5</constraints>
20 </feature_model>

```

Figure 2.9.: Syntax of a Feature Model in SXFM

**FAMA (Benavides et al., 2007)** FAMA (FeAture Model Analyzer) is a framework for the automated analysis of feature models, including recognizing anomalies as dead features (i.e., features that not part of any product) and false-optionals (i.e., features that must be selected if their parents are selected). FAMA offers essentially two input formats. First, a XML format for feature models with cardinality-based relationships, but only simple constraints. Second, a plain text format for either the same feature models as for XML, or attributed feature models with the possibility to use complex constraints.

However, when we used the attributed format in the interactive shell offered by FAMA<sup>2</sup>, crucial analyses (e.g. counting the numbers of valid products) were not available anymore. Hence, Figure 2.10 shows an example of the car feature model in the basic plain text format.

```

1 %Relationships
2 car: engine body gear [keyless_entry] [power_locks];
3 engine: [1,2]{electric gas};
4 gear: [1,1]{manual automatic};
5 %Constraints
6 keyless_entry REQUIRES power_locks;
```

Figure 2.10.: Syntax of a FAMA Feature Model

**TVL (Boucher et al., 2010).** TVL is a text-based feature modeling language with a C-like syntax that claims to be "[...] both light and comprehensive, meaning that it covers most constructs of existing languages [...]" (Boucher et al., 2010). One of the goals of TVL is to be *scalable* by offering mechanisms for modularity while also being *readable* by humans. TVL enhances basic feature models with cardinality-based decomposition and feature attributes. Additionally, feature models in TVL can be directed acyclic graphs. Typical cross-tree constraints in TVL are boolean expressions, meaning that arbitrary propositional formulas as constraints can be defined. Figure 2.11 shows an example of the car feature model in TVL.

```

1 root car {
2     group allOf {
3         engine group [1..2] {
4             electric,
5             gas
6         },
7         body,
8         gear group [1..1] {
9             manual,
10            automatic
11        },
12        opt keyless_entry,
13        opt power_locks
14    }
15    keyless_entry requires power_locks;
16 }
```

Figure 2.11.: Syntax of a Feature Model in TVL

**Velvet (Rosenmüller et al., 2011).** VELVET aims at support for multidimensional variability modeling. Feature models in VELVET can be combined to model dependent software product lines. The concrete syntax is similar to object-oriented classes, using specialized keywords to indicate features, feature groups and constraints. Features are hierarchically decomposed and by default optional. Mandatory features are denoted with the *mandatory* keyword. Similar to FEATUREIDE, features can be *abstract*. Feature groups are either alternative relationships or or relationships, explicitly indicated through the keywords *oneOf* and *someOf*. Additionally, VELVET offers the possibility for defining attributes. VELVET has full support for complex constraints. Furthermore,

<sup>2</sup>Repository for FAMA: <https://code.google.com/archive/p/famats/downloads>

there exist also constraints for attributes in particular. Figure 2.12 shows an example of the car feature model in VELVET.

```

1 concept car {
2     mandatory feature engine {
3         someOf { feature electric; feature gas;}
4     }
5     mandatory feature body;
6     mandatory feature gear {
7         oneOf { feature manual; feature automatic;}
8     }
9     feature keyless_entry;
10    feature power_locks;
11
12    constraint keyless_entry -> power_locks;
13 }

```

Figure 2.12.: Syntax of a VELVET Feature Model

**Familiar (Acher et al., 2011).** FAMILIAR is a scripting language that focuses on the management of feature models. Different operators are offered including the combination, manipulation, and analysis of feature models. FAMILIAR is Eclipse-based and, among other things, consists of a textual editor, an interpreter, reasoner, an integration of the textual language to the FEATUREIDE graphical editor, and the integration for some model formats. FAMILIAR allows the basic modeling of alternative and or groups as well as mandatory and optional features. Additionally, propositional formulas can be specified, thus, allowing complex constraints. Figure 2.13 shows an example of the car feature model in FAMILIAR.

```

1 fm1 = FM(car : engine body gear [keyless_entry] [power_locks];
2         engine: (electric|gas)+; gear: (automatic|manual);)
3 c1 = { keyless_entry implies power_locks;}

```

Figure 2.13.: Syntax of a FAMILIAR Feature Model

**Clafer (Bak et al., 2013).** CLAfer (CLAss, FEature, Reference) is a framework offering meta-modeling and first-class support for feature models. Feature models in CLAfer are trees. The language offers basic feature groups (alternative group and or group). Moreover, group cardinalities for each feature can be specified. Cross-tree constraints in CLAfer are either arbitrary propositional formulas or *if-then-else* expressions (higher level concept). Other notable constructions include the specification of attributes and abstract features. Figure 2.14 shows an example of the car feature model in CLAfer.

```
1  abstract car
2      or engine
3          electric
4          gas
5      body
6      xor gear
7          manual
8          automatic
9      keyless_entry ?
10     power_locks ?
11     [keyless_entry => power_locks]
```

Figure 2.14.: Syntax of a CLAFER Feature Model

### 2.2.3. Comparison of Feature Model Representations

Table 2.1 summarizes important characteristics of all surveyed languages. Again, we are particularly interested in concepts that enhance the expressiveness of a language. For instance, every language that supports directed acyclic graphs is already capable of expressing *every* possible product line (Schobbens et al., 2007). However, feature trees are more common in today’s product-line engineering. Moreover, complex constraints are also sufficient, as we will prove formally in Chapter 4. Unfortunately, basic feature models like FODA or GP lack expressiveness (Schobbens et al., 2007). We also provided a column for *attributes*. Attributes do not enhance expressiveness, as they deal with non-functional properties. Nevertheless, attributes are an important part of *extended feature models* and offer a different type of variability, which is why we included them. In the next section, we investigate to what extent typical feature modeling applications deal with complex constraints.

Feature Modeling Language	Representation	Graph type <sup>1</sup>	Decomposition types <sup>2</sup>	Constraint type	Attributes	Abstract features	Remarks
FODA (Kang et al., 1990)	Graphical	Tree	{ $\wedge$ , $\oplus$ , opt}	Simple	-	-	-
FEATURSEB (Griss et al., 1998)	Graphical	DAG	{ $\wedge$ , $\vee$ , $\oplus$ , opt}	Simple	-	-	First with or groups
FORM (Kang et al., 1998)	Graphical	DAG	{ $\wedge$ , $\oplus$ , opt}	Simple	-	-	First with DAG
GP (Czarnecki and Eisenecker, 2000)	Graphical	Tree	{ $\wedge$ , $\vee$ , $\oplus$ , opt}	Simple	-	-	Mandatory/optional in groups, multiple groups under one feature
Gurp et al. (2001)	Graphical	DAG	{ $\wedge$ , $\vee$ , $\oplus$ , opt}	Simple	-	-	-
Riebisch et al. (2002)	Graphical	DAG	{card, opt}	Simple	-	-	First with group cardinality
PLUSS (Eriksson et al., 2005)	Graphical	Tree	{ $\wedge$ , $\vee$ , $\oplus$ , opt}	Simple	-	-	Notation of multiple adapters
Benavides et al. (2005)	Graphical	Tree	{ $\wedge$ , $\vee$ , $\oplus$ , opt}	Simple	yes	-	Attributes
FEATUREIDE (Kästner et al., 2009)	Both	Tree	{ $\wedge$ , $\vee$ , $\oplus$ , opt}	Complex	-	yes	Explicit abstract features
Van Deursen and Klint (2002)	Textual	Tree	{ $\wedge$ , $\vee$ , $\oplus$ , opt}	Simple	-	-	-
GUIDSL (Batory, 2005)	Textual	Tree	{ $\wedge$ , $\vee$ , $\oplus$ , opt}	Complex	-	-	First with complex constraints
FAMA (Benavides et al., 2007)	Textual	Tree	{card, opt}	Simple	yes	yes	-
VELVET (Rosenmüller et al., 2011)	Textual	Tree	{ $\wedge$ , $\vee$ , $\oplus$ , opt}	Complex	yes	yes	Relational expressions
SXFM (Mendonça et al., 2009)	Textual	Tree	{card, opt}	Complex	-	-	Constraints are in conjunctive normal form
TVL (Boucher et al., 2010)	Textual	Tree	{ $\wedge$ , $\vee$ , $\oplus$ , card, opt}	Complex	-	-	Relational expressions
FAMILIAR (Acher et al., 2011)	Both	Tree	{ $\wedge$ , $\vee$ , $\oplus$ , opt}	Complex	-	-	-
CLAFER (Bak et al., 2013)	Textual	Tree	{ $\wedge$ , $\vee$ , $\oplus$ , card, opt}	Complex	yes	yes	Relational expressions

<sup>1</sup> DAG: Directed acyclic graph<sup>2</sup>  $\wedge$ : and group,  $\vee$ : or group,  $\oplus$ : alternative group, *card*: group cardinality, *opt*: optionality

Table 2.1.: Comparison of Feature Modeling Languages

## 2.3. Applications of Feature Models

In the previous section, we presented a survey of some important feature modeling languages we found in the literature, and reviewed whether they support complex constraints. In this section, we address the second part of *RQ1: which approaches and tools do support/do not support complex constraints?* To answer this question, we chose five feature model application areas. In particular, we discuss *automated analysis of feature models*, *synthesis of feature models*, *generation of feature models as test data*, *product-line testing and analysis*, and *optimal feature selection*.

**Automated Analysis of Feature Models.** The automated analysis deals with computer-aided extraction of information from feature models (Batory et al., 2006). Automation is needed, as the manual analysis is not only highly error-prone but also infeasible for large-scale feature models. Benavides et al. (2010) have contributed a literature review on the automated analysis of feature models, highlighting the increase in quality and quantity of analysis operations over the last decades. In total, they identified 30 analysis operations in the literature to be performed on feature models. For example, some of the more prominent operations are checking for a void feature model (i.e., a model that has no valid products), counting the number of generated products, finding dead features (i.e., features that are not part of any configuration), or identifying false-optionals (i.e., optional features that behave like mandatory features). Since the beginning, a whole community has been built around *the automated analysis of feature models*, which led to an increase in the number of proliferated methods and analysis tools (Benavides et al., 2010).

An important step towards more effective techniques in the automated analysis is the translation from a feature model to propositional logic (Batory, 2005). This has opened the whole squad of logic-based mathematics to reason about feature models, including off-the-shelf satisfiability solvers. Many of the analysis operations can be therefore formulated as SAT-problems. For example, a satisfiability solver can check whether a legal assignment of a propositional formula, translated from a feature model, exists. If not, the feature model is a void feature model. Obviously, complex constraints are propositional formulas, hence analysis tools relying on SAT-solving techniques can easily integrate complex constraints. Analysis tools are very important applications in feature modeling, especially in practice, as they also help in *interactive configurations*, where a user derives a product by consecutively selecting features. The tool guides the user by enabling or disabling features to select based on a SAT-analysis.

Some prominent analysis tools are S.P.L.O.T. (Software Product Lines Online Tools) (Mendonça et al., 2009), FAMA (FeAture Model Analyser) (Benavides et al., 2007), and FEATUREIDE (Kästner et al., 2009). S.P.L.O.T. is a web-based reasoning and configuration system for software product lines. It offers services for automated statistics computation and basic analysis of feature models, i.e. checking for a void model and detecting the presence of dead and common features (Mendonça et al., 2009). As mentioned in Section 2.2, S.P.L.O.T. uses the SXFM format and as such can make use of complex constraints. FAMA is another tool to analyze feature models. FAMA can be either used with a stand-alone shell (front-end), as a web-service, or integrated as a JAVA library, and offers similar services as S.P.L.O.T. i.e. calculating the number of products or checking for consistency. FAMA supports both, feature models with only simple constraints and extended feature models (cf. Section 2.2). Extended feature models support complex constraints (even with attributes). However, as we already mentioned in the last section, some basic analysis



operations (e.g., counting the number of valid products) seem not to work properly with the extended version. Moreover, we have not found a simple example employing complex constraints.

A different problem in this discipline is addressed by BETTY (Segura et al., 2012). More and more in-house analysis tools are created by researchers. Rather than simply developing basic research prototypes, the goal should be to develop bug-free and efficient high quality analysis tools. BETTY offers a platform to test and compare these analysis tools. However, BETTY can only deal with basic (FAMA) feature models at this moment, which renders the testing and comparing of analysis tools with a richer feature modeling language insignificant.

**Synthesis of Feature Models.** Given the rigorous task of manually reverse engineering a feature model from a configuration, a propositional formula, or even an informal product description, the product-line community has shown significant interest in automating this process.

Acher et al. (2012) proposed a semi-automated method to construct a feature model hierarchy and cross-tree constraints based on tabular product descriptions. They explicitly use requires and excludes constraints in their formulation, but conclude themselves that those two constraints are not expressively sufficient. Their evaluation shows an over-approximation of configurations for various real-world cases. Bécan et al. (2015) build up on the work of Acher et al. (2012) and allow arbitrary propositional formulas. Al-Msie 'deen et al. (2014) propose a method to mine a feature model from a set of software configurations using formal concept analysis, a mathematical principle based on complete lattice (Ganter and Wille, 2012). However, they only give algorithms for extracting feature groups, requires and excludes constraints. Moreover, in their experiments, they derive a set of configurations from basic feature models, which they use for their mining approach. Hence, they do not deal with complex constraints at all. The method proposed by She et al. (2011) requires a set of feature names, feature descriptions, and feature dependencies as input, and heuristically identifies parental relationships and requires and excludes constraints, resulting in a reverse engineered feature model. Analogous to Acher et al. (2012), they tolerate the over-approximation of configurations in their results. Nevertheless, both methods identify dependencies that cannot be entailed in basic feature models as a propositional formula, which essentially is a complex constraint. A later extension to this work is based on input of feature models in either conjunctive normal form or disjunctive normal form (She et al., 2014). Furthermore, She et al. (2014) encodes requires and excludes constraints now in their formulation of feature diagrams and explicitly carry an extra propositional formula for more complex feature dependencies. Haslinger et al. (2013) also propose an algorithm to synthesize feature models from valid configurations. Their approach is based on finding patterns among valid configurations to estimate parental relationships. They mention the existence of more complex constraints, but discuss only requires and excludes constraints. (Lopez-Herrejon et al., 2015) evaluated three search based methods (evolutionary algorithms, hill climbing, and random search) for reverse engineering feature models in the domain of genetic algorithms. They use a special array for cross-tree constraints, on which mutations and crossovers are applied. However, they only consider requires and excludes constraints. A similar approach is proposed by Linsbauer et al. (2014), using evolutionary algorithms, too. Again, only requires and excludes constraint are considered. This may correlate with the difficulty of encoding arbitrary cross-tree constraints in such algorithms.

**Generation of Feature Models as Test Data.** A related field is the automated generation of computationally hard feature models. As mentioned before, analysis tools in feature modeling are important applications and should aspire efficiency. The problem of generating hard test data (i.e., computationally hard to analyze feature models) to find vulnerable weak points in such tools has been largely studied.

Obviously, as feature models become bigger, solvers take more time for analysis. However, one conclusion is that random values are not enough for discovering efficiency reducing flaws in these systems (McMinn, 2004). However, to empirically evaluate one's tools, authors often rely on more realistic feature models with hundreds of features. Moreover, only a small portion of meaningful feature models is typically publicly available. Hence, authors oftentimes must generate their test data themselves.

Thüm et al. (2009) randomly generate feature models based on probabilities for certain groups to show the scalability of their classification approach for feature model edits. They elucidate their algorithm, used probabilities, and results. Furthermore, complex constraints are generated, too, and according to them, the resulting feature models are close to *realistic* feature models. Guo et al. (2011) adopt the approach and probabilities used by Thüm et al. (2009) but discards the generation of complex constraints, as for their work only requires and excludes are needed. Segura et al. (2014) addresses the absence for real approaches to generate sufficient test data in general and propose to model the finding of computationally hard feature models as an optimization problem, solved by using a novel evolutionary algorithm (ETHOM). This algorithm is publicly available and part of the BETTY framework (Segura et al., 2012). The algorithm allows the user to specify the number of features and percentage of cross-tree constraints. However, ETHOM can only generate basic feature models with simple constraints. Segura et al. (2014) concluded in their publication that a more flexible algorithm dealing with complex constraints and cardinality is desirable and will be part of future work.

**Product-Line Testing and Analysis.** As product lines are increasingly applied to safety-critical software projects (e.g., automotive or aviation) and their software assets may be reused several times, verification and analysis become important means to ensure reliability and correctness of all generated products. However, testing an entire product line is a difficult task, as even feature models with a few features can comprise an exponential number of products. There are mainly three strategies of product-line testing.

*Product-based strategy* follows a brute-force strategy and tries to verify the correctness of a product line by generating each product individually (Thüm et al., 2012). However, it is obvious, that this strategy does not scale well for product lines with many features.

Alternatively, a *sample-based strategy* can be applied. This strategy aims on a subset of possible products, oftentimes realized through combinatorial interaction testing, to reduce the problem space, and to identify defects faster (Al-Hajjaji et al., 2014). Interactions can be tested among  $t = 1, 2, 3, \dots$  features. The general case where  $t$  can be chosen freely is therefore called *t-wise testing*, whereas the typical case between exactly two features is called *pairwise testing* (Al-Hajjaji et al., 2014).

The third strategy is called *family-based strategy* (Thüm et al., 2012), an approach to apply verification on implementation artifacts of a whole product line instead of each product separately.

Perrouin et al. (2010) propose a sample-based algorithm. They built a metamodel for feature

diagrams based on the formal semantics defined by Schobbens et al. (2007). They use the same textual constraint language, i.e., allowing only requires and excludes cross-tree constraints. However, their approach is based on a model transformation from a feature diagram to Alloy.<sup>3</sup> This may allow an easy integration of complex constraints. Shi et al. (2012) propose a sample-based approach that not only uses a feature model, but incorporates underlying code, too. Both are used to build a feature dependency graph representing (and limiting) the set of possible interaction among features. However, only requires and excludes constraints are discussed. Ensan et al. (2012) uses genetic algorithms to minimize the number of products that needs to be tested. They refer to requires and excludes constraints as integrity constraints and do not discuss any other forms of constraints besides those two. Al-Hajjaji et al. (2014) propose a similarity-based prioritization before products are generated and tested, aiming at faster increasing interaction coverage. Moreover, they extended FEATUREIDE with product-line testing and similarity-based prioritization, and hence, use arbitrary cross-tree constraints.

**Optimal Feature Selection.** Earlier industrial case studies have shown that product derivation can be a time consuming and costly process (Deelstra et al., 2004, 2005). Practitioners face a couple of challenges when trying to derive an optimized feature selection based on both functional and arbitrary non-functional requirements. For example, consider a database product line where a product might be used in a mobile application. The application has certain requirements towards a database’s functionality. However, today’s mobile phone may have a limited amount of memory space. Therefore, the optimization goal could be to select a configuration that does not exceed a certain memory space threshold while at the same time maximizing functionality. Many *exact methods* for calculating the real optimum haven been proposed. However, optimal feature selection with such non-functional requirements has been proven to be NP-hard (White et al., 2009). Hence, many heuristics have been proposed, too.

Benavides et al. (2005) choose an exact method and modeled the problem of optimal product configuration as a constraint satisfaction problem. However, they take neither complex nor simple constraints into account. White et al. (2009) propose an approximation algorithm which transforms the problem into a multidimensional multiple choice knapsack problem and solves it with the modified heuristic method. They only discuss requires and excludes constraints. A later approach uses a formulation as constraint satisfaction problem, too White et al. (2014). Again, complex constraints are left out. Guo et al. (2011) proposed GAFES, an evolutionary algorithm to heuristically solve this optimization problem. GAFES does not support complex constraints. (Machado et al., 2014) proposes SPLConfig, a tool built upon FEATUREIDE for optimal configuration derivation. As FEATUREIDE supports complex constraints, SPLConfig does, too. (Zanardini et al., 2016) presents several strategies for optimal feature selection, mostly regarding resource-usage for functional properties. They report and prototypical implementation of some strategies and conduct an industrial case study. They also include complex constraints in their implementation.

---

<sup>3</sup>The Alloy Analyzer is a software tool for automated analyses of specifications written in the Alloy specification language. See <http://alloy.mit.edu/alloy/>.

## Support for Complex Constraints

In the following, we concisely conclude to what extent complex constraints are supported in our reviewed publications. We chose a small representative set of 26 publications in total, classified in five applications (cf. Section 2.3). Each publication represents either an algorithmic method or an introduced tool. We also give a brief description and important characteristics for each publication. Furthermore, we state for each publication if complex constraints are supported in one of three ways. A *Yes* means integrated support. Analogously, a *No* means no integrated support. A *(Yes)* in brackets indicates that either support is discussed and integration is intended, or can easily be integrated but is not discussed in detail (e.g., if the method works with a feature model represented as a propositional formula). Table 2.2 illustrates our results.

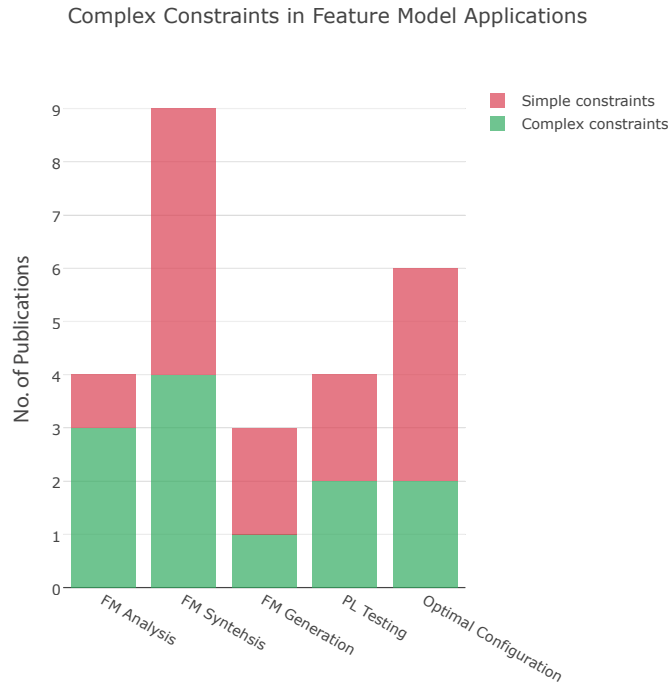


Figure 2.15.: Frequency of Complex Constraints in Feature Model Applications

In Figure 2.15, we present the number of reviewed publications for each feature model application individually, and highlight how many of them support complex constraints. It can be seen that in almost all fields, the ratio between simple and complex constraints is balanced. Obviously, the number of reviewed publications is too few to make substantial assumptions about the acceptance of complex constraints in these fields.

Furthermore, we reviewed many older publications which we assumed to have a tendency against complex constraints. However, in Figure 2.16, it can be seen that the number of publications not supporting complex constraints does not decrease over time. In total, less than roughly 50% of all reviewed publications discuss the integration of complex constraints.

Automated analysis of feature models				
Reference	Contribution	Characteristics	Complex constraints	Remarks
S.P.L.O.T. (Mendonça et al., 2009)	Tool	Analysis; Feature model generation	Yes	Online access <sup>1</sup>
FAMA (Benavides et al., 2007)	Tool	Analysis	Yes	JAVA library, stand-alone, or web service
FEATUREIDE (Kästner et al., 2009)	Tool	Combination of software product line methods	Yes	Eclipse-based
BETTY (Segura et al., 2012)	Tool	Benchmarking; Feature model generation	No	Online access <sup>2</sup>
Feature Model Synthesis				
Reference	Contribution	Characteristics	Complex constraints	Remarks
She et al. (2011)	Method	From descriptions	No	Evaluated on large real-world models
Ryssel et al. (2011)	Method	From configurations	Yes	Formal concept analysis
She et al. (2014)	Method	From propositional formulas	(Yes)	-
Acher et al. (2012)	Method	Semi-automated; from product descriptions	(Yes)	-
Al-Msie 'deen et al. (2014)	Method	From configurations	No	Formal concept analysis
Bécan et al. (2015)	Method	Semi-automated; from product descriptions	Yes	-
Haslinger et al. (2013)	Method	From configurations	No	-
Lopez-Herrejon et al. (2015)	Method	From configurations	No	Genetic algorithms
Linsbauer et al. (2014)	Method	From configurations	No	Genetic algorithms
Feature Model Generation as Test Data				
Reference	Contribution	Characteristics	Complex constraints	Remarks
Thüm et al. (2009)	Method	Close to realistic feature models	Yes	Probabilistic parameters for groups and constraints
Guo et al. (2011)	Method	Close to realistic feature models	No	Based on method proposed by Thüm et al. (2009)
ETHOM (Segura et al., 2014)	Method	Generating hard feature models	No	Integrated in BETTY
Software product line testing				
Reference	Contribution	Characteristics	Complex constraints	Remarks
Perrouin et al. (2010)	Method	Sample-based strategy	(Yes)	t-wise coverage
Shi et al. (2012)	Method	Sample-based strategy	No	Compositional symbolic execution
Ensan et al. (2012)	Method	Sample-based strategy	No	Evolutionary algorithm for test case generation
Al-Hajjaji et al. (2014)	Method	Prior to sample-based strategies	Yes	Ordering tests through similarity-based prioritization
Optimal feature selection				
Reference	Contribution	Characteristics	Complex constraints	Remarks
Benavides et al. (2005)	Method	Automatic	No	Constraint satisfaction problem
White et al. (2009)	Method	Automatic; Polynomial-time	No	Multidimensional multi-choice knapsack problem
GAFES (Guo et al., 2011)	Method	Automatic; Fast selection time	No	Evolutionary algorithm
White et al. (2014)	Method	Automatic; Multi-step	No	Constraint satisfaction problem
Machado et al. (2014)	Tool	Automatic; Built upon FEATUREIDE	Yes	
Zanardini et al. (2016)	Method	Automatic; Re-source-usage-aware configuration	Yes	$\mu$ TVL; SACO analyzer

<sup>1</sup> <http://www.splot-research.org/><sup>2</sup> <http://www.isa.us.es/betty/betty-online>

Table 2.2.: Summary of publications we reviewed for each feature modeling application including characteristics and information about support for complex constraints.

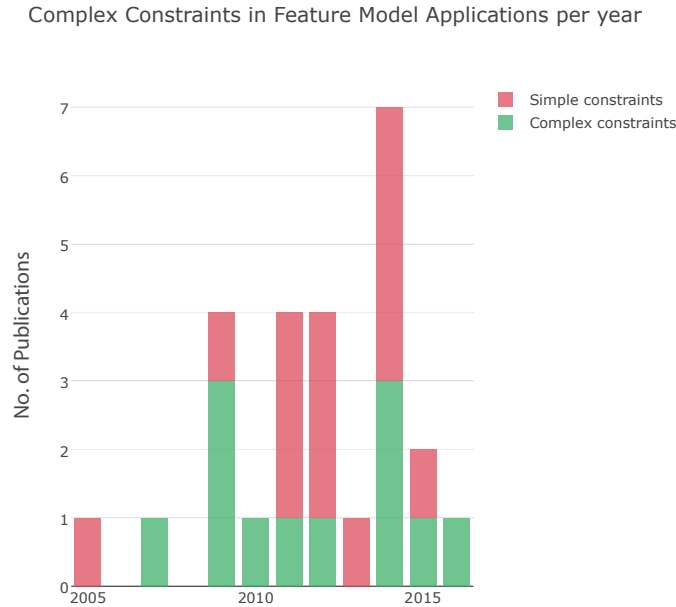


Figure 2.16.: Frequency of Complex Constraints Over Time

## 2.4. Summary

To capture commonality and variability of a product line, numerous feature modeling languages exist. While most of them originated from the feature diagram proposed by Kang et al. (1990), a couple of extensions have been submitted by the product-line community over the last decades. One of these extensions is the use of arbitrary propositional formulas for cross-tree constraints. Beforehand, only requires and excludes constraints could be used. Complex constraints offer practitioners a great tool to ease the arduous modeling process. However, they can also deteriorate the comprehension of feature interactions provoked by the potentially complex nature of propositional formulas. For this reason, questioning the acceptance of complex constraints in feature modeling was the goal of this chapter.

Overall, we surveyed 18 different graphical and textual feature model languages, seven of whom support complex constraints. We also briefly examined tool support and published methods in five different feature model applications and examined given support for complex constraints. For example, the analysis of feature models is mostly formulated as a satisfiability problem. A feature model is then translated into a propositional formula and can be solved with one of many SAT-solvers or binary decision diagrams. Therefore, complex constraints can easily be integrated, and tools like S.P.L.O.T. and FAMA already support them. In other areas, for instance *optimal feature selection* or *feature model synthesis*, we found very few attempts in literature so far that explicitly focus on the integration of complex constraints.

# 3 Formal Foundations of Feature Models

In this chapter, we introduce formal foundations of feature models as a means to mathematically reason about capabilities and limitations of different feature modeling languages. In the latter half of *RQ1*, we asked how valuable complex constraints are. We measure the value of a feature modeling *concept* (e.g., supported decomposition types, attributes, or constraint types) in terms of the *expressiveness* it adds to a language. The *expressive power* of a feature modeling language is a measurement for the number of product lines it can express. For example, a language that only knows alternative groups is less expressive than a language that additionally knows or groups. The underlying motivation for this chapter is to analyze the problem whether feature models with simple constraints and feature models with complex constraints are equally expressive.

First, based on our observations on applications of feature models (cf. Section 2.3) and in regard to *RQ1*, we briefly motivate the necessity of formal foundations in Section 3.1. Next, to employ theoretical considerations on feature models, we create a general formal semantics that covers the most important concepts of our surveyed languages (cf. Section 2.2) in Section 3.2. Afterwards, in Section 3.3, we quantitatively analyze the *expressive power* of basic feature models. Finally, we give a summary.

## 3.1. Motivation for a Formal System

In Section 2.2, we already emphasized the syntactical dissimilarities in a number of feature modeling languages. This variety in dialects makes it difficult to discuss definitions and results on a common basis if feature models are only presented informally. For example, it should be defined whether mandatory features in a feature model are only part of a product if their parents are part of it, or if they are part of all products. Without such clarification obscurity and ambiguity rise. Moreover, many authors do not distinguish between configurations (i.e., a valid feature selection including concrete and abstract features) and products (i.e., a valid feature selection including only concrete features).

In this context, Sun et al. (2005) propose the concept of *semantic equivalence*. Two feature models may therefore be *semantically equivalent*, even though they syntactically differ. Moreover, the semantic domain of feature models are product lines (Schobbens et al., 2007). Without a formal foundation of feature models, reasoning about equivalence of feature models (i.e., determining whether they represent the same product line) when changing one's syntax becomes a daunting and imprecise task. On the other hand, this raises an interesting question. Is there a feature model for every product line?



The answer to this question depends on the feature modeling language that is used. An important measure of feature model languages is the *expressive power* (Schobbens et al., 2007). Informally, the expressive power of a formal language is a measurement to what extent a language can represent the elements of its predefined semantics. For feature models, *expressive power* correlates with the question asked above. If a language can represent more product lines than another, its expressive power is simply greater. A follow-up question highlights the essence of this thesis and refines *RQ1*. Do complex constraints add expressive value to feature modeling languages, and if so, to what extent?

In the next two sections, we first formulate an unambiguous and rigorous formal semantics for feature models. Afterwards, we analyze the expressive power of basic feature models and feature models with complex constraints to eventually transpire the expressive gap between them. Again, formal foundations are necessary to exploit the power of mathematics and to avoid ambiguity.

## 3.2. A Formal Semantics for Feature Modeling Languages

Schobbens et al. (2007) propose a general formal semantics, namely *free feature diagram* (FFD), to capture numerous feature modeling notations with a single abstract syntax. In this thesis, we use the concept of FFD as a basis. However, we need to extend and modify FFD to suffice the needs of this thesis. For instance, FFD does not support complex constraints but only requires and excludes constraints.

The construction of FFD is based on the guidelines of Harel and Rumpe (2000), according to whom each modeling language  $\mathcal{L}$  is mathematically and unambiguously defined by three parts: a syntactic notation  $\mathcal{L}_L$  (syntax), a semantic domain  $\mathcal{S}_L$  (semantics) and a semantic function  $\llbracket \cdot \rrbracket_L : \mathcal{L}_L \rightarrow \mathcal{S}_L$ .

### 3.2.1. Defining an Abstract Syntax

The syntactic notation (also sometimes called syntactic domain) encapsulates the abstract syntax of a feature model. It determines what can be written using a feature modeling language. In Section 2.2, we already described common characteristics of feature models. In the following, we first want to informally agree on the characteristics that will be part of our abstract syntax. Then, we give a more formal definition.

A feature model in our case is a tree with a single root feature and each feature is decomposed into one or more features, except for terminal features. Features can also be labeled as optional, and we distinguish between abstract features (i.e., no influence on the final product) and concrete features (i.e., influence on the final product). This allows us to find out whether two or more product lines are identical in their products, meaning that they result in the exact same programs based on their code artifacts. Basic compound features, only used for decomposition in most feature modeling languages, are the same as abstract features. In contrast to FFD, we do not specify the decomposition type by well-defined operators (e.g.,  $and_k$ ,  $xor_k$ ,  $or_k$ , ...) but by group cardinality. Even if we do not allow cardinalities in our concrete syntax, all decomposition types



(i.e., and, alternative, or) can be represented by a cardinality in the abstract syntax as shown in Table 3.1. The reason for this approach is that all results in this thesis can be applied to feature modeling languages explicitly using group cardinalities in their concrete syntax. Finally, complex constraints can be specified.

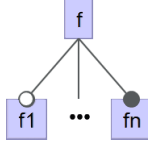
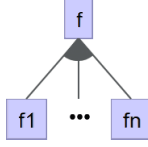
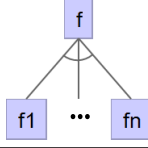
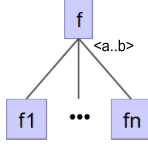
Cardinality	Operator	Concrete syntax
$\langle n..n \rangle$	$and_n$	
$\langle 1..n \rangle$	$or_n$	
$\langle 1..1 \rangle$	$alternative$	
$\langle a..b \rangle$	$card_{a,b}$	

Table 3.1.: Translation Between Group Cardinality and Concrete Syntax in a Feature Diagram

According to the name *Free Feature Diagram* defined by Schobbens et al. (2007), we denote our formal semantics of feature models as *Free Feature Model* ( $\mathcal{FFM}$ ). As mentioned earlier, the role of  $\mathcal{FFM}$  is to capture all necessary characteristics of a feature model (i.e., decomposition type, concrete/abstract features, parent-child-relations, optionality, and even cardinality) in an abstract syntax and provide a *semantic function* that maps each feature model of  $\mathcal{FFM}$  to its product line. Constructs such as *attributes* or *feature cardinality* are ignored, as they do not overlap with our contributions in this thesis.

Regarding the variety of different feature model languages (cf. Section 2.2),  $\mathcal{FFM}$  enables the comparison of two or more feature models regardless of their concrete syntax. Informally, two features models presented in different notations are therefore *equivalent* if their representation in  $\mathcal{FFM}$  represents the exact same product line. We denote the language of all feature models in  $\mathcal{FFM}$  as  $\mathcal{L}_{\mathcal{FFM}}$ . The definition of a feature model of  $\mathcal{L}_{\mathcal{FFM}}$  is as follows.

**Definition 3.1.** A feature model in  $\mathcal{L}_{\mathcal{FFM}}$  is a 7-tuple  $(N, P, r, \omega, \lambda, DE, \Phi)$  such that

- $N$  is the set of features.
- $P \subseteq N$  is the set of concrete features. Therefore,  $N \setminus P$  is the set of abstract features.  $P$  can be empty.
- $r \in N$  is the root feature.
- $\omega : N \rightarrow \{0, 1\}$  declares a feature as either optional (0) or mandatory (1).

- $\lambda : N \rightarrow \mathbb{N} \times \mathbb{N}$  represents the relationship of a parent feature and its sub-features in terms of cardinality  $\langle a..b \rangle$ , where  $a$  is the lower bound of sub-features required and  $b$  the upper bound of sub-features allowed.
- $DE \subseteq N \times N$  is the set of decomposition edges that represent parental relationships between features.  $(f, g) \in DE$  ( $f$  is parent of  $g$ ) is sometimes written as  $f \rightarrow g$ . We will denote by  $sub_f = \{f' \mid (f, f') \in DE\}$  the sub-features of  $f$ .
- $\Phi \subset \{\phi \mid \phi \in \mathbb{B}(N)\}$  is the set of cross-tree constraints (i.e., requires constraints, excludes constraints, or arbitrary propositional formulas).

Every feature model in  $\mathcal{L}_{\mathcal{FFM}}$  must also satisfy the following five well-formedness constraints.

- c1** Feature  $r$  has no parent: there exists no  $f \in N$  such that  $f \rightarrow r$ .
- c2**  $DE$  is acyclic: there exist no  $f_1, \dots, f_n \in N$  such that  $f_1 \rightarrow f_2, f_2 \rightarrow f_3, \dots, f_n \rightarrow f_1$ .
- c3** Terminal features are  $\langle 0..0 \rangle$ -decomposed.
- c4** Except for  $r$ , each feature has a single parent: for all  $g \in N \setminus \{r\}$ , there exists exactly one  $f \in N$  such that  $f \rightarrow g$ .
- c5** If a feature  $f$  is mandatory, it must be part of an and decomposition: if  $g \rightarrow f$  with  $\omega(f) = 1$ , then  $\lambda(g) = \langle n..n \rangle$  with  $n = |sub_g|$ .  $r$  is always mandatory.

As we use group cardinalities as a means for general decomposition, we may refer to *custom group cardinality* when neither and, or, nor alternative decompositions are involved. Implicitly, every feature that is not part of a  $\langle n..n \rangle$  decomposition, with  $n$  being the total number of features in such a group, is therefore optional by default (cf. Definition 3.1 (c5)). This is a justifiable simplification in our abstract syntax, as we restrict the concept of optionality of solitary features to only and decompositions. This obviously also means a restriction to our abstract syntax, as mandatory features mixed with custom cardinalities are conceivable. However, to the best of our knowledge, no such structures are used in the literature, and hence we do not take them into consideration in our abstract syntax. Later in this chapter, we still present necessary changes to be made to obtain an equivalent structure that conforms to our abstract syntax. The following example based on the mobile phone product line illustrates a feature model in our abstract syntax satisfying Definition 3.1.

**Example 3.1.** Table 3.2 illustrates a shortened abstract representation of the mobile phone feature model presented in Figure 2.2 on Page 8. Except for the root (*MobilePhone*), each feature has a single parent. Terminal features are  $\langle 0..0 \rangle$  decomposed. Other decompositions are according to Table 3.1 on Page 29. Each feature is concrete, and thus there are no abstract features. Furthermore,  $\Phi$  is a set of three cross-tree constraints in propositional logic.  $\phi_1$  is a simple excludes constraints, whereas  $\phi_2$  and  $\phi_3$  are complex constraints.

Feature	$\omega$	$\lambda$	DE (Parent)
MobilePhone	1	$\langle 5..5 \rangle$	
Calls	1	$\langle 0..0 \rangle$	MobilePhone
GPS	0	$\langle 0..0 \rangle$	MobilePhone
Screen	1	$\langle 1..1 \rangle$	MobilePhone
Basic	0	$\langle 0..0 \rangle$	Screen
Colour	0	$\langle 0..0 \rangle$	Screen
HighResolution	0	$\langle 0..0 \rangle$	Screen
Media	0	$\langle 1..2 \rangle$	MobilePhone
Camera	0	$\langle 0..0 \rangle$	Media
MP3	0	$\langle 0..0 \rangle$	Media
AccuCal	1	$\langle 1..1 \rangle$	MobilePhone
Strong	0	$\langle 0..0 \rangle$	AccuCal
Medium	0	$\langle 0..0 \rangle$	AccuCal
Weak	0	$\langle 0..0 \rangle$	AccuCal
$\Phi = \{\phi_1, \phi_2, \phi_3\}$			
$\phi_1 =$	$\neg \text{GPS} \vee \neg \text{Basic}$		
$\phi_2 =$	$\text{Color} \vee \text{HighResolution} \vee \neg \text{Camera}$		
$\phi_3 =$	$\text{Weak} \Rightarrow \neg \text{HighResolution} \vee \neg \text{GPS}$		

Table 3.2.: Abstract Representation of the Mobile Phone Product Line

In the following, we formally define the terms *configuration*, *product*, and *product line* according to the abstract syntax described above.

**Definition 3.2** (Configuration, Product, Product Line). *We define configuration, product, and product line as follows.*

- A configuration is any element of  $\mathcal{P}(N)$ .
- A product is a configuration that contains only concrete features. Let  $c \in \mathcal{P}(N)$  be a configuration. Then  $p = c \cap P$  is the corresponding product.
- A product line is a set of products, i.e., any element of  $\mathcal{P}(\mathcal{P}(P))$ . Throughout this thesis, we denote any product line by  $\pi$ .

Instead of products, we may also say *program variants*. Moreover, it is necessary to extract the product line from a set of configurations. Hence, we define the projection of one set on another as follows.

**Definition 3.3** (Projection  $A|_B$ ). *For two given sets  $A$  and  $B$ , the projection of  $A$  on  $B$  is defined by  $A|_B := \{a \cap B \mid a \in A\}$*

**Example 3.2.** *Given a set of configurations  $C \in \mathcal{P}(\mathcal{P}(N))$ , where  $C = \{\{A, B\}, \{A, B, C\}\}$  and  $N = \{A, B, C\}$ . Let further be  $P = \{A, C\}$ . Then, the corresponding product line is  $\pi = C|_P = \{\{A\}, \{A, C\}\}$ . Feature  $B$  is abstract as  $N \setminus P = \{B\}$ , and hence omitted in the product line  $\pi$ .*

### 3.2.2. Semantic Domain: Giving Meaning to Syntax

In Definition 3.1, we defined the formal language  $\mathcal{L}_{\mathcal{FFM}}$ , which represents an *abstract syntax* that we use throughout this thesis to formalize feature models (i.e., characterizing what is allowed and disallowed in terms of relationships, cross-tree constraints, ...). Besides syntax, feature models have also a meaning. The meaning of a feature model is obviously its product line (i.e., the set of program variants that conform to the feature model). The set of all possible meanings that a language can have is called the *semantic domain* (Harel and Rumpe, 2000). Giving meaning to feature models allows us to compare them beyond syntax. For instance, two feature models can represent the same product line, yet have different syntax. We will denote the semantic domain of  $\mathcal{FFM}$  by  $\mathcal{S}_{\mathcal{FFM}}$ , and define it as follows.

**Definition 3.4** (Semantic Domain  $\mathcal{S}_{\mathcal{FFM}}$ ). *Given a set  $P$  of concrete features. Then we denote by  $\mathcal{S}_{\mathcal{FFM}} = \mathcal{P}(\mathcal{P}(P))$  the semantic domain, i.e., the set of all possible product lines expressible by the language  $\mathcal{L}_{\mathcal{FFM}}$ .*

It can be seen in Definition 3.5 that here product lines are represented as a mathematical set of sets of features. However, there exist other representations. In the later stage of this section, we introduce propositional logic as a means to represent product lines. But first, we need to connect the syntax of a feature model with its product line. According to Harel and Rumpe (2000), the *semantic function* is the function that maps a syntactic notion to its semantic domain. It is defined as follows.

**Definition 3.5** (Semantic Function  $\llbracket \cdot \rrbracket_{\mathcal{FFM}}$ ). *The semantic function  $\llbracket \cdot \rrbracket_{\mathcal{FFM}} : \mathcal{L}_{\mathcal{FFM}} \rightarrow \mathcal{S}_{\mathcal{FFM}}$  returns a set of all valid feature configurations  $C \in \mathcal{P}(\mathcal{P}(N))$  restricted to concrete features.*

*Given a feature model  $m = (N, P, r, \omega, \lambda, DE, \Phi)$  in  $\mathcal{L}_{\mathcal{FFM}}$ . Then, the semantic function maps  $m$  to its product line, i.e.,  $\llbracket m \rrbracket_{\mathcal{FFM}} = C|_P$  if  $C$  is the set of feature model  $m$ 's valid configurations. Each  $c \in C$  is such that*

- *it contains the root:  $r \in c$ .*
- *it satisfies the decomposition type, i.e.,*

$$\text{for all } f \in c, \text{ if } \lambda_f = \langle a..b \rangle \text{ then } a \leq |sub_f \cap c| \leq b \text{ and } mand_f \subseteq c$$

$$\text{where } mand_f = \{g \in N \mid \omega(g) = 1 \text{ and } f \rightarrow g\}.$$

- *its parent-child-relationships hold: if  $f' \in c$  and  $f' \in sub_f$  then  $f \in c$ .*
- *it satisfies each cross-tree constraint: for all  $\phi \in \Phi$ ,  $c \models \phi$  (i.e., if  $c$  is valid,  $\phi$  must evaluate to true).*

In completion, we define the semantic function that is not restricted to concrete features (i.e., representing all configurations instead of program variants) as  $\llbracket \cdot \rrbracket_{\mathcal{FFM}}^c : \mathcal{L}_{\mathcal{FFM}} \rightarrow \mathcal{P}(\mathcal{P}(N))$ . Moreover, for convenience, we may write  $\llbracket \cdot \rrbracket$  instead of  $\llbracket \cdot \rrbracket_{\mathcal{FFM}}$ .

**Example 3.3.** *Consider the previous example of a mobile phone product line whose feature model shall be denoted by  $m$ . The semantic function then maps the feature model presented in Figure 2.2*

on Page 8 to its product line, which, in this case, is a set of 50 program variants. Each program variant itself is a set of concrete features.

$p_i \in \llbracket m \rrbracket$	Program variant
$p_1$	$\{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{Basic}, \text{AccuCell}, \text{Strong}\}$
$p_2$	$\{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{Colour}, \text{AccuCell}, \text{Strong}\}$
$p_3$	$\{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{HighResolution}, \text{AccuCell}, \text{Strong}\}$
$p_4$	$\{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{Basic}, \text{AccuCell}, \text{Medium}\}$
$p_5$	$\{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{Colour}, \text{AccuCell}, \text{Medium}\}$
$p_6$	$\{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{HighResolution}, \text{AccuCell}, \text{Medium}\}$
$p_7$	$\{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{Basic}, \text{AccuCell}, \text{Weak}\}$
$p_8$	$\{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{Colour}, \text{AccuCell}, \text{Weak}\}$
$p_9$	$\{\text{MobilePhone}, \text{Calls}, \text{Screen}, \text{HighResolution}, \text{AccuCell}, \text{Weak}\}$
...	...
$p_{50}$	$\{\text{MobilePhone}, \text{GPS}, \text{Calls}, \text{Screen}, \text{Colour}, \text{Media}, \text{Camera}, \text{MP3}, \text{AccuCell}, \text{Strong}\}$

Feature models are ambiguous; syntactically different feature models may express the same product line. However, with the semantic function, we can analyze product lines based on set theory. Product lines expressed as sets are unambiguous, as the mapping from distinct product lines to sets of program variants is bijective. This allows us to reason about whether two feature models may express the same product line. Now the question arises, which feature models do exactly express the same product line? For instance, can we find a feature model with only requires and excludes constraints for any product line or do we need complex constraints? Answering this question obviously helps in examining the role of complex constraints in feature modeling. We need to define two more terms to be able to compare feature models based on their product line.

**Definition 3.6** (Feature Model Equivalence). *We say that two feature models  $m, m' \in \mathcal{L}_{\mathcal{FFM}}$  are equivalent if and only if their product lines are identical, i.e.,  $\llbracket m \rrbracket = \llbracket m' \rrbracket$ . We say that  $m, m'$  are strict equivalent if and only if they represent the same set of configurations, i.e.,  $\llbracket m \rrbracket^c = \llbracket m' \rrbracket^c$ . We write  $m \equiv m'$  for equivalent feature models and  $m \equiv_c m'$  for strict equivalent feature diagrams.*

If two feature models are equivalent, their semantic function maps both feature models to the same set of program variants. Strict equivalence is stronger in the sense that both feature models represent the same set of configurations, and thus including abstract features as well. The following example emphasizes the meaning of equivalent and strict equivalent feature models.

**Example 3.4.** *Figure 3.1 illustrates three syntactically different feature models, namely  $m_1, m_2$ , and  $m_3$ , representing the same product line.  $m_1$  and  $m_3$  share even the same configurations, as is illustrated by the list of program variants and configurations of each feature model in Table 3.3. Gray feature names represent abstract features. Hence,  $m_1, m_2$ , and  $m_3$  are equivalent, and  $m_1$  and  $m_3$  are even strict equivalent, i.e.,  $m_i \equiv m_j$  for  $i, j \in \{1, 2, 3\}$  but  $m_1 \not\equiv_c m_2$  and  $m_2 \not\equiv_c m_3$ .*

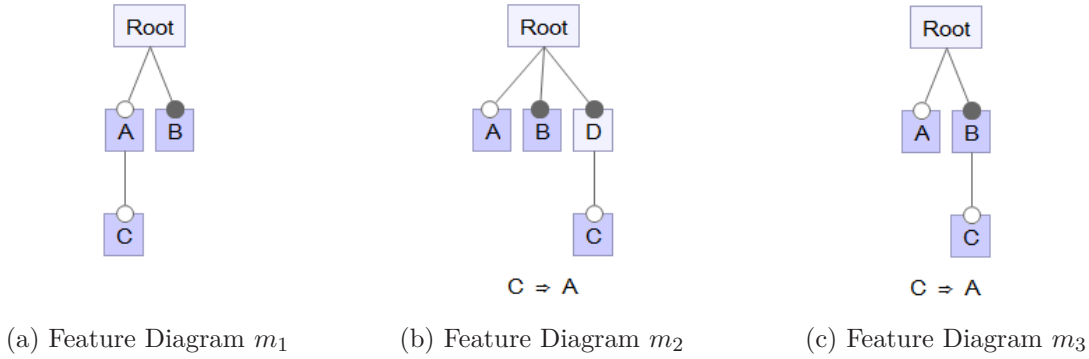


Figure 3.1.: Three Syntactically Different Yet Equivalent Feature Models

Program variants	Configurations $m_1$	Configurations $m_2$	Configurations $m_3$
$\{B\}$	$\{Root, B\}$	$\{Root, D, B\}$	$\{Root, B\}$
$\{A, B\}$	$\{Root, A, B\}$	$\{Root, D, A, B\}$	$\{Root, A, B\}$
$\{A, B, C\}$	$\{Root, A, B, C\}$	$\{Root, D, A, B, C\}$	$\{Root, A, B, C\}$

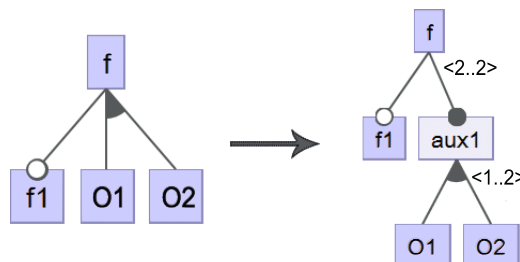
Table 3.3.: Comparison of Program Variants and Configurations

### 3.2.3. Capturing Feature Model Extensions

In Section 2.2, we identified several differences in the concrete syntax of feature modeling languages. For example, FORM and FEATURSEB use directed acyclic graphs in contrast to trees, and the notation by Czarnecki and Eisenecker (2000) allows a feature to have multiple decomposition types. Moreover, we mentioned a special case where mandatory features and group cardinality are mixed. We consider it as necessary to provide instructions on how to transform these three syntactical special cases to our abstract syntax without distorting the product line.

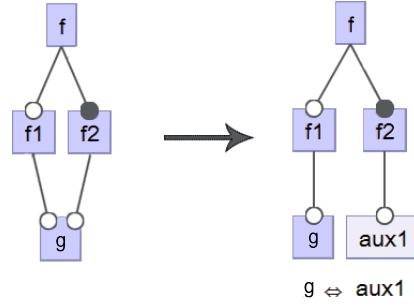
**Multiple decomposition types.** The following changes ensure that every feature has exactly one decomposition type.

- Each *or* and *alternative* decomposition below feature  $f \in N$  is substituted by an auxiliary abstract feature  $aux_i \in N$  such that  $f \rightarrow aux_i$  for each  $i$ .
- $aux_i$  is labeled as mandatory:  $\omega(aux_i) = 1$  for each  $i$ .
- $f$  becomes *and* decomposed (cf. Table 3.1).



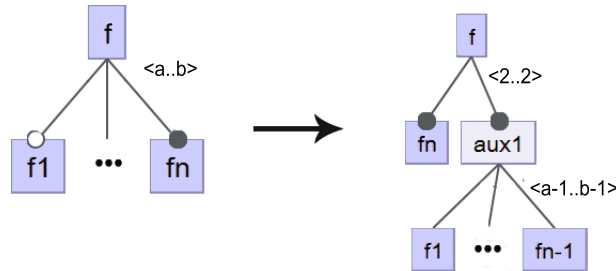
**Directed acyclic graphs.** The following changes convert a feature  $g$  with  $n$  parents  $f_1, \dots, f_n$  to a tree where the feature  $g$  has only one parent  $f_1$ .

- $n - 1$  new auxiliary nodes are added  $aux_1, \dots, aux_{n-1}$  to  $N$ .
- Edges from  $f_2, \dots, f_n$  to  $g$  are replaced by  $f_2 \rightarrow aux_1, \dots, f_n \rightarrow aux_{n-1}$ .
- Additional constraints are added:  $\{aux_i \Leftrightarrow g \mid i \geq 1\} \subseteq \Phi$ .



**Mix of group cardinality and mandatory features.** The following changes separates optional and mandatory features in a group with custom group cardinality.

- Let feature  $f$  be decomposed with  $\lambda(f) = \langle a..b \rangle$  and let features  $f_1, \dots, f_k$  be optional children and  $f_{k+1}, \dots, f_n$  be mandatory children of feature  $f$ . We denote by  $|m| = n - k$  the number of mandatory children of feature  $f$ .
- We create an abstract auxiliary feature  $aux$  such that  $f \rightarrow aux$ .
- $f_1, \dots, f_k$  are moved to  $aux$ :  $aux \rightarrow f_i$  for  $1 \leq i \leq k$ .
- $f$  becomes and decomposed:  $\lambda(f) = \langle |m| + 1, |m| + 1 \rangle$ .
- Finally, the number of mandatory features must be subtracted from feature  $f$ 's original group cardinality for feature  $aux$ :  $\lambda(aux) = \langle a - |m|, b - |m| \rangle$ .



It should be obvious that those trivial changes lead to an equivalent feature model in the context of  $\mathcal{FFM}$ . We therefore waive proofs verifying their correctness. In Chapter 4, we give a more formal definition of a feature model *refactoring*, that is, changing the syntax of a feature model while maintaining semantics. We then present and prove our approach of eliminating complex constraints. Next, we briefly illustrate how feature models can be converted to propositional logic, another representation of product lines.

### 3.2.4. Mapping Feature Models to Propositional Logic

Product lines can be expressed in various ways. Besides feature models, we also expressed product lines as a set of sets of valid feature combinations (i.e., set of program variants), for which we defined the semantic function (cf. Definition 3.5). Another expression of product lines is propositional logic (Batory, 2005). Today's feature models are mostly analyzed by SAT-solvers due to their efficiency as illustrated by Liang et al. (2015). Feature models are converted to a propositional formula that is used as an input for SAT-solvers. Legal feature combinations can then be determined by solving the *satisfiability problem*. Obviously, using this representation of product lines allows us to exploit the whole mathematical background that is given by propositional logic. For instance, we will use propositional logic to prove the correctness of our approach presented in Chapter 4.

The conversion of a feature model to a propositional formula is straight forward. Each decomposition can be translated to a propositional formula (Batory, 2005). The resulting formula is then a conjunction ( $\wedge$ ) of those formulas. Table 3.4 illustrates the mapping from a feature model *primitive* to a propositional formula. We use the equivalent concrete syntax to our abstract syntax for readability.




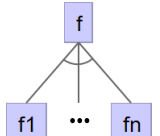
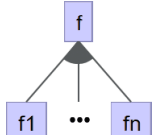
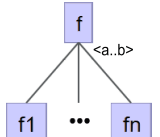
Primitive	Syntax	Propositional Logic
root		$r$
optional sub-feature		$f' \Rightarrow f$
mandatory sub-feature		$f' \Leftrightarrow f$
alternative group		$(f_1 \vee \dots \vee f_n \Leftrightarrow f) \wedge \bigwedge_{i < j} \neg(f_i \wedge f_j)$
or group		$f_1 \vee \dots \vee f_n \Leftrightarrow f$
custom group cardinality		$\bigvee_{M \in P_{a,b}} \left( \bigwedge_{l \in M} l \wedge \bigwedge_{l \in \{f_1, \dots, f_n\} \setminus M} \neg l \right)$ with $P_{a,b} = \{A \in \mathcal{P}(\{f_1, \dots, f_n\}) \mid a \leq  A  \leq b\}$

Table 3.4.: Mapping of Feature Models to Propositional Logic

**Example 3.5.** Consider the following (shortened) feature model of the mobile phone product line depicted in Figure 3.2. Each primitive is individually converted into a propositional formula and connected by conjunction. The resulting propositional formula has then the form



$$\begin{aligned}
& \text{MobilePhone} \\
& \wedge (\text{MobilePhone} \Leftrightarrow \text{Calls}) \wedge (\text{GPS} \Rightarrow \text{MobilePhone}) \\
& \wedge (\text{MobilePhone} \Leftrightarrow \text{Screen}) \wedge (\text{Media} \Rightarrow \text{MobilePhone}) \\
& \wedge (\text{Screen} \Leftrightarrow \text{Basic} \vee \text{Colour}) \wedge (\neg \text{Basic} \vee \neg \text{Colour}) \\
& \wedge (\text{Media} \Leftrightarrow \text{Camera} \vee \text{MP3}) \\
& \wedge (\text{GPS} \Rightarrow \neg \text{Basic}).
\end{aligned}$$

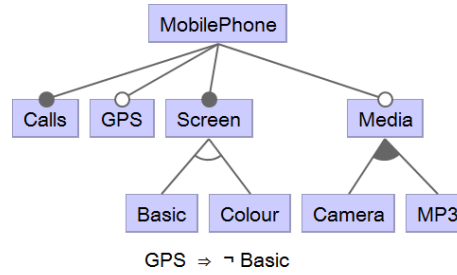


Figure 3.2.: Shortened Version of the Mobile Phone Product Line

### 3.3. Expressive Power of Feature Models

As already mentioned in the beginning of this chapter, the expressive power of a language is commonly understood as the breadth of what can be expressed in a language according to its semantics. For formal languages, as we developed one in the previous section for feature models, we can give a more precise definition of expressiveness (inspired by [Schobbens et al. \(2007\)](#)).

**Definition 3.7** (Expressive Power). *The expressive power of a language  $\mathcal{L}$  is  $E(\mathcal{L}) = \{\llbracket d \rrbracket_{\mathcal{L}} \mid d \in \mathcal{L}\}$ . If  $E(\mathcal{L}_1) \subset E(\mathcal{L}_2)$  then  $\mathcal{L}_2$  is more expressive than  $\mathcal{L}_1$ . If  $E(\mathcal{L}_1) = \mathcal{S}_{\mathcal{L}}$  then  $\mathcal{L}$  is expressive complete.*

The expressiveness  $E$  of a language  $\mathcal{L}$  is therefore a subset of its semantic domain  $\mathcal{S}_{\mathcal{L}}$ , and the codomain of its semantic function  $\llbracket \cdot \rrbracket_{\mathcal{L}}$ . Usually, to prove that a language is as expressive as another language, we must give a translation from the first to the second language and prove that the translation is indeed correct. Expressive completeness is oftentimes easier to show. If  $\llbracket \cdot \rrbracket_{\mathcal{L}}$  is a surjective total function,  $\mathcal{L}$  is expressive complete. This can be analyzed by constructing an algorithm that has any element of  $\mathcal{S}_{\mathcal{L}}$  as input and can construct a syntactic element in  $\mathcal{L}$ . Fortunately, showing that  $\mathcal{L}_{\mathcal{FFM}}$  is expressive complete is fairly easy, as we can exploit the capability of propositional logic in complex constraints, which is known to be expressive complete ([Nolt et al., 1998](#)).

**Theorem 3.1.**  *$\mathcal{L}_{\mathcal{FFM}}$  is expressive complete.*

*Proof.* Let  $\pi \in \mathcal{S}_{\mathcal{FFM}}$  be a product line. We can construct a feature model  $m = (N, P, r, \omega, \lambda, DE, \Phi)$  in  $\mathcal{L}_{\mathcal{FFM}}$  such that for each feature  $f \in P$  the following conditions hold.

- $f$  is a child feature of root  $r$ :  $r \rightarrow f$ .
- $f$  is optional:  $\omega(f) = 0$ .
- $f$  is a terminal feature:  $\lambda(f) = \langle 0..0 \rangle$ .

$r$  is and decomposed. Finally, we have exactly one complex constraint representing the product line in disjunctive normal form:  $\Phi = \{(\bigvee_{p \in \pi} (\bigwedge_{f \in p} f \wedge \bigwedge_{f \in P \setminus p} \neg f))\}$ .  $\square$

Theorem 3.1 is important. Every application or proposed method that builds upon a feature modeling language with complex constraints can be used with every product line. An interesting question is what can be said about the expressiveness of feature modeling languages with only simple constraints? If their expressiveness would be likewise complete, complex constraints would add no expressive value. First, we need to formulate the syntactic domain for a basic feature model to further investigate its expressiveness.

## A Formal Semantics for Basic Feature Models

In Section 2.1.2, we already described the elements of basic feature models. As a reminder, we list the characteristics of basic feature models once again. Features in a basic feature model can be *and*, *or* or *alternative* decomposed. Features in an *and* group are additionally labeled as either mandatory or optional. Terminal features must be concrete. Furthermore, *requires* and *excludes* constraints are only allowed between concrete features.

**Definition 3.8** (Syntactic domain for Basic Feature Models). *A basic feature model in  $\mathcal{L}_{BFM}$  is a 7-tuple  $(N, P, r, \omega, \lambda, DE, \Phi)$  where:*

- $N, P, r, \omega, DE$  follow Definition 3.1.
- $\lambda : N \rightarrow \mathbb{N} \times \mathbb{N}$  is restricted to  $\langle 1..1 \rangle$  (alternative groups),  $\langle 1..n \rangle$  (or groups with  $n$  sub-features),  $\langle n..n \rangle$  (and group with  $n$  sub-features), and  $\langle 0..0 \rangle$  (terminal features).
- All terminal features must be concrete: for all  $f \in N$  with  $\lambda(f) = \langle 0..0 \rangle$  it follows that  $f \in P$
- Only simple constraints between concrete features are allowed:  $\Phi \subset \{f \Rightarrow g \mid f, g \in P\} \cup \{f \Rightarrow \neg g \mid f, g \in P\}$

It can be seen that  $\mathcal{L}_{BFM}$  is a subset of  $\mathcal{L}_{\mathcal{FFM}}$ , as we only restrict the handling of certain aspects like decompositions and cross-tree constraints. Therefore, the semantic function  $\llbracket \cdot \rrbracket_{\mathcal{FFM}}$  (Definition 3.5 on Page 32) holds also for  $\mathcal{L}_{BFM}$ . We may write  $\llbracket \cdot \rrbracket_{BFM}$  to explicitly refer to the syntactic domain of basic feature models.

**Theorem 3.2.**  $\mathcal{L}_{BFM}$  is not expressive complete.

*Proof.* We prove the theorem with proof by contradiction. Assume  $\mathcal{L}_{BFM}$  is expressive complete. Then there exists a syntactic representation in  $\mathcal{L}_{BFM}$  for the product line

$$\{\{A, B\}, \{A, C\}, \{B\}, \{B, C\}, \{A, B, C\}\}$$

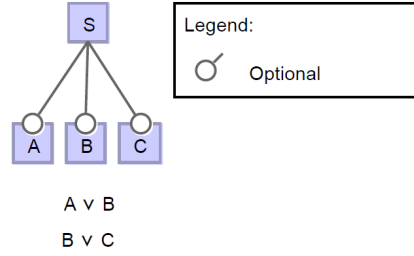


Figure 3.3.: A feature model that has two complex constraints. There exists no feature model in  $\mathcal{L}_{BFM}$  that is equivalent to this representation.

(root feature  $S$  is omitted). Figure 3.3 visualizes the product line in a feature diagram with two complex constraints. Because there is no feature occurring in every product, none of the features  $A$ ,  $B$ , and  $C$  is a mandatory feature of  $S$ . Because no feature is occurring with any other feature in each product, there are no parent-child-relationships or requires constraints between  $A$ ,  $B$ , and  $C$ . Because of product  $\{A, B, C\}$ , there are no excludes constraints and no alternative groups. Because the products  $\{A\}$  and  $\{C\}$  are not part of the product line and there are no excludes constraints, none of these are part of an or group, leaving no or groups at all. Therefore,  $A$ ,  $B$ , and  $C$  must be just optional sub-features of  $S$ , which does not represent the product line either.  $\square$

An important question is: does group cardinality help? Introducing group cardinality does not help either, as we can only specify one interval for the number of feature that should be selected. The counter example above, however, requires us to select two to three features of  $\{A, B, C\}$ , or only  $B$ .

Consider a feature model in  $\mathcal{L}_{FFM}$ . There may be no equivalent feature model in  $\mathcal{L}_{BFM}$ . Corollary, Theorem 3.2 disproves the existence of a translation from  $\mathcal{L}_{FFM}$  to  $\mathcal{L}_{BFM}$ . In the following, we highlight the restricted expressiveness of basic feature models.

## Basic Feature Model Existence

Even though the expressive powers of  $\mathcal{L}_{BFM}$  and  $\mathcal{L}_{FFM}$  differ, there are numerous product lines that can be described by basic feature models. Again, we may ask the question: which product lines can be represented by  $\mathcal{L}_{BFM}$ ? This may further indicate the gap between the usage of complex and only simple constraints. We define the problem of finding a basic feature model given a product line as follows.

**Definition 3.9** (Feature Model Correspondence Problem (FMCP)). *Given a set of concrete features  $P$  and a product line  $\pi \in \mathcal{P}(\mathcal{P}(P))$ . We call the problem whether a feature model  $d \in \mathcal{L}_{BFM}$  exists such that  $\llbracket m \rrbracket_{BFM} = \pi$  feature model correspondence problem.*

**Theorem 3.3.** *The feature model correspondence problem is decidable.*

*Proof.* The number of syntactically distinct feature models in  $\mathcal{L}_{BFM}$  is finite. This can be concluded from the fact that there exist only three types of constraints between any pair of features (two requires and one excludes constraint), and for each concrete feature there exists at most one abstract feature (cf. Definition 3.8). We can construct an algorithm that has a feature model  $m \in \mathcal{L}_{BFM}$  and a product line  $\pi \in \mathcal{P}(\mathcal{P}(P))$  as input, and returns true if  $\llbracket m \rrbracket_{BFM} = \pi$ :

- $m$  can be encoded as a propositional formula (cf. Section 3.2.4). Let  $m_{\mathbb{B}}$  be the propositional formula of  $m$ .
- $\pi$  can be encoded as the disjunctive normal form over the variables in  $P$ :  $\pi_{dnf} = \bigvee_{p \in \pi} (\bigwedge_{f \in p} f \wedge \bigwedge_{f \in P \setminus p} \neg f)$ .
- Then,  $\llbracket m \rrbracket_{BFM} = \pi$  if  $m_{\mathbb{B}} \Rightarrow \pi_{dnf}$  is a tautology. This correlates to the entailment problem, which is known to be decidable and in coNP-complete (Das, 2008, p. 95).

Because  $\mathcal{L}_{BFM}$  is finite, we have eventually valuated each syntactically distinct feature model to either represent  $\pi$  or not.  $\square$

It is important to know that the feature model correspondence problem is decidable. Otherwise, the search for an algorithm to find a basic feature model, given a product line, would be needless. Furthermore, an (desirably efficient) algorithm can help in finding expressive holes in the language of basic feature models. For instance, any product line not expressible by basic feature models might be so uncommon to a degree that it would never be used in industrial cases, rendering the value of complex constraints to a minimum. However, this can only be evaluated if we investigate the expressiveness of basic feature models further.

Unfortunately, to this point, an efficient algorithm to solve the feature model correspondence problem is still missing. However, to illustrate the expressive gap between feature models with complex constraints and feature models with only simple constraints, we rely on a brute force algorithm which generates for a given number of concrete features every possible basic feature model and counts the number of represented product lines.

We only consider product lines that cover every feature from a given set of concrete features. For instance, if  $P = \{A, B\}$  is our feature set with  $|P| = 2$ , we do not care about the product line  $\{\{A\}\}$ , as this product line is already part of all product lines with  $|P| = 1$ . The number of distinct product lines covering a set features  $P$  with  $|P| = n$  is

$$\sum_{k=0}^n \binom{n}{k} (-1)^k 2^{2^{n-k}}$$

The sequence<sup>1</sup> of product lines covering  $n$  features starts then with 2, 2, 10, 218, 64594, 4294642034, ...

Figure 3.4 depicts our results. The brute force method is very time consuming. We only show the results up to five features which already took more than five hours to calculate on a single machine. However, it can be seen that already after five features, less than 1% of all available product lines can be represented by a basic feature model. The number above each stack states the exact number of expressible product lines. Next, we show how to relax some of the restrictions on  $\mathcal{L}_{BFM}$  to increase expressiveness without introducing complex constraints.

<sup>1</sup>The problem of finding all power sets covering a set of elements is well studied and results in OIES sequence A000371. See <https://oeis.org/A000371> for further information.

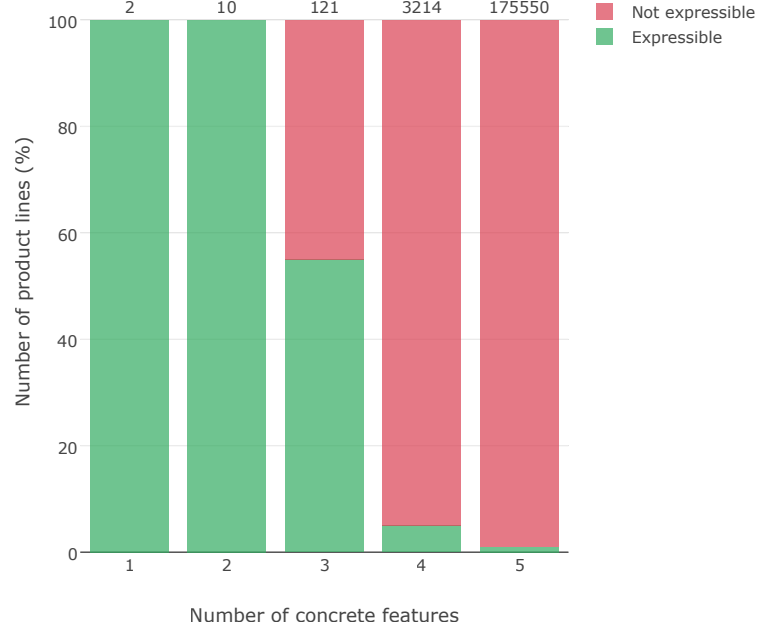


Figure 3.4.: We depict the difference in expressiveness of distinct feature models with only simple constraints up to five features.

## Relaxing Rules on Basic Feature Models

We saw that  $\mathcal{L}_{BFM}$  lacks expressiveness which prevents a translation from  $\mathcal{L}_{FFM}$  to  $\mathcal{L}_{BFM}$ . We therefore endeavor to find an expressive complete sub-language of  $\mathcal{L}_{FFM}$  which is as similar as possible to  $\mathcal{L}_{BFM}$ . Fortunately, relaxing some of the constraints on basic feature models already guarantees expressive completeness, which leads us to the notation of *relaxed basic feature models*.

**Definition 3.10** (Syntactic domain for Relaxed Basic Feature Models). *A relaxed basic feature model in  $\mathcal{L}_{RBFM} \subset \mathcal{L}_{FFM}$  is a 7-tuple  $(N, P, r, \omega, \lambda, DE, \Phi)$  where*

- $N, P, r, \omega, DE$  follow Definition 3.1.
- $\lambda : N \rightarrow \mathbb{N} \times \mathbb{N}$  is restricted to  $\langle 1..1 \rangle$  (alternative groups),  $\langle 1..n \rangle$  (or groups with  $n$  sub-features),  $\langle n..n \rangle$  (and groups with  $n$  sub-features), and  $\langle 0..0 \rangle$  (terminal features).
- Only simple constraints between features are allowed:  $\Phi \subset \{f \Rightarrow g \mid f, g \in P\} \cup \{f \Rightarrow \neg g \mid f, g \in N\}$

The difference to  $\mathcal{L}_{BFM}$  is that we have simply softened the restrictive use on abstract features. Terminals do not need to be concrete anymore, but can be abstract. Moreover, simple constraints between abstract features are now permitted.

**Theorem 3.4.**  $\mathcal{L}_{RBFM}$  is expressive complete.

*Proof.* Let  $\pi \in \mathcal{S}_{FFM}$  be a product line. We can construct a feature model such that for each feature  $f \in P$  the following conditions hold:

- $f$  is a sub-feature of root  $r$ :  $r \rightarrow f$ .
- $f$  is optional:  $\omega(f) = 0$ .
- $f$  is a terminal feature:  $\lambda(f) = \langle 0..0 \rangle$ .

Finally, we add a mandatory node  $T \in N$  to  $r$  with an underlying alternative group. For each product  $p_i \in \pi$ , we create a sub-feature  $t_i$  for  $T$ , where  $i$  is the  $i$ th product. For each feature in product  $p_i$ , we create a single requires constraint:  $g \in p_i \Rightarrow (t_i \Rightarrow g) \in \Phi$ . For every other feature, we create as single excludes constraint  $g \in P \setminus p_i \Rightarrow (t_i \Rightarrow \neg g) \in \Phi$ . Each abstract terminal feature now symbols a product in the product line  $\pi$ . With requires and excludes constraints we explicitly include and exclude features.  $\square$

**Example 3.6.** Figure 3.5 illustrates an equivalent feature model in  $\mathcal{L}_{RBFM}$  to the feature model depicted in Figure 3.3 on Page 39. Both express the product line

$$\{\{A, B\}, \{A, C\}, \{B\}, \{B, C\}, \{A, B, C\}\}.$$

Each abstract terminal feature below feature  $T$  represents exactly one product. Hence, there are five terminal features. Each feature then includes all concrete features of its represented product and excludes all other concrete features.

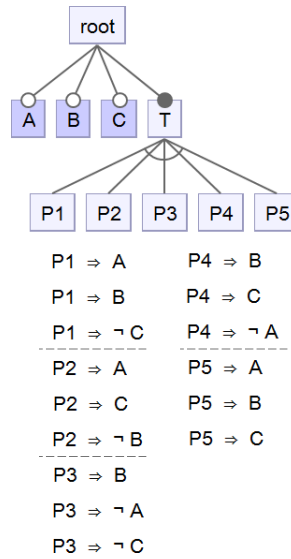


Figure 3.5.: Example of a Potential Feature Model in  $\mathcal{L}_{RBFM}$

We now know that  $\mathcal{L}_{RBFM}$  is expressive complete. Hence, there may exist a more succinct translation from  $\mathcal{L}_{FFM}$  to  $\mathcal{L}_{RBFM}$  as seen in Example 3.6. In the next chapter, we present a more intelligent approach to translate a feature model from  $\mathcal{L}_{FFM}$  to  $\mathcal{L}_{RBFM}$  without changing the product line.

## 3.4. Summary

In this chapter, we created a formal semantics to capture a variety of specific feature model characteristics. Moreover, based on this semantics, we gave insight on the expressive power of

basic feature models. Not all product lines can be covered by basic feature models. An interesting problem is to investigate how many product lines are really covered by basic feature models, and also, what patterns in product line cannot be represented by basic feature models. Future work may deal with the analysis of expressive incompleteness of basic feature models.

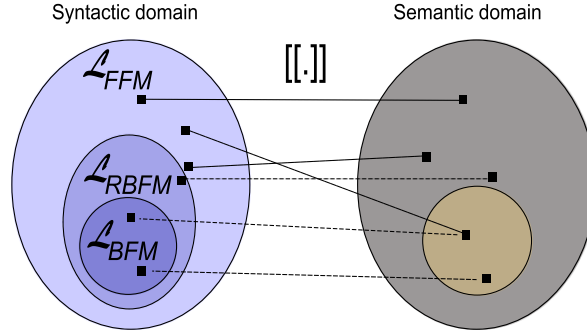


Figure 3.6.: Highlighted differences in expressive power.  $\mathcal{L}_{FFM}$  and  $\mathcal{L}_{RBFM}$  cover the whole semantic domain of product lines whereas  $\mathcal{L}_{BFM}$  covers only a fractional amount.

Moreover, the expressive incompleteness led us to relax some of the restrictions on basic feature models. The result is  $\mathcal{L}_{RBFM}$ , where abstract features can be terminal features and participate in cross-tree constraints. Both languages,  $\mathcal{L}_{FFM}$  and  $\mathcal{L}_{RBFM}$ , are expressive complete. This means that we can give a translation from  $\mathcal{L}_{FFM}$  to  $\mathcal{L}_{RBFM}$ . Figure 3.6 highlights the differences in expressiveness of all three languages.  $\mathcal{L}_{FFM}$  target the whole semantic domain  $\mathcal{L}_{RBFM}$ , whereas  $\mathcal{L}_{BFM}$  is limited to a fractional amount.

In the next chapter, we present an approach to *refactor* feature models from  $\mathcal{L}_{FFM}$  to feature models in  $\mathcal{L}_{RBFM}$ , that is, an approach to eliminate complex constraints without changing semantics.





# 4 Eliminating Complex Constraints

In the previous chapter, we examined the expressive power of feature models in terms of product lines. We identified that some product lines cannot be described by basic feature models. As a result, we introduced the concept of *relaxed basic feature models*, an expressive-complete alternative to basic feature models allowing abstract features (i.e., features that do not influence any program variants) to be part of cross-tree constraints. These abstract features do also not need to have any child features. Hence, we have already addressed part of RQ2: *Under which circumstances is it possible to eliminate complex constraints?*. In this chapter, we introduce our algorithm for eliminating complex constraints by converting a *free feature model* to an equivalent *relaxed basic feature model*.

First, in Section 4.1, we refine our goal of achieving a relaxed basic feature model from an arbitrary one by applying *feature model refactorings* and elaborate on the necessary mathematical properties a refactoring algorithm must fulfill. In Section 4.2, we explain how to eliminate group cardinalities, as they are not allowed in a relaxed basic feature model. Then, in Section 4.3, we continue to illustrate how we simplify the refactoring process by identifying trivial cases and how we can obtain a refactoring of complex constraints using two well-known logical normal forms: *negation normal form* and *conjunctive normal form*. Finally, we give a summary.

## 4.1. General Refactoring of Feature Models

One research goal of this thesis is the elimination of complex constraints and, as a result, obtaining an equivalent basic feature model ( $RG3$ ). As we exposed earlier, basic feature models are less expressive than cardinality-based feature models or feature models with complex constraints (cf. Chapter 3). Some product lines have no expression as a basic feature model, and thus a transformation from a feature model with complex constraint will fail. Hence, we needed to relax some properties of basic features models to acquire a *close* feature modeling notation with the same expressive power as feature models with complex constraints (cf. Definition 3.10 on Page 41).

In Definition 3.6 on Page 33, we already provided a definition of the *equivalence* of feature models according to our constructed formal semantics (cf. Section 3.2). In essence, Sun et al. (2005) and Czarnecki and Wasowski (2007) showed that given a product line there may exist multiple feature models representing this exact product line. We call such feature models *equivalent*.

**Example 4.1.** In Figure 4.1, we depict three feature models noticeably differing in their structure and cross-tree constraints. All three feature models represent the same product line, which is the following set of feature combinations:  $\{\{\emptyset\}, \{B\}, \{A, B\}\}$ . In contrast to the other two feature

diagrams, the right feature diagram has an additional feature  $C$ . The abstract property ensures that feature  $C$  cannot be part of any program variant and is therefore omitted in the product line. Nevertheless, feature  $C$  is relevant in the configuration process, as it enables the selection of feature  $B$ .

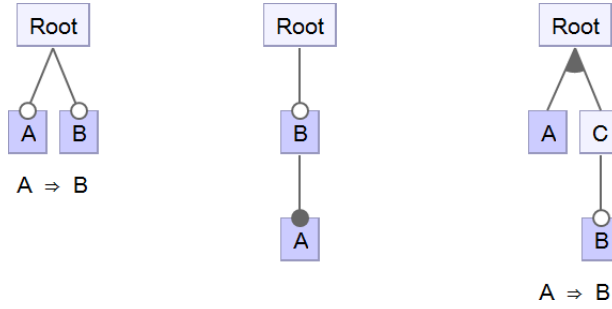


Figure 4.1.: Three Semantically Equivalent Feature Models

In our elimination approach, we aim at changing the structure of a feature model with complex constraints such that the resulting feature model is compliant to the definition of a relaxed basic feature model (cf. Definition 3.10 on Page 41), while also preserving all program variants. Therefore, we call such a change to a feature model a *feature model refactoring*.

**Definition 4.1** (Feature Model Refactoring). A feature model refactoring, *short refactoring*, is a change made to the structure of a feature model, allowing a change to its configurability, but without changing its derivable products. Formally, based on our formal semantics, a refactoring is a total function  $\mathcal{R} : \mathcal{L} \rightarrow \mathcal{L}'$  that preserves semantics:  $m \in \mathcal{L}, \llbracket \mathcal{R}(m) \rrbracket_{\mathcal{L}'} = \llbracket m \rrbracket_{\mathcal{L}}$ .

We use  $\mathcal{L}$  and  $\mathcal{L}'$  as a placeholder for  $\mathcal{L}_{FFM}$  and  $\mathcal{L}_{RBFM}$ , respectively. Obviously, a refactoring can only exist between two languages that are able to express the same set of product lines.

Definition 4.1 is similar to the definition provided by Thüm et al. (2009).<sup>1</sup> Thüm et al. (2009) contributed an algorithm to automatically classify the changes made to a feature model into four classes: *specialization* (program variants are removed), *generalization* (program variants are added), *refactoring* (program variants are maintained), and *arbitrary edit* (program variants are changed). A trivial approach for refactoring complex constraints would be to let a user manually modify the feature model and then automatically check if the made changes result in an equivalent feature model, utilizing the algorithm provided by Thüm et al. (2009). However, with increasing complexity of feature models and complex constraints, modifications become more complicated, and it can be extremely difficult to manually see how to refactor a feature model. Our goal is therefore to provide a refactoring algorithm that automates the refactoring process.

We use the terms *refactoring* and *refactoring algorithm* interchangeably, although their application area slightly differs. Refactoring is the overall concept whereas a refactoring algorithm involves the necessary steps according to our formal semantics to reach such a refactoring.

To formulate a refactoring that transforms an arbitrary feature model to a relaxed basic feature model, we, once again, make use of our formal semantics (cf. Section 3.2). Our formal semantics

<sup>1</sup>Alves et al. (2006) differentiate between uni- and bidirectional refactoring, meaning that a unidirectional refactoring includes an increase in the number of program variants, whereas a bidirectional refactoring follows our definition.

helps us to formally proof the correctness of our approach. In the following, we briefly recall the essentials of our formal semantics.

## From Free Feature Model to Relaxed Basic Feature Model

In Chapter 3, we constructed a general formal semantics called *free feature models* ( $\mathcal{FFM}$ ) as a means to cover most common properties of a number of feature model languages. These properties include basic decompositions (i.e., and, or, alternative), optionality, group cardinalities, and cross-tree constraints consisting of arbitrary propositional formulas. We denote the abstract syntax by  $\mathcal{L}_{\mathcal{FFM}}$ . Then, the semantic function  $\llbracket \cdot \rrbracket_{\mathcal{FFM}}$  maps a feature model  $M \in \mathcal{L}_{\mathcal{FFM}}$  to its product line  $\pi \in S_{\mathcal{FFM}}$  (recall that  $S_{\mathcal{FFM}}$  is the semantic domain of  $\mathcal{L}_{\mathcal{FFM}}$ , i.e., the set of all possible product lines).

Given a feature model  $m \in \mathcal{L}_{\mathcal{FFM}}$ . A valid refactoring leading to a modified feature model  $m'$  is illustrated in Figure 4.2. Applying the semantic function to both feature models,  $m$  and  $m'$ , must result in the same product line  $\pi$ . In the case of our formal semantics, we are also saying that *semantics* is preserved, meaning that both feature models indeed express the same product line.

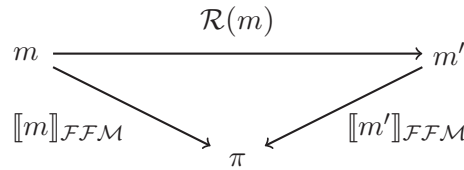


Figure 4.2.: Refactoring of a Feature Model  $m$

$\mathcal{L}_{\mathcal{RBFM}}$  is a subset of  $\mathcal{L}_{\mathcal{FFM}}$ . Only custom group cardinalities and complex constraints are disallowed in  $\mathcal{L}_{\mathcal{RBFM}}$ . Nevertheless,  $\mathcal{L}_{\mathcal{FFM}}$  and  $\mathcal{L}_{\mathcal{RBFM}}$  are both expressive complete (cf. Section 3.3), which leads us to the following corollary.

**Corollary 4.1.** *For all  $\pi \in S_{\mathcal{FFM}}$ , there exist a feature model  $m \in \mathcal{L}_{\mathcal{FFM}}$  and a feature model  $m' \in \mathcal{L}_{\mathcal{RBFM}}$  such that both feature models express  $\pi$ :  $\llbracket m \rrbracket_{\mathcal{FFM}} = \llbracket m' \rrbracket_{\mathcal{RBFM}} = \pi$ . It follows that a refactoring algorithm  $\mathcal{R} : \mathcal{L}_{\mathcal{FFM}} \rightarrow \mathcal{L}_{\mathcal{RBFM}}$  must exist.*

*Proof.* Follows from expressive completeness of  $\mathcal{L}_{\mathcal{FFM}}$  and  $\mathcal{L}_{\mathcal{RBFM}}$  (cf. Section 3.3).  $\square$

Corollary 4.1 ensures that the formalization of a refactoring algorithm is promising. However, finding a sufficient refactoring and formally proving its correctness is a difficult task. One could think of a simpler option that allows the algorithm to produce a *generalization*, that is, allowing the resulting feature model to express a product line that is a superset of the expressed product line of the input feature model. However, as we already know that a refactoring must exist, we are aiming at the more accurate solution. In Example 4.2, we highlight the differences between a *generalization* and a *refactoring*.

**Example 4.2.** *Consider an allegedly constructed refactoring algorithm  $\mathcal{R} : \mathcal{L}_{\mathcal{FFM}} \rightarrow \mathcal{L}_{\mathcal{FFM}}$  that adds an optional concrete node to each terminal node. The transformation using a concrete syntax is depicted in Figure 4.3. The product line of the left-hand side feature model is the*

set  $\pi_{lhs} = \{\{\emptyset\}, \{A\}\}$ . The product line of the right-hand side feature model is the set  $\pi_{rhs} = \{\{\emptyset\}, \{A\}, \{opt2\}, \{A, opt1\}, \{A, opt2\}, \{A, opt1, opt2\}\}$ . Because  $\pi_{lhs}$  is a subset of  $\pi_{rhs}$ ,  $\pi_{rhs}$  is at least as big as  $\pi_{lhs}$ . But because  $\pi_{rhs}$  is not a subset of  $\pi_{lhs}$ ,  $\pi_{lhs}$  must be smaller than  $\pi_{rhs}$ . Therefore,  $\mathcal{R}$  is not a valid refactoring algorithm. Here,  $\mathcal{R}$  produces a generalization. If each added node would be abstract,  $\pi_{lhs}$  and  $\pi_{rhs}$  would be identical.  $\mathcal{R}$  would then be a valid constructed refactoring algorithm.

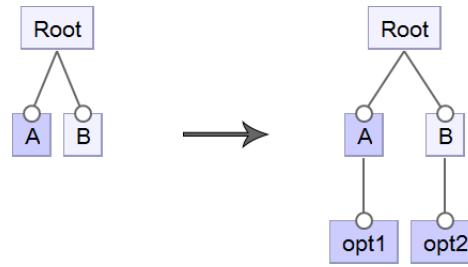


Figure 4.3.: Example of a Feature Model Change

In the following, we informally introduce the base idea of our refactoring algorithm that, given a complex constraint in propositional formula, constructs an equivalent *abstract subtree*. An abstract subtree consists of a feature model with only abstract features and a set of simple constraints in propositional formula. Original feature model and all abstract subtrees can then be conjoined to form an equivalent feature model without complex constraints.

## Refactoring Constraints Through Abstract Subtree Construction

Our approach is based on the idea that feature models can be converted into propositional formulas (cf. Section 3.2.4). Corollary, this induces that we may find a feature model structure (potentially including simple constraints) which, if converted into propositional logic, is equivalent to a given complex constraint. Adding new features to a feature model usually may result in larger program variants. However, our feature modeling language allows abstract features that do not influence the source code and therefore do not harm any program variant. We call such a feature model structure an *abstract subtree* and define it as follows.

**Definition 4.2** (Abstract Subtree). An abstract subtree  $\mathcal{AS}$  is a pair  $(M, \Psi)$ , where  $M = (N, \emptyset, r, \lambda, \omega, DE, \emptyset)$  is a feature model in  $\mathcal{L}_{\mathcal{RFM}}$  with no concrete features and no cross-tree constraints, and  $\Psi$  is a set of requires and excludes constraints in propositional logic.

We are using the term *subtree* to highlight its function in this thesis. An abstract subtree is never used independently but is eventually conjoined with a feature model.

**Example 4.3.** Figure 4.4 shows a feature model (left-hand side) and a possible isolated abstract subtree (right-hand side). An abstract subtree is not a feature model, as both cross-tree constraints are invalid in the isolated abstract subtree (i.e., features  $A$  and  $C$  are missing).

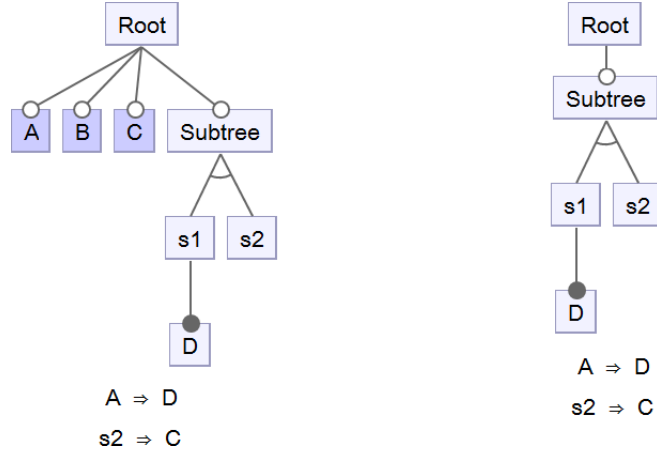


Figure 4.4.: Example of an Original Feature Model and an Abstract Subtree

The idea is now that, given a complex constraint  $\phi$  in propositional formula, we can translate this formula into an abstract subtree. We first traverse through the *parsing tree* of  $\phi$ . Disjunctions ( $\vee$ ) are modeled as or groups and conjunctions ( $\wedge$ ) are modeled as and groups with mandatory sub-features. Literals become abstract terminal features. Each terminal feature in the abstract subtree now represents the occurring of a literal in the complex constraint. We can connect abstract features of the abstract subtree and features used in the original complex constraint through simple constraints. If the literal is positive in the complex constraint, we add a requires constraint. If the literal is negative, we add an excludes constraint.

**Example 4.4.** Figure 4.5 illustrates how to construct an abstract subtree based on a cross-tree constraint. First, the constraint is decomposed multiple times, as highlighted in the parsing tree. In this case, the decomposition types are  $\wedge$  and  $\vee$ , which can be directly translated into and groups and or groups of a feature model. Finally, each terminal feature is associated to either a positive literal (in case of feature A, B, and C) or negative literal (in case of feature D). A simple constraint is added accordingly.

Example 4.4 adumbrates a limitation of our approach. Negations in front of groups with more than one literal cannot be encoded in the abstract subtree, as there exist simply no decomposition type. Furthermore, operators  $\{\Rightarrow, \Leftrightarrow\}$  in a complex constraint cannot be encoded either. Therefore, we first must translate each complex constraint to a form, where negations only occur in literals, and only operators  $\{\wedge, \vee, \neg\}$  may be used.

If we have encoded a complex constraint into an abstract subtree, we finally can conjoin original feature model and abstract subtree. First, we compose the original feature model and the abstract subtree's feature model. Then, we add all abstract subtree's propositional formulas as simple constraints. We define the composition of two feature models as follows.

**Definition 4.3** (Join operator  $\bullet$ ). Let  $m, m'$  be feature models. The join operator  $\bullet$  takes two feature models as input and returns a new combined feature model  $m'' = m \bullet m'$ , which is a merge of parent-child-relationships and cross-tree constraints. Formally,

$$\begin{aligned} m \bullet m' &= (N, Pr, \lambda, \omega, DE, \Phi) \bullet (N', P', r', \lambda, \omega, DE', \Phi') \\ &= (N \cup N', P \cup P', r'', \lambda'', \omega'', DE \cup DE', \Phi \cup \Phi') \end{aligned}$$

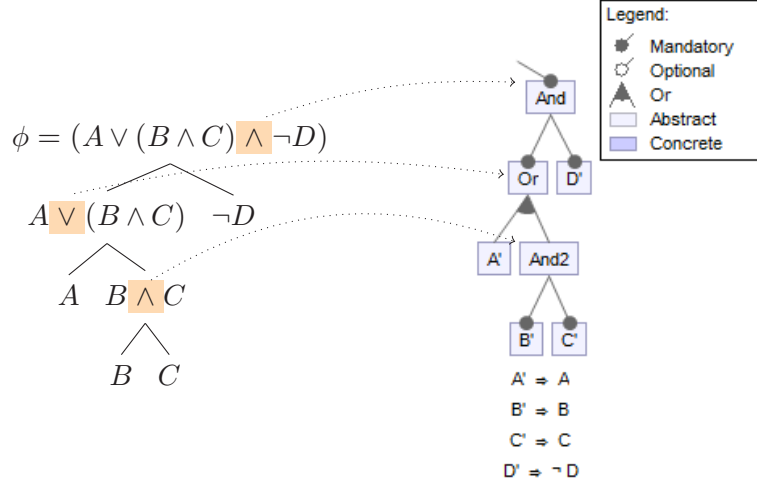


Figure 4.5.: On the left-hand side is the parsing tree of a complex constraint. On the right-hand side is the constructed abstract subtree.

Functions  $\lambda''$  and  $\omega''$  adapt as intended and  $r''$  becomes new root (either  $r$  or  $r'$ ). Hence, for all features  $f \in N \cup N'$ , the following redefinitions take place:

$$\lambda''(f) = \begin{cases} \lambda'(f), & f \in N' \setminus N \\ \lambda(f), & f \in N \setminus N' \\ \langle a..b \rangle, & f \in N \cap N' \end{cases} \quad \omega''(f) = \begin{cases} \omega'(f), & f \in N' \setminus N \\ \omega(f), & \text{otherwise} \end{cases} \quad r'' = \begin{cases} r', & f \in N' \setminus N \\ r, & \text{otherwise} \end{cases}$$

If feature  $f$  is an element of  $N \cap N'$  then  $a$  and  $b$  depend on the decomposition type that feature  $f$  inherits after merge (either and groups, or groups, or alternative-group). We disallow a merge of features with custom group cardinality, as, in this thesis, we use the join operator only between a feature model in  $\mathcal{L}_{\mathcal{FFM}}$  and (possibly many) feature models from abstract subtrees, which are by definition in  $\mathcal{L}_{\mathcal{RBFM}}$  (cf. Definition 4.2 on Page 48).

$m$  and  $m'$  are only allowed to conjoin if and only if the combined feature model would not result in contradicting relationships. Furthermore, all merged features need to have the same decomposition type (either and group, or group, or alternative group), and either  $r$  must be element in  $N'$  or  $r'$  must be element in  $N$ . The join operator allows chaining, but is generally not commutative. Example 4.5 demonstrates the overall conversion procedure from a feature model to an equivalent feature model without complex constraints.

**Example 4.5.** Consider the original feature model shown on the left-hand side in Figure 4.6. Obviously, the only program variant excluded from its product line is  $\{C\}$ . We want to eliminate its single complex constraint  $A \vee B \vee \neg C$  by constructing an equivalent abstract subtree. The abstract subtree, including additional simple constraints, is depicted in the middle of Figure 4.6. We can now remove the complex constraint  $A \vee B \vee \neg C$  and conjoin original feature model and abstract subtree to acquire an equivalent feature model without complex constraints (cf. Figure 4.6). The refactored feature model excludes only the program variant  $\{C\}$ . Hence, both product lines are indeed equivalent.

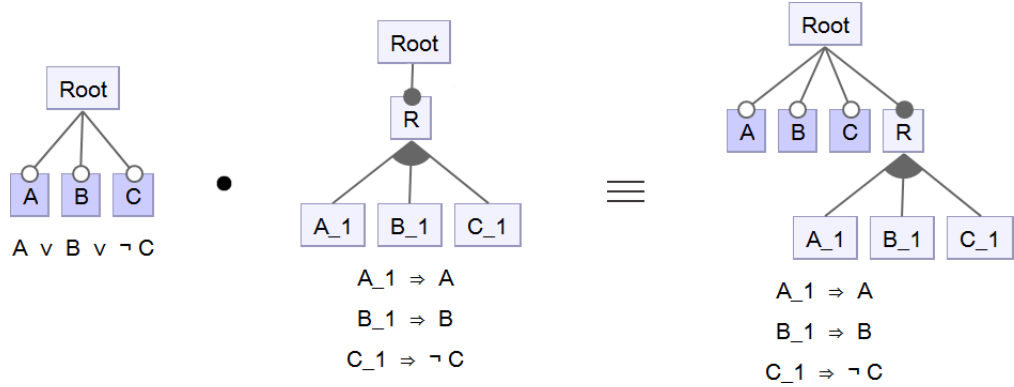


Figure 4.6.: Replacing a Complex Constraint by an Equivalent Abstract Subtree.

## 4.2. Refactoring Group Cardinality

Feature models of  $\mathcal{L}_{\mathcal{FFM}}$  can utilize custom group cardinality. However, our target syntactic domain  $\mathcal{L}_{RBFM}$  disallows them (cf. Definition 3.10 on Page 41).

**Example 4.6.** Consider the part of a feature model shown in Figure 4.7. Selecting feature  $A$  enforces the selection of two out of three features of  $A$ 's sub-features (i.e., features  $B$ ,  $C$ , and  $D$ ). The definition of  $\mathcal{L}_{RBFM}$  requires us to find a transformation that eliminates such group cardinality and at the same time preserves all valid feature combinations.

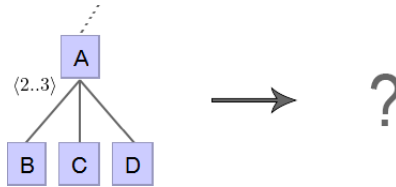


Figure 4.7.: Problem of Refactoring Group Cardinality

We define the refactoring algorithm removing custom group cardinality as follows.

**Definition 4.4** ( $\mathcal{R}_{card}$ ).  $\mathcal{R}_{card} : \mathcal{L}_{\mathcal{FFM}} \rightarrow \mathcal{L}_{\mathcal{FFM}}$  is a refactoring removing group cardinalities from all features of a feature model that neither represent and groups, or groups nor alternative groups (cf. Table 3.1).

As mentioned in Section 3.2, each basic decomposition type (i.e., and group, or group, and alternative group) can also be expressed by group cardinality (cf. Table 3.1).  $\mathcal{R}_{card}$  targets only group cardinalities besides basic decomposition types. In the following we present a simple construction scheme for  $\mathcal{R}_{card}$ .



### Construction of $\mathcal{R}_{card}$

Let  $m = (N, P, r, \omega, \lambda, DE, \Phi)$  be a feature model in  $\mathcal{L}_{\mathcal{FFM}}$ . We construct  $\mathcal{R}_{card}$  the following way.

- Let  $F$  be the set of features having a custom group cardinality:  $f \in F$  if and only if  $\lambda(f) = \langle a..b \rangle$  with  $a, b \notin \{1, |sub_f|\}$  and  $a \leq b$  for each  $f \in N$ .
- Given a feature  $f$  in  $F$  with  $\lambda(f) = \langle a..b \rangle$ . Let  $C_f$  be the set of all valid sub-feature combinations of  $f$ :  $C_f = \{A \mid A \subseteq sub_f \text{ and } a \leq |A| \leq b\}$
- For all features  $f$  in  $F$ , add a complex constraint to  $\Phi$  stating their valid sub-feature combinations of  $C_f$ :  $\{ \bigvee_{P \in C_f} \bigwedge_{l \in P} l \bigwedge_{l \in sub_f \setminus C} \neg l \} \in \Phi$ .
- Finally, change group cardinality of every feature  $f$  in  $F$  to an and decomposition (all features become optional):  $\lambda(f) = \langle n..n \rangle$  with  $n = |sub_f|$ .

**Example 4.7.** The refactoring algorithm  $\mathcal{R}_{card}$  is depicted in Figure 4.8. Once again, the selection of feature  $A$  of the left-hand side feature model enforces the selection of two out of three sub-features of  $A$  (i.e.,  $B$ ,  $C$ , and  $D$ ). The valid feature combinations are therefore  $\{B, C\}$ ,  $\{B, D\}$ ,  $\{C, D\}$ , and  $\{B, C, D\}$ .  $\mathcal{R}_{card}$  changes the group cardinality of feature  $A$  to an and-group with only optional sub-features. Finally, a complex constraint is added that represents the valid feature combinations between features  $B$ ,  $C$ , and  $D$ .

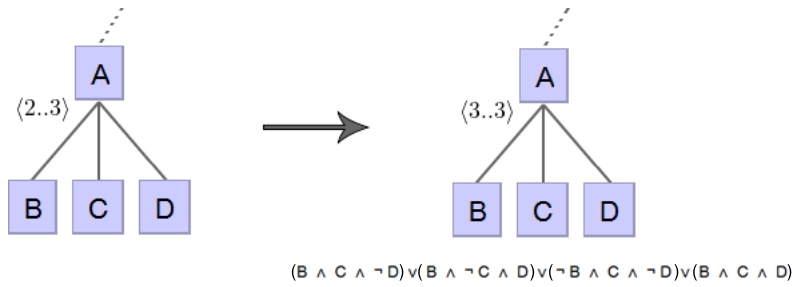


Figure 4.8.: Eliminating Group Cardinality

**Theorem 4.1.**  $\mathcal{R}_{card}$  is a correct refactoring algorithm, i.e., preserves semantics.

*Proof.* Follows directly from Table 3.1, as the according propositional formula of a custom group cardinality is simply added as a complex constraint.  $\square$

After applying  $\mathcal{R}_{card}$  to a feature model, the resulting feature model has no custom group cardinalities left. We are one step closer to a representation in  $\mathcal{L}_{\mathcal{RBFM}}$ . The next and final step is to refactor the feature model such that no complex constraints are left.

## 4.3. Refactoring Complex Constraints

In this section, we introduce our approach for disposing complex constraints in a feature model while preserving program variants. First, in Section 4.3.1, our goal is to identify trivial complex



constraints that request no refactoring of the feature model's structure except the constraints themselves. Then, in Section 4.3.2, we use the negation normal form of a complex constraint to build an equivalent sub-tree with simple constraints. In Section 4.3.3, we discuss advantages and disadvantages using the conjunctive normal form instead of the negation normal form.

### 4.3.1. Pseudo-Complex Constraints and Trivial Simplifications

A trivial simplification in the elimination process is the identification of complex constraints that form a conjunction ( $\wedge$ ) of simple and complex constraints at the highest level. Simple constraints, if identified, can then be chopped off from the constraint without further refactoring the feature model, resulting in smaller complex constraints and, thus, additional (abstract) features overall.

**Example 4.8.** Consider the following complex constraint:

$$\phi = (A \Rightarrow B) \wedge (B \vee C) \wedge (\neg C \vee \neg A)$$

On the highest level,  $\phi$  is a conjunction of sub-formulas. The green highlighted parts mark simple constraints. The first is a requires constraint ( $A$  requires  $B$ ) and the second is an excludes constraint ( $C$  and  $A$  are mutually exclusive). Obviously,  $(B \vee C)$  is the only real complex constraint that needs to be eliminated. The other two constraints can simply be added as two separate simple constraints. After simplification,  $\phi$  is removed and the following constraints are added to the feature model:

$$\begin{aligned}\phi_1 &= A \Rightarrow B \\ \phi_2 &= B \vee C \\ \phi_3 &= \neg C \vee \neg A\end{aligned}$$

Another trivial simplification is the identification of complex constraints that can be converted to an equivalent formula consisting of a conjunction of only simple constraints. In this thesis, we call such complex constraints *pseudo-complex constraints*. Moreover, we call complex constraints that are not pseudo-complex *strict-complex constraints*.

**Definition 4.5** (Pseudo-Complex Constraints). A pseudo-complex constraint is a complex constraint  $\phi$  that is logically equivalent to a propositional formula  $\phi' = \bigwedge_i c_i$  where  $c_i \equiv A \Rightarrow B$  or  $c_i \equiv A \Rightarrow \neg B$  for any  $A, B \in \mathbb{B}$ .

**Definition 4.6** (Strict-Complex Constraints). A strict-complex constraint is a constraint  $\phi$  that is neither pseudo-complex nor simple.

**Example 4.9.** Consider the following cross-tree constraint:

$$\phi = \neg A \vee (B \wedge C)$$

Obviously,  $\phi$  is a complex constraint. However, we can transform  $\phi$  to an equivalent propositional formula, revealing it as pseudo-complex:

$$\phi' = (\neg A \vee B) \wedge (\neg A \vee C)$$

Then, again, we remove  $\phi$  from the feature model and add the following new constraints:

$$\begin{aligned}\phi_1 &= \neg A \vee B \\ \phi_2 &= \neg A \vee C\end{aligned}$$

Analyzing the *conjunctive normal form* of a complex constraint allows us to refactor pseudo-complex constraints. We will introduce the *conjunctive normal form* in more detail in Section 4.3.3. For now, it is only important, that the *conjunctive normal form* has the form of a conjunction ( $\wedge$ ) of clauses, where each clause consists of a disjunction ( $\vee$ ) of literals. Furthermore, any propositional formula can be converted into a conjunctive normal form (Büning and Lettmann, 1999, p. 24ff.). If each clause symbols a simple constraint, it can be trivially refactored.

We also provide refactorings for trivial simplifications and pseudo-complex constraints. We waive the construction schemes for both, as the idea of their functioning should be clear from the examples above. Furthermore, they are not crucial for the refactoring, but serve only as improvements.

**Definition 4.7** ( $\mathcal{R}_{trivial}$ ).  $\mathcal{R}_{trivial} : \mathcal{L}_{FFM} \rightarrow \mathcal{L}_{FFM}$  is a refactoring reducing the size of complex constraints by extracting simple constraints if possible.

**Definition 4.8** ( $\mathcal{R}_{pseudo}$ ).  $\mathcal{R}_{pseudo} : \mathcal{L}_{FFM} \rightarrow \mathcal{L}_{FFM}$  is a refactoring eliminating pseudo-complex constraints.

In the worst case, the conjunctive normal form can lead to an exponential increase in size of clauses and literals (Büning and Lettmann, 1999, p. 24ff.). This may result in an exponential increase in additional features and simple constraints. We therefore do not want to convert a complex constraint into conjunctive normal form before applying  $\mathcal{R}_{trivial}$  at this preprocessing stage. As we will see in Section 4.3.3, we could apply  $\mathcal{R}_{trivial}$  a second time after utilizing the conjunctive normal form for our abstract subtree construction.

**Theorem 4.2.**  $\mathcal{R}_{trivial}$  and  $\mathcal{R}_{pseudo}$  are correct refactoring algorithms, i.e., both preserve semantics.

*Proof.* Follows directly from logical equivalence (Büning and Lettmann, 1999) and the representation of feature model in propositional logic (cf. Section 3.2).  $\square$

### 4.3.2. Refactoring Using Negation Normal Form

In Section 4.1, we already mentioned that our approach of converting complex constraints into an equivalent abstract subtree requires a special form. First, the negation operator may only be applied directly to atoms. Second, the only other allowed connectors are disjunction ( $\vee$ ) and conjunction ( $\wedge$ ). One idea is therefore to convert all complex constraints to a logically equivalent constraint in *negation normal form*.

**Definition 4.9** (Negation Normal Form (Büning and Lettmann, 1999)). A propositional formula is in negation normal form (NNF) if and only if every negation is directly attached to one atom and the only other logical connectors are  $\wedge$  and  $\vee$ . Implicitly, two negations are not permitted to directly follow each other.

**Lemma 4.1.** *For every propositional formula  $\phi$  there exists an equivalent representation  $\phi'$  in negation normal form.*

*Proof.* See (Büning and Lettmann, 1999). □

An algorithm for propagating negations in a propositional formula to its atoms can be acquired by using rules from propositional logic to eliminate  $\Rightarrow$  and  $\Leftarrow$ , De Morgan's laws, and the fact that double negation is equivalent to no negation (Büning and Lettmann, 1999).

**Lemma 4.2.** *Given two propositions  $P$  and  $Q$ , then the following statements hold:*

- $P \Rightarrow Q$  is the same as  $\neg P \vee Q$ .
- $P \Leftarrow Q$  is the same as  $(\neg P \vee Q) \wedge (\neg Q \vee P)$ .
- $\neg(P \wedge Q)$  is the same as  $(\neg P) \vee (\neg Q)$  (De Morgan's first law).
- $\neg(P \vee Q)$  is the same as  $(\neg P) \wedge (\neg Q)$  (De Morgan's second law).
- $\neg\neg P$  is the same as  $P$  (Double negation).

*Proof.* See (Büning and Lettmann, 1999). □

**Example 4.10.** *Given the formula  $\phi = \neg(A \vee B) \wedge \neg C$ . We can convert  $\phi$  to an equivalent representation in negation normal form using De Morgan's second law:  $\phi_{nnf} = \neg A \wedge \neg B \wedge \neg C$ .*

An algorithm that converts a propositional formula to an equivalent propositional formula in negation normal form is depicted in Algorithm 4.1. Boolean connectors  $\Rightarrow$  and  $\Leftarrow$  must be eliminated beforehand according to Lemma 4.2.

---

```

function PROPAGATENEGATION(Formula  $\phi$ , boolean negated)
   $\delta \leftarrow \emptyset$ 
  if  $\phi == \neg\alpha$  then
    negated = !negated
     $\delta = \text{PROPAGATENEGATION}(\alpha, \text{negated})$ 
  else if  $\phi == \alpha \wedge \beta$  then
    if negated then
       $\delta = \text{PROPAGATENEGATION}(\alpha, \text{negated}) \vee \text{PROPAGATENEGATION}(\beta, \text{negated})$ 
    else
       $\delta = \text{PROPAGATENEGATION}(\alpha, \text{negated}) \wedge \text{PROPAGATENEGATION}(\beta, \text{negated})$ 
  else if  $\phi == \alpha \vee \beta$  then
    if negated then
       $\delta = \text{PROPAGATENEGATION}(\alpha, \text{negated}) \wedge \text{PROPAGATENEGATION}(\beta, \text{negated})$ 
    else
       $\delta = \text{PROPAGATENEGATION}(\alpha, \text{negated}) \vee \text{PROPAGATENEGATION}(\beta, \text{negated})$ 
  else  $\triangleright \phi$  is an atom
    if negated then
       $\delta = \neg\phi$ 
    else
       $\delta = \phi$ 
  return  $\delta$ 

```

---

**Algorithm 4.1** Converting a propositional formula to an equivalent formula in negation normal (adapted from Büning and Lettmann (1999))

---

We now give a definition for a refactoring that aims at eliminating complex constraints in a feature model and its according construction scheme utilizing the negation normal form.

**Definition 4.10** ( $\mathcal{R}_{complex}$ ).  $\mathcal{R}_{complex} : \mathcal{L}_{\mathcal{FFM}} \rightarrow \mathcal{L}_{\mathcal{FFM}}$  is a refactoring that eliminates complex constraints in negation normal form by constructing an equivalent abstract subtree.

#### Construction of $\mathcal{R}_{complex}$

Let  $m = (N, P, r, \omega, \lambda, DE, \Phi)$  be a feature model in  $\mathcal{L}_{\mathcal{FFM}}$  and  $\Phi' \subseteq \Phi$  be the set of complex constraints. We construct  $\mathcal{R}_{complex}$  as follows.

- For all  $\phi_i \in \Phi'$ , eliminate logical connectors  $\Rightarrow$  and  $\Leftrightarrow$  in  $\phi_i$  (cf. Lemma 4.2). Afterwards, convert  $\phi_i$  into negation normal form (cf. Algorithm 4.1).
- For all  $\phi_i \in \Phi'$ , construct an abstract subtree  $(M_i, \Psi_i)$  equivalent to the parsing tree of  $\phi_i$  (cf. Section 4.1).
- Finally,  $\mathcal{R}_{complex}(m) = (N, P, r, \omega, \lambda, DE, (\Phi \setminus \Phi') \cup \Psi_1 \cup \dots \cup \Psi_n) \bullet M_1 \bullet \dots \bullet M_{|\Phi'|}$ .

**Example 4.11.** We have already given examples indirectly using the negation normal form for constructing and joining abstract subtrees in Example 4.4 and Example 4.5.

In the following, we formally prove the correctness of  $\mathcal{R}_{complex}$ , i.e.,  $\mathcal{R}_{complex}$  indeed returns an equivalent feature model without complex constraints.

#### Proof of Correctness

To prove that  $\mathcal{R}_{complex}$  is indeed a refactoring algorithm, we exploit the fact that feature models can be translated into propositional formulas to reason about them (Thüm et al., 2009). First, we need to give one definition and two lemmas to ease the proof procedure.

**Definition 4.11** (Boolean Mapping Function  $\mathcal{B}$ ). Let  $N$  be the set of all features. The mapping function  $\mathcal{B} : \mathcal{L}_{\mathcal{FFM}} \rightarrow \mathbb{B}(N)$  maps a feature model to its representation in propositional logic by transforming all occurring relations according to Table 3.4 and concatenating them through logical conjunction ( $\wedge$ ).

**Lemma 4.3.** Given two feature models  $m, m' \in \mathcal{L}_{\mathcal{FFM}}$ . If  $\mathcal{B}(m) \Rightarrow \mathcal{B}(m')$  is a tautology, then  $\llbracket m \rrbracket \subseteq \llbracket m' \rrbracket$ .

*Proof.* We prove this lemma by contradiction. Assume  $\llbracket m \rrbracket \not\subseteq \llbracket m' \rrbracket$ . Then there exists a product  $p$  in  $\llbracket m \rrbracket$  that is not in  $\llbracket m' \rrbracket$ . Let  $\phi_p$  be a propositional formula representing product  $p$ . Then  $\phi_p \wedge \mathcal{B}(m) \wedge \neg \mathcal{B}(m')$  evaluates to true. Since  $\mathcal{B}(m) \Rightarrow \mathcal{B}(m')$  is a tautology, it follows that

$$\begin{aligned}
 & \phi_p \wedge \mathcal{B}(m) \wedge \neg \mathcal{B}(m') \\
 \equiv & \phi_p \wedge \neg(\neg \mathcal{B}(m) \vee \mathcal{B}(m')) \\
 \equiv & \phi_p \wedge \neg(\mathcal{B}(m) \Rightarrow \mathcal{B}(m')) \\
 \equiv & \perp
 \end{aligned}$$

where  $\perp$  is a symbol stating the valuation to false. Hence,  $\mathcal{B}(m) \Rightarrow \mathcal{B}(m')$  cannot be a tautology. This is obviously contradicting to our assumption.  $\square$

Part of our upcoming proof involves showing that the implication between two feature models in propositional formula is a tautology. We can then conclude that one feature model semantically represents a superset of the other feature model. To do so, we need rules to syntactically infer from one formula to another. Furthermore, if we want to show that  $A \Rightarrow B$  is a tautology, we can also show that  $A \Rightarrow A'$  and  $A' \Rightarrow B$  are both tautologies, potentially chaining more steps in between. These rules are also known as logical substitution, as they syntactically change a propositional formula without changing its semantics. For our proof, we only need three rules, depicted in Lemma 4.4.

**Lemma 4.4.** *Given two propositions  $P, Q$ . Then the following three propositional formulas are tautologies.*

- $(P \Rightarrow Q) \wedge P \Rightarrow Q$  (modus ponens)
- $(P \Rightarrow \neg Q) \wedge P \Rightarrow \neg Q$  (modus porendo tollens)
- $(P \Leftrightarrow Q) \wedge P \Rightarrow Q$  (biconditional elimination)

*Proof.* Can easily be shown by constructing truth tables. □

**Example 4.12.** *Consider the propositional formula  $\phi = P \wedge Q \wedge (P \Rightarrow S \vee T)$ . Modus ponens states that if  $\phi$  evaluates to true, so does  $\phi' = (S \vee T) \wedge Q$ .*

**Theorem 4.3.**  $\mathcal{R}_{\text{complex}}$  is a correct refactoring algorithm, i.e., preserves semantics.

*Proof.* We demonstrate  $\llbracket \mathcal{R}_{\text{complex}}(m) \rrbracket_{\mathcal{FFM}} = \llbracket m \rrbracket$  in two steps. For simplicity, we denote  $\mathcal{R}_{\text{complex}}(m)$  as  $m_{\mathcal{R}}$ .

**(1) each program variant of  $\llbracket m_{\mathcal{R}} \rrbracket$  is also program variant of  $\llbracket m \rrbracket$ .** Because of Lemma 4.3, we need to show that  $\mathcal{B}(m_{\mathcal{R}}) \Rightarrow \mathcal{B}(m)$  is a tautology. Without loss of generality, let  $\phi$  be one complex constraint in negation normal form of  $m$  and  $(M, \Psi)$  the corresponding abstract subtree. When we construct the abstract subtree, we start with the root feature  $r$ . We show that  $\mathcal{B}(M) \wedge \Psi \Rightarrow \phi$  is a tautology by utilizing our three substitution rules (cf. Lemma 4.3). We use the notation  $A \rightarrow B$  to indicate that  $A$  is substituted by  $B$ . We have to distinguish the following cases.

- Root feature. According to Table 3.4, root feature  $r$  is always true.
- And groups. Let  $A$  be an abstract feature (and decomposed) and  $\alpha'_1, \dots, \alpha'_n$  its mandatory children (each can be either decomposed further or be a terminal feature). According to Table 3.4, the corresponding conversion in propositional formula is  $(\alpha'_1 \Leftrightarrow A) \wedge \dots \wedge (\alpha'_n \Leftrightarrow A)$ . We assume that feature  $A$  is selected (otherwise all  $\alpha'_i$  would simply be false). For all  $i = 1..n$ , the following substitution follows by rule of *biconditional elimination* (cf. Lemma 4.4).

$$A \wedge (\alpha'_i \Leftrightarrow A) \rightarrow \alpha'_i$$

This leads to the conjunction  $\alpha'_1 \wedge \dots \wedge \alpha'_n$ .

- Or groups. Let  $B$  be an abstract feature (or decomposed) and  $\beta'_1, \dots, \beta'_m$  its children (again, each can be either decomposed further or be a terminal feature). According to Table 3.4, the corresponding conversion in propositional formula is  $\beta'_1 \vee \dots \vee \beta'_m \Leftrightarrow B$ . Analogously, we assume that feature  $B$  is selected. We can once again use rule of *biconditional elimination* (cf. Lemma 4.4).

$$B \wedge (\beta'_1 \vee \dots \vee \beta'_m \Leftrightarrow B) \rightarrow \beta'_1 \vee \dots \vee \beta'_m$$

- Terminal features. Let  $\gamma'_k$  be a terminal feature. We constructed our abstract subtree such that  $\{\gamma'_k \Rightarrow \gamma_k\} \in \Psi$  or  $\{\gamma'_k \Rightarrow \neg\gamma_k\} \in \Psi$  with  $\gamma_k$  being a variable (feature) in  $\phi$ . The substitution using *modus ponens* or *modus ponendo tollens* (cf. Lemma 4.4), respectively, is as follows.

$$\gamma'_k \wedge (\gamma'_k \Rightarrow \gamma_k) \rightarrow \gamma_k \quad \text{or} \quad \gamma'_k \wedge (\gamma'_k \Rightarrow \neg\gamma_k) \rightarrow \neg\gamma_k$$

In total, we can substitute each proposition in  $\mathcal{B}(M) \wedge \Psi$  by the original propositions of  $\phi$ . If we do this for all complex constraints in  $m$ , we have syntactically derived  $\mathcal{B}(m)$  from  $\mathcal{B}(m_{\mathcal{R}})$ , and thus  $\mathcal{B}(m_{\mathcal{R}}) \Rightarrow \mathcal{B}(m)$  is a tautology. Based on Lemma 4.3, it follows that  $\llbracket m_{\mathcal{R}} \rrbracket \subseteq \llbracket m \rrbracket$ .

- (2) **each program variant of  $\llbracket m \rrbracket$  is also program variant of  $\llbracket m_{\mathcal{R}} \rrbracket$ .** We prove this by *reductio ad absurdum*, i.e., that the following does not hold.

There exists at least one product in  $\llbracket m \rrbracket$  that is not in  $\llbracket m_{\mathcal{R}} \rrbracket$ .

$\mathcal{R}_{\text{complex}}$  only adds abstract features and does not change existing parent-child-relationships (cf. Section 4.3). Then it follows that  $\mathcal{R}_{\text{complex}}$  must eliminate program variants by introducing or eliminating cross-tree constraints. The only cross-tree constraints that  $\mathcal{R}_{\text{complex}}$  adds are simple constraints by constructing and composing an abstract subtree  $(M, \Psi)$  from a complex constraint  $\phi$ . Through *modus ponens* and *modus ponendo tollens*, terminal features of  $M$  can be substituted by according literals in  $\phi$ . We also showed in (1) that any complex constraint  $\phi$  is a syntactical consequence of  $\mathcal{B}(M) \wedge \Psi$ . The change of cross-tree constraints we perform does therefore not change the product line. The only way now to eliminate program variants is when  $\mathcal{R}_{\text{complex}}$  introduces new concrete features that are used in  $(M, \Phi)$ , which it does not. Hence, no program variants are removed by  $\mathcal{R}_{\text{complex}}$ , which means that  $\llbracket m \rrbracket \subseteq \llbracket m_{\mathcal{R}} \rrbracket$ .

□

**Example 4.13.** Consider the propositional formula  $\phi = (A \vee (B \wedge C) \wedge \neg D)$  and according

abstract subtree, here denoted as  $(M, \Psi)$ , in Figure 4.5. It follows that

$$\begin{aligned}
\mathcal{B}(M) \wedge \Psi &\equiv \text{And} \wedge (\text{And} \Leftrightarrow \text{Or}) \wedge (\text{And} \Leftrightarrow D') \wedge (A' \vee \text{And2} \Leftrightarrow \text{Or}) \wedge (B' \Leftrightarrow \text{And2}) \\
&\quad \wedge (C' \Leftrightarrow \text{And2}) \wedge (A' \Rightarrow A) \wedge (B' \Rightarrow B) \wedge (C' \Rightarrow C) \wedge (D' \Rightarrow \neg D) \\
&\stackrel{(1)}{\Rightarrow} \text{Or} \wedge (D') \wedge (A' \vee \text{And2} \Leftrightarrow \text{Or}) \wedge (B' \Leftrightarrow \text{And2}) \wedge (C' \Leftrightarrow \text{And2}) \\
&\quad \wedge (A' \Rightarrow A) \wedge (B' \Rightarrow B) \wedge (C' \Rightarrow C) \wedge (D' \Rightarrow \neg D) \\
&\stackrel{(2)}{\Rightarrow} D' \wedge (A' \vee \text{And2}) \wedge (B' \Leftrightarrow \text{And2}) \wedge (C' \Leftrightarrow \text{And2}) \\
&\quad \wedge (A' \Rightarrow A) \wedge (B' \Rightarrow B) \wedge (C' \Rightarrow C) \wedge (D' \Rightarrow \neg D) \\
&\stackrel{(3)}{\Rightarrow} D' \wedge (A' \vee (B' \wedge C')) \wedge (A' \Rightarrow A) \wedge (B' \Rightarrow B) \wedge (C' \Rightarrow C) \wedge (D' \Rightarrow \neg D) \\
&\stackrel{(4)}{\Rightarrow} D' \wedge (A \vee (B \wedge C)) \wedge (D' \Rightarrow \neg D) \\
&\stackrel{(5)}{\Rightarrow} \neg D \wedge (A \vee (B \wedge C)) \equiv \phi
\end{aligned}$$

In (1) we use biconditional elimination to eliminate *And*. In (2) we use biconditional elimination to eliminate *Or*. In (2) we use again biconditional elimination to substitute *And2* with *B* and *C*. Finally, we use modus ponens in (4) for *A'*, *B'*, and *C'*, and modus ponendo tollens in (5) for *D'*.

### 4.3.3. Refactoring Using Conjunctive Normal Form

In the previous section, we used the negation normal form of a complex constraint to construct an equivalent abstract subtree. The subtree's structure depends directly on the formula's parsing tree. Based on this approach, we can convert a complex constraint to a *different*, but equivalent, negation normal form that may be even more simplified. The goal is to scale down complex constraints in terms of number of literals. As we saw in Section 4.3.1, simplifying complex constraints can be achieved by extracting simple constraints. It is only natural to have a look at the *conjunctive normal form*, as simple constraints already are a disjunction of two literals.

We briefly recall the essentials on a propositional formula in conjunctive normal form.

**Definition 4.12** (Conjunctive Normal Form (Davis et al., 1962)). *A propositional formula  $\Phi$  is in conjunctive normal form if and only if it is of the form*

$$c_1 \wedge \dots \wedge c_n$$

and each (so-called) clause  $c_i$  is a disjunction of literals

$$l_1 \vee \dots \vee l_m$$

Analogous, a disjunction of conjunctions is called a disjunctive normal form.

**Lemma 4.5.** *For every propositional formula  $\phi$  there exists an equivalent representation  $\phi'$  in conjunctive normal form.*

*Proof.* See Davis et al. (1962). □

Algorithm 4.2 converts a propositional formula into conjunctive normal form. After converting all complex constraints into conjunctive normal form using Algorithm 4.2, we can apply  $\mathcal{R}_{trivial}$  a second time (cf. Section 4.3.1).  $\mathcal{R}_{complex}$  can then be applied as is.

---

```

function ToCNF(Formula  $\phi$ )
  if  $\phi == \alpha \wedge \beta$  then  $\phi = \text{ToCNF}(\alpha) \vee \text{ToCNF}(\beta)$ 
  else if  $\phi == \alpha \vee \beta$  then
     $\alpha \leftarrow \text{ToCNF}(\alpha)$ 
     $\beta \leftarrow \text{ToCNF}(\beta)$ 
    if  $\alpha == \gamma \wedge \delta$  then
       $\phi \leftarrow (\gamma \vee \beta) \wedge (\delta \vee \beta)$ 
    else if  $\beta == \gamma \wedge \delta$  then
       $\phi \leftarrow (\alpha \vee \gamma) \wedge (\alpha \vee \delta)$ 
    else
       $\phi \leftarrow \alpha \vee \beta$ 
  return  $\phi$ 

```

---

**Algorithm 4.2** Converting a propositional formula to an equivalent formula in conjunctive normal (adapted from Büning and Lettmann (1999))

---

**Example 4.14.** Consider the following propositional formula.

$$\phi = (A \vee (B \wedge C)) \wedge \neg D$$

Transforming formula  $\phi$  into its conjunctive normal form according to Algorithm 4.2 results in

$$\phi_{cnf} = (A \vee B) \wedge (A \vee C) \wedge \neg D$$

The constructed abstract subtree is depicted in Figure 4.9. It can be seen that the subtree consists of only two levels besides its root feature CNF.

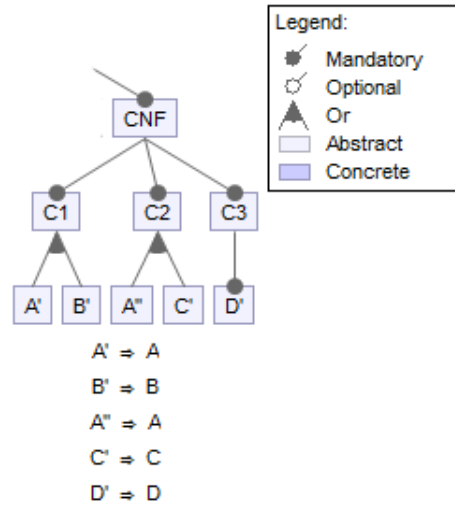


Figure 4.9.: Resulting Abstract Subtree Based on Conjunctive Normal Form

**Example 4.15.** Consider the following cross-tree constraint.

$$\phi = \neg A \vee (B \wedge C) \wedge (D \vee E)$$

Formula  $\phi$  is a complex constraint. However, we can transform  $\phi$  to an equivalent propositional formula in conjunctive normal form:

$$\phi' = (\neg A \vee B) \wedge (\neg A \vee C) \wedge (D \vee E)$$



*In this case, the conjunctive normal form results in a simplification (two simple constraints highlighted in green), as the only complex part left is  $(D \vee E)$ .*

We already mentioned earlier that, although the conjunctive normal leaves more space for simplifications, it is not always the better option. Translating a formula to conjunctive normal form may lead to an exponential increase of the formula in number of clauses and literals (Büning and Lettmann, 1999; Tseitin, 1983), resulting in a more complex feature model as when the standard negation normal form is used. However, an option for practical use might be to decide individually for each complex constraint which form to use. In Chapter 6, we evaluate the strengths and weaknesses of each strategy.

**Example 4.16.** *Consider the following propositional formula in negation normal form.*

$$\phi = \neg F \vee (A \wedge X) \vee (B \wedge (Y \vee Z)) \vee C$$

*Once again, formula  $\phi$  is a complex constraint. If we transform  $\phi$  to an equivalent propositional formula in conjunctive normal form, we get the following result.*

$$\phi' = (\neg F \vee A \vee B \vee C) \wedge (\neg F \vee A \vee Y \vee Z \vee C) \wedge (\neg F \vee X \vee B \vee C) \wedge (\neg F \vee X \vee Y \vee Z \vee C)$$

*In this case, the conjunctive normal form results in a bigger formula, leading eventually to a bigger feature model.*

#### 4.3.4. One-to-One Correspondence of Configurations

Recall that, at Definition 3.6 on Page 33, we defined two meanings for semantic equivalence of feature models. First, basic *equivalence* in terms of program variants. Second, *strict equivalence* in terms of configurations. Our approach so far complies with equivalence but cannot comply with strict equivalence, as we introduce additional abstract features.

However, when we introduce new features, as presented in the previous sections, we also increase the number of configurations, which is not always desired. For instance, tools that are not aware of the concept of abstract features will configurations mistake with program variants. If we then introduce too many new configurations in our approach, the resulting feature model is far from equivalent.

To preserve the initial configuration state, we can at least create a bijection between old and new configurations. We therefore replace all requires constraints in our abstract subtree with a bidirectional implication (that can obviously split up into two requires constraints). Each old configuration is then a subset of its corresponding new configuration. We say that a refactoring algorithm who provides such a bijection is *coherent*, and define it as follows.

**Definition 4.13** (Configuration Coherence). *Given a feature model  $m$  and a refactoring algorithm  $\mathcal{R}$ .  $\mathcal{R}$  is coherent if and only we can pair all  $p \in \llbracket m \rrbracket^c$  with exactly one  $q \in \llbracket \mathcal{R}(m) \rrbracket^c$  such that  $p \subseteq q$ . Otherwise we say that  $\mathcal{R}$  is incoherent.*

**Example 4.17.** *Consider again the abstract subtree in Figure 4.9. Supplementary to  $A' \Rightarrow A$ , we add  $A \Rightarrow A'$ . Analogous for  $B$  and  $C$ . Excludes constraints are symmetrical and just remain as*

they are. Figure 4.10 illustrates the result when a bijection between configurations is established. The additional simple constraints are highlighted in red.

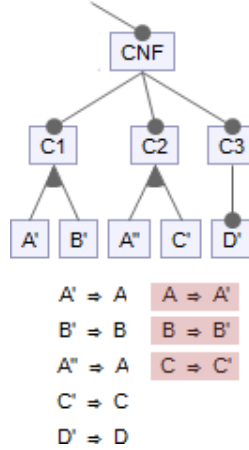


Figure 4.10.: Resulting Feature Model when a Coherent Refactoring is Performed

As can be seen in Example 4.17, our refactoring algorithm for strict complex constraints can simply be extended through adding a bidirectional implication for each requires constraint to become coherent. A disadvantage is, however, the increase in additional cross-tree constraints.

## 4.4. Summary

We proposed an approach to refactor a feature model with complex constraints to a relaxed basic feature model. We first construct semantically equivalent *abstract subtrees* for each complex constraint that are then conjoined with the original feature model. The approach additionally involves identifying pseudo-complex constraints to decrease the size of the resulting feature model, and simplifying constraints beforehand in general. In the following, we summarize all defined refactoring algorithm that are involved in the overall refactoring process.

### Required Refactorings

Recall that there are only two main differences between  $\mathcal{L}_{\mathcal{FFM}}$  and  $\mathcal{L}_{\mathcal{RBFM}}$ . On the one hand,  $\mathcal{L}_{\mathcal{FFM}}$  can make use of group cardinalities which  $\mathcal{L}_{\mathcal{RBFM}}$  cannot. On the other hand,  $\mathcal{L}_{\mathcal{RBFM}}$  allows only simple constraints, whereas  $\mathcal{L}_{\mathcal{FFM}}$  allows complex constraints. To provide a feature model refactoring from  $\mathcal{L}_{\mathcal{FFM}}$  to  $\mathcal{L}_{\mathcal{RBFM}}$ , we have constructed the refactoring algorithms summarized in Table 4.1. Refactoring  $\mathcal{R}_{total}$  represents the overall refactoring procedure. Refactoring  $\mathcal{R}_{complex}$  uses either the negation normal form, conjunctive normal form, or a combination of both. The end result is a feature model in  $\mathcal{L}_{\mathcal{RBFM}}$ . In the next chapter, we give an implementation in FEATUREIDE for refactoring feature models with complex constraints to feature models with only simple constraints.

Refactoring Algorithm	Description
$\mathcal{R}_{card} : \mathcal{L}_{FFM} \rightarrow \mathcal{L}_{FFM}$	$\mathcal{R}_{card}$ results in an equivalent feature model without group cardinalities (cf. Section 4.2).
$\mathcal{R}_{trivial} : \mathcal{L}_{FFM} \rightarrow \mathcal{L}_{FFM}$	$\mathcal{R}_{trivial}$ results in an equivalent feature model with potentially shrunk complex constraints (cf. Section 4.3.1).
$\mathcal{R}_{pseudo} : \mathcal{L}_{FFM} \rightarrow \mathcal{L}_{FFM}$	$\mathcal{R}_{pseudo}$ results in an equivalent feature model with no more pseudo-complex constraints (cf. Section 4.3.1).
$\mathcal{R}_{complex} : \mathcal{L}_{FFM} \rightarrow \mathcal{L}_{FFM}$	$\mathcal{R}_{complex}$ results in an equivalent feature model without complex constraints (cf. Section 4.3)
$\mathcal{R}_{total} : \mathcal{L}_{FFM} \rightarrow \mathcal{L}_{RBFM}$	$\mathcal{R}_{total}$ is the total refactoring algorithm comprising all other refactorings: $\mathcal{R}_{total} = \mathcal{R}_{card} \circ \mathcal{R}_{trivial} \circ \mathcal{R}_{pseudo} \circ \mathcal{R}_{complex}$

Table 4.1.: Overview of Defined Refactoring Algorithms



# 5 Eliminating Complex Constraints with FeatureIDE

In Section 2.3, we took a look at five different areas for feature model applications and transpired that only roughly 50% out of 26 of our reviewed publications discussed complex constraints. However, we know from Section 3.3 that complex constraints add expressive value to a feature modeling language. The problem is that it may be hard to extend some feature modeling applications, that use only simple constraints, to support complex constraints. Our approach for eliminating complex constraints in feature models from the previous chapter may overcome this limitation. To get more insight of the practical significance of our algorithm, we need to conduct an empirical evaluation with industrial feature models. Therefore, we decided to implement our refactoring algorithm in FEATUREIDE.<sup>1</sup>

Moreover, our approach enables the possibilities to export feature models with complex constraints to basic feature model formats (cf. Section 2.2). This is a practical application for our approach that we also want to integrate in FEATUREIDE. We chose the FAMA file format serving as a real application scenario for the implementation of an exporter to a basic feature model format in FEATUREIDE.

In Section 5.1, we introduce all necessary tools and explain how our concept is implemented. Then, in Section 5.2, we illustrate necessary preprocessing steps. Next, we explain implementation details of our elimination strategies in Section 5.3. We elaborate on the implementation steps for our FAMA exporter in Section 5.4. Finally, we summarize this chapter in Section 5.5.

## 5.1. Overview

As mentioned in the beginning, FEATUREIDE is an Eclipse-based open-source framework for feature-oriented software development. In general, FEATUREIDE aims at covering the the whole development process of software product lines, from feature modeling over code artifact implementation to program-variant generation. FEATUREIDE offers a graphical editor based on Eclipse for modeling feature diagrams (cf. Section 2.2.1 for an example of its format) and supports different software product line composition techniques (e.g., aspect-oriented programming, feature-oriented programming, and preprocessor-based development).

Our approach presented in Chapter 4 focuses on a refactoring of feature models. Hence, we find ourselves primarily in the phase of feature modeling. Our goal is to enhance FEATUREIDE's graphical modeling editor to give support for the elimination of complex constraints. We provide the elimination of complex constraints of a feature model in FEATUREIDE by an entry in the context

---

<sup>1</sup>Available in FEATUREIDE v3.1.0: <https://github.com/FeatureIDE/FeatureIDE/releases/tag/v3.1.0>

menu. Moreover, we provide an exporter for the FAMA file format based on our approach, also accessible by the context menu. Figure 5.1 shows a new feature modeling project in FEATUREIDE and the context menu of the project's feature model, offering both of our extensions.

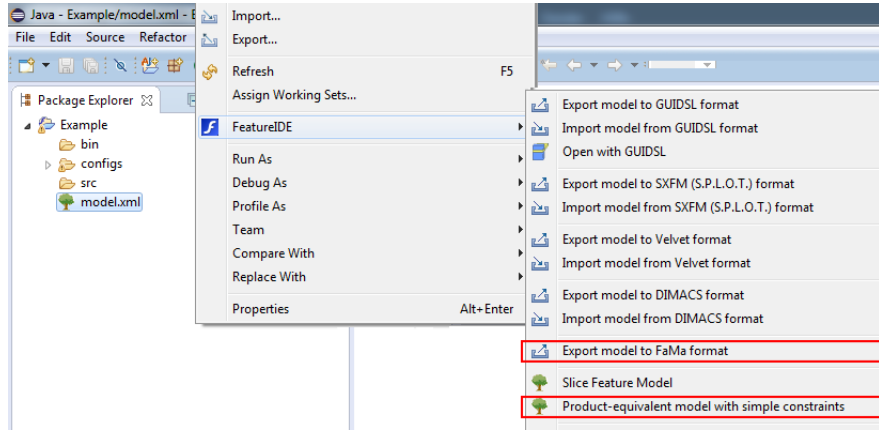


Figure 5.1.: Feature Modeling Project in FeatureIDE with Context Menu

## Refactoring Implementation

Figure 5.2 depicts the refactoring overview as a sequence of actions. We begin with a preprocessing phase where tautological and redundant constraints are removed and pseudo-complex constraints are refactored as described in Section 4.3.1. Subsequently, we iterate through all strict-complex constraints and eliminate them by constructing a equivalent abstract subtree which is then conjoined with the original feature model (cf. Section 4.1).

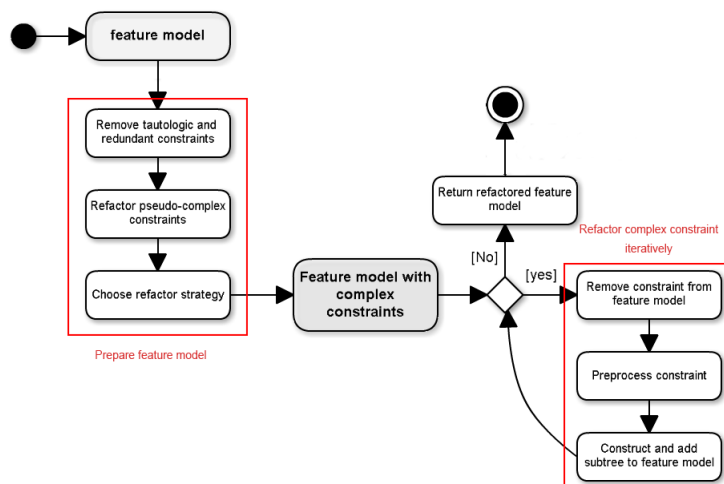


Figure 5.2.: Activity Diagram of the Refactoring Process

To provide a main anchor point for our refactoring process, we implemented the class `Complex-ConstraintConverter` in a newly created package called `conversion`. We refer to our process as *conversion*, as we convert strict-complex constraints before elimination to either conjunctive

---

<code>IFeatureModel convert(IFeatureModel m, IConverterStrategy conv)</code>
<b>Description.</b> <i>The conversion method is an accurate portrayal of the activity diagram shown in Figure 5.2. The result is an equivalent feature model without complex constraints.</i>

---

<code>static boolean isSimple(IConstraint c)</code>
<b>Description.</b> <i>isSimple checks whether c is a simple constraint.</i>

---

<code>static boolean isComplex(IConstraint c)</code>
<b>Description.</b> <i>isComplex checks whether isSimple(c) evaluates to false.</i>

---

<code>static boolean isPseudoComplex(IConstraint c)</code>
<b>Description.</b> <i>isPseudoComplex checks whether c is a conjunction of simple constraints. isPseudoComplex converts c into conjunctive normal form beforehand.</i>

---

<code>static boolean onlyTrivialRefactoring(List&lt;IConstraint&gt; C)</code>
<b>Description.</b> <i>onlyTrivialRefactoring checks whether isPseudoSimple(c) evaluates to true for all c in C.</i>

---

Table 5.1.: Publicly Available Methods of Class ComplexConstraintConverter

normal form or negation normal form, and eventually to an abstract subtree. ComplexConstraintConverter offers static methods to recognize simple, pseudo-complex, and complex constraints in general, and is responsible for the refactoring process depicted in Figure 5.2. Table 5.1 gives an overview of our publicly implemented methods in class ComplexConstraintConverter.

Remember that all pseudo-complex constraints are complex constraints by definition (cf. Section 4.3.1). It is therefore important to first refactor pseudo-complex constraints into simple constraints. Afterwards, if a constraint is recognized as *complex*, it is ensured that it is a strict-complex constraint. Next, we briefly explain the actions we take in our preprocessing phase.

## 5.2. Preprocessing Phase

The preprocessing phase aims at simplifying the input feature model by removing tautological and redundant constraints, as a refactoring of those is unnecessary by definition. However, this is optional for a user based on two observations. First, a user may want to get a refactored feature model that inherits all flaws from the original one. Second, the recognizing of tautological and redundant constraints is costly, as a SAT-analysis is required. For large feature models, this can consume a huge amount of time.





each strict-complex constraint into an equivalent abstract subtree. Abstract subtree and original feature model are then conjoined. Both of our strategies (i.e., conjunctive normal form and negation normal form) create an abstract subtree that is added directly below the root feature. Future strategies may choose to add abstract subtrees to other (potentially concrete) features of the feature model.

Listing 5.2 shows the preprocessing method of the negation normal form strategy. For the negation normal form, the returned list comprises only one constraint. First, all logical connectors instead of  $\{\wedge, \vee, \neg\}$  are eliminated by the rules of propositional calculus. Then, the negation  $\neg$  is recursively propagated until it vanishes or reaches an atom (cf. Section 4.3.2).

```

1  public class NNFCConverter implements IConverterStrategy {
2      /* ... */
3      Override
4      public List<IConstraint> preprocess(IConstraint constraint) {
5          List<IConstraint> result = new LinkedList<IConstraint>();
6          IConstraint elem = constraint.clone();
7
8          elem = eliminateNotSupportedConnectors(elem);
9          elem = propagateNegation(elem, false);
10
11         result.add(elem);
12         return result;
13     }
14     /* ... */
15 }

```

Listing 5.2.: Preprocessing Method of NNFCConverter

A conversion strategy must only be chosen if there are any complex constraints left that cannot be refactored trivially (cf. Section 4.3). In this case, we provide a wizard that guides a user through the elimination process.

### A Wizard for Equivalent Conversion

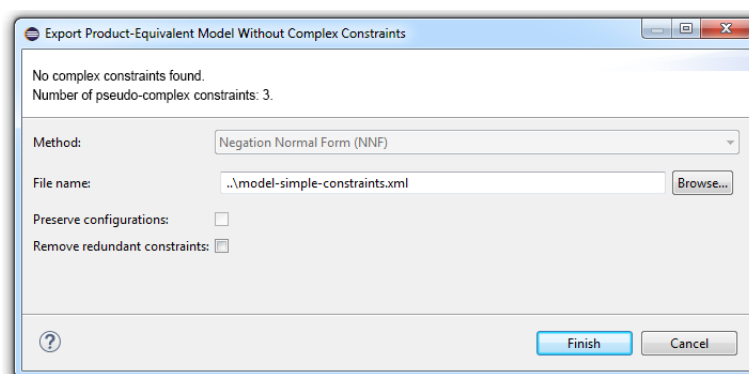


Figure 5.4.: A Wizard for the Conversion Process

Beside the options to choose a file path, a user can have tautological and redundant automatically removed, and can demand a coherent refactoring through the option to preserve semantics (cf. Section 4.3.4). Furthermore, we provide three different strategies for constructing all abstract subtrees if strict-complex constraints exist.

**Negation Normal Form:** Uses the negation normal form of a strict-complex constraint to construct an abstract subtree that is conjoined with the feature model (cf. Section 4.3.2). Implemented by class NNConverter.

**Conjunctive Normal Form:** Analogous to the negation normal form, but uses the conjunctive normal form instead. Implemented by class CNFConverter.

**Combined:** Mixes both strategies and decides individually for each strict-complex constraint which of the other two methods to use (see explanation below). Can also be used with future strategies implementing the interface IConverterStrategy. Implemented by class CombinedConverter.

The *combined method* is an optimization. It estimates for each complex constraint the increase in size regarding additional features and simple constraints and chooses the method with minimal impact according to a *cost function*. Given a strict-complex constraint  $c$ , the number of estimated additional simple constraints denoted as  $|c|_\phi$ , and the number of estimated additional abstract features denoted as  $|c|_F$ . The according cost function is as follows.

$$f(c) = w_\phi * |c|_\phi + w_F * |c|_F$$

$w_\phi$  and  $w_F$  represent weights to penalize either additional constraints or additional features. In our implementation, constraints and features are equally weighted. Listing 5.3 shows the simple preprocessing method of CombinedConverter. As mentioned, any conversion strategy implementing the IConverterStrategy-interface may participate. After choosing the constraint with minimal impact, the equivalent abstract subtree can be conjoined with the input feature model as explained in Section 4.3.2. Listing 5.3 presents the preprocessing method of CombinedConverter. The cost function is implemented by the method estimatedCosts.

```

1  public class CombinedConverter implements IConverterStrategy {
2    /*...*/
3    List<IConverterStrategy> strategies = new LinkedList<
        ↳ IConverterStrategy>();
4    /*...*/
5    Override
6    public List<IConstraint> preprocess(IConstraint constraint) {
7        List<IConstraint> result = new LinkedList<IConstraint>();
8
9        int costs = Integer.MAX_VALUE;
10       for(IConverterStrategy strat : strategies) {
11           List<IConstraint> preprocessed = strat.preprocess(constraint);
12           int cost = 0;
13           if((cost = estimatedCosts(preprocessed)) < costs) {
14               result = preprocessed;
15               costs = cost;
16           }
17       }
18       return result;
19   }
20 }

```

Listing 5.3.: Preprocessing Method of CombinedConverter

The class diagram depicted in Figure 5.5 illustrates the relationships between the interface `IConverterStrategy` and its implementations `NNFConverter`, `CNFConverter`, and `CombinedConverter`.

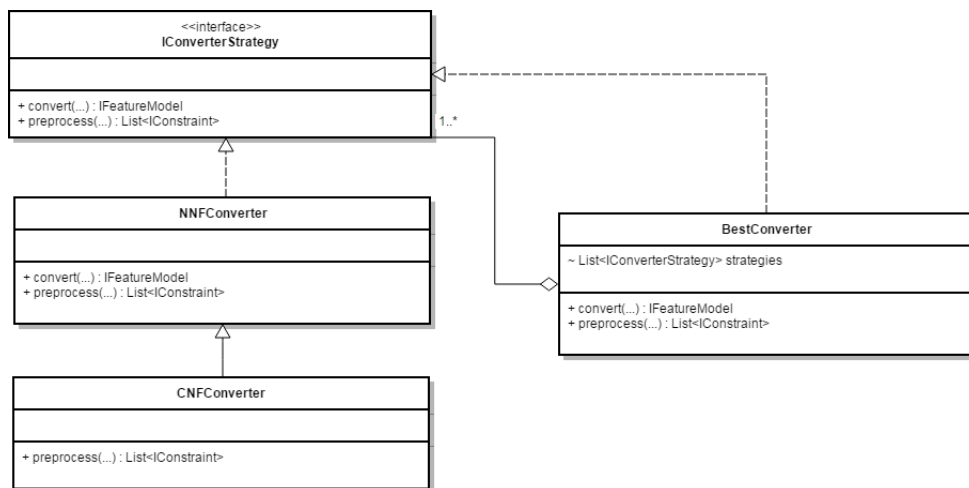


Figure 5.5.: Class diagram depicting the relationships between all currently implemented refactoring strategies.

`CNFConverter` inherits from `NNFConverter` and only needs to override the preprocessing method. The conversion process is for both strategies the same, as conjunctive normal form is only a special case of negation normal form. Therefore, it may happen that both strategies, given a strict-complex constraint, produce an identical abstract subtree.

`CombinedConverter` holds a number of strategies for which it decides which to use for prepro-

cessing each strict-complex constraint individually. In its convert method, it also produces an abstract subtree for each preprocessed constraint according to minimal impact.

## 5.4. Implementing an Exporter for the Fama File Format

FAMA is a framework for automated analysis of feature models (Benavides et al., 2007). We chose to implement an exporter to a basic feature model file format for FEATUREIDE as a real application scenario for our approach. FAMA satisfies this need, as it is highly used for automated analysis of feature models.

FAMA offers two file formats.<sup>2</sup> First, a file format that can use group cardinalities but only simple constraints. Second, an extended format with complex and attributed constraints. Nevertheless, we decided to implement the first file format, which is better integrated. We refer to Section 2.2.2 for a description of the plain text FAMA file format for basic feature models.

FEATUREIDE offers already classes for exporting feature models to different file formats (SXF, VELVET, GUIDSL, and DIMACS). We provide a new package named `de.ovgu.featureide.fm.io.fama` and implement the class `FAMAWriter`, which inherits from `AbstractFeatureModelWriter`. First, a feature model clone is refactored as explained in the previous sections. Second, the refactored model is exported into the FAMA file format. However, all abstract features become concrete in the process. We therefore offer a user the option to perform a coherent refactoring (cf. Figure 5.4).

Table 5.2 exemplifies the conversion from a feature model with complex constraints (left-hand side) to the FAMA file format (right-hand side). The refactored feature model (center) is the result of an incoherent refactoring with our approach. In this case, all three strategies (conjunctive normal form, negation normal form, and the combined method) produce the same result, as both strict-complex constraints are already in conjunctive normal form.

### Known Limitations

Our result is obviously not optimal regarding minimal additional features and constraints, as the implementation of a minimization of all constraints while also preserving the structure of the feature model is beyond the scope of this thesis.

Moreover, the FAMA file format does not support abstract features. This obviously leads to a non-equivalent feature model in the sense of this thesis. However, products are only increasing in terms of additional features. This means that at least every product of the input feature model is also part of a product of the FAMA feature model. If a coherent refactoring performed, there is a one-to-one mapping between configurations of the input feature model and products of the FAMA feature model.

<sup>2</sup>Further information in the user manual: <https://famats.googlecode.com/files/FaMa%20User%20Manual.pdf>

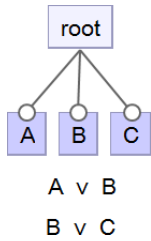
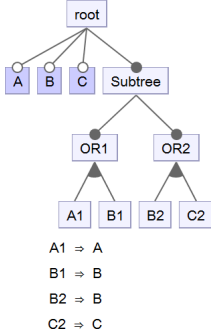
Original Feature Model	Refactored Feature Model	FAMA Feature Model
 <p>A v B B v C</p>	 <p>A1 <math>\Rightarrow</math> A B1 <math>\Rightarrow</math> B B2 <math>\Rightarrow</math> B C2 <math>\Rightarrow</math> C</p>	<pre>%Relationships root: [A] [B] [C] Subtree; OR1: [1,2]{A1 B1}; OR2: [1,2]{B2 C2}; Subtree: OR1 OR2;  %Constraints A1 REQUIRES A; B1 REQUIRES B; B2 REQUIRES B; C2 REQUIRES C;</pre>

Table 5.2.: Exporting a Feature Model into the FAMA File Format

## 5.5. Summary

In this chapter, we explained how we integrated our approach from the previous chapter into the FEATUREIDE framework. Our contribution includes an equivalent refactoring of feature models with complex constraints to feature models with only simple constraints. Based on this implementation, we also extended FEATUREIDE with an exporter for the basic FAMA file format as a real application scenario. Before a feature model can be exported to FAMA, all complex constraints must be eliminated. In the next chapter, we use our implementation to empirically evaluate the practical significance of our proposed refactoring algorithm.



# 6 Evaluation

Given our implemented tool support for FEATUREIDE from the previous chapter, it is now possible to eliminate complex constraints of a feature model, while also preserving its program variants. In Chapter 4, we introduced three strategies to accomplish such a refactoring. We can either use *conjunctive normal form*, *negation normal form*, or an optimized combination of both. Furthermore, we pointed out the importance of identifying pseudo-complex constraints, i.e., cross-tree constraints that can be directly refactored into simple constraints, resulting in less additional abstract features.

A number of questions arise. What is the portion of simple, pseudo-complex, and strict-complex constraints in real-world feature models? How *complicated* are typical strict-complex constraints? What can we say about the scalability of our approach, i.e., do some feature models become disproportionately difficult to analyze based on their increase in size? What is the performance per strategy, i.e., speed and memory consumption?

In Section 6.1, we introduce the goals and the methodology for this evaluation. Next, in Section 6.2, we present and discuss our results. Finally, in Section 6.3, we discuss threats to the validity of our evaluation.

## 6.1. Methodology

As a reminder, we recall the remaining two *research questions* we formulated at the beginning of this thesis.

**RQ3** *To what extent are complex constraints used in real-world feature models?*

**RQ4** *What are the costs of eliminating complex constraints?*

For this chapter, we refine those two research questions to become more specific. To get more insight of whether our approach works in practice, the following research questions need to be answered by empirical evaluation.

**RQ3.1** *To what extent are simple, pseudo-complex, and strict-complex constraints used in real-world feature models?*

**RQ3.2** *What is the typical size of strict-complex constraints?*

**RQ4.1** *How efficient is our algorithm in the preprocessing and elimination of complex constraints, regarding speed and memory consumption?*

**RQ4.2** *Does the size of the refactored models increase significantly?*

Answering *RQ3.1* gives us a tendency towards the role of strict-complex constraints in the real world. It is yet unclear to what extend strict-complex constraints are used in industrial cases.

A follow-up question is *RQ3.2*. Strict-complex constraints have different lengths and, hence can therefore be considered differently complicated. Based on the work of [von Rhein et al. \(2015\)](#), who also needed to take the difficulty of a propositional formula into account, we use the counted number of literals occurring in a strict-complex constraint as a measurement for its complication. To avoid bias, we first convert a constraint into conjunctive normal form. There is also a connection to SAT problems, that we discuss in a later stage.

For *RQ4.1* and *RQ4.2*, we attempt to additionally use generated feature models of different sizes in terms of features (50, 100, 200, 500, 1,000, 2,000, 5,000, and 10,000) and constraints (5, 10, 20, 50, 100, 200, 500, and 1,000). They are provided in the GUIDSL format (cf. Section 2.2.2), and have been used as a benchmark before ([Thüm et al., 2009](#)). Those generated feature models do not represent industrial cases, but can reveal information about efficiency and memory consumption of our algorithm. We refer to Appendix A for the results.

In line with our notation, the following symbols are used in this chapter for number of features and number of (classified) cross-tree constraints, respectively.

Symbol	Description
$ N $	The total number of features
$ \Phi $	The total number of cross-tree constraints
	$ \Phi  =  \Phi _{simp} +  \Phi _{pseudo} +  \Phi _{strict}$
$ \Phi _{simp}$	The number of simple constraints
$ \Phi _{pseudo}$	The number of pseudo-complex constraints
$ \Phi _{strict}$	The number of strict-complex constraints

Table 6.1.: Notation Used for Evaluation

## Experimental Subjects

We choose a representative sample set comprising 17 feature models. A complete evaluation of including the generated feature models can be found in Appendix A. The evaluation uses five different evolutions of the Automotive feature model from our industrial partners, three large feature models imported from DIMACS file format (Linux kernel, eCos, and part of the FreeBSD kernel), also used by [She et al. \(2011\)](#), and several smaller feature models from the repositories of S.P.L.O.T. and FEATUREIDE with at least 50 features and at least one cross-tree constraint.

Table 6.2 gives an overview of the representative feature models we chose. The first column shows the feature model name. The second and third column state the number of features and cross-tree constraints. We also present the cross-tree constraint ratio, i.e., the relative number of features participating in one or more constraints to the total number of features of a feature model. The last column presents the number of clauses in conjunctive normal form over all cross-tree constraints for each feature model. The feature models consist of up to 18,616 features and up to 80,715 cross-tree constraints.



Model name	$ N $	$ \Phi $	CTCR <sup>1</sup> (%)	#Clauses in CNF
Automotive 2.4 <sup>2</sup>	18,616	1,369	8	1,823
Automotive 2.3 <sup>2</sup>	18,434	1,300	7	1,676
Automotive 2.2 <sup>2</sup>	17,742	914	6	1,050
Automotive 2.1 <sup>2</sup>	14,010	666	6	788
Linux kernel v2.6.28.6 <sup>3</sup>	6,889	80,715	99	80,715
Automotive 1 <sup>2</sup>	2,513	2,833	51	2,833
FreeBSD kernel 8.0.0 <sup>3</sup>	1,397	14,295	93	14,295
eCos 3.0 i386p <sup>3</sup>	1,245	2,478	99	2,478
EIS-Investment <sup>4</sup>	366	192	93	192
e-Shop <sup>2</sup>	326	21	10	21
Electronic Shopping <sup>4</sup>	291	21	11	21
FMTTest <sup>4</sup>	168	46	29	46
Violet <sup>2</sup>	101	27	66	27
Billing <sup>4</sup>	88	59	66	59
BerkeleyDB <sup>2</sup>	76	20	42	20
Arcade Game <sup>4</sup>	65	34	52	34
Classic Shell <sup>4</sup>	65	11	20	11

<sup>1</sup> CTCR: cross-tree constraint ratio

<sup>2</sup> FeatureIDE examples

<sup>3</sup> [She et al. \(2011\)](#)

<sup>4</sup> S.P.L.O.T.

Table 6.2.: Representative Sample Set of Evaluated Feature Models

## Set-Up

We computed all experiments on a single machine with an Intel Core i5-4670K with 3.4Ghz, 16 GB RAM, Window 7 (64 Bit), and Java 1.8.0.05 (64 Bit). To automate the evaluation, we implemented an evaluation tool that, given a list of feature models, eliminates strict-complex constraints according to our three strategies (i.e., negation normal form, conjunctive normal for, and the combination of both), and gathers statistics about the change in number of features, constraints, elapsed time and memory consumption in different stages of the algorithm.

To prevent the heap space from running out of memory, we set the maximum heap space from 1 GB to 2 GB. Moreover, to measure the elapsed time for a computation, we use `System.nanoTime()`, which returns a specific time of the virtual machine in nanoseconds.

## 6.2. Experimental Results

In this section, we employ our approach presented in Chapter 4 to conduct an experimental evaluation on a mix of large real-word feature models differing in size and complexity (cf. Section 6.1). In Section 6.2.1, we first determine the number of simple, pseudo-complex, and complex constraints.

Moreover, we also determine the average and worst length of strict-complex constraints. In Section 6.2.2, we evaluate the efficiency of our approach in terms of elapsed time and memory consumption. Next, in Section 6.2.3, we evaluate the scalability of our approach for large feature models. After each section, we give a discussion on our results.

### 6.2.1. Constraint Classification

In this thesis, we distinguish between three classes of cross-tree constraints.

- *Simple constraints.* Every constraint that requires exactly one feature to either include or exclude exactly one other feature (cf. Section 2.1.2).
- *Pseudo-complex constraints.* Constraints that can be transformed into an equivalent conjunction of simple constraints. These can be trivially refactored (cf. Definition 4.5 on Page 53).
- *Strict-complex constraints.* All constraints that are neither simple nor pseudo-complex (cf. Definition 4.6 on Page 53).

To gain insight to what extent feature models rely on strict-complex constraints, we are interested in the fractional amount of each class used in our feature models. In Figure 6.1, we visualize the number of constraints per feature model of our representative sample set (cf. Table 6.2) for each of the three classes accordingly.

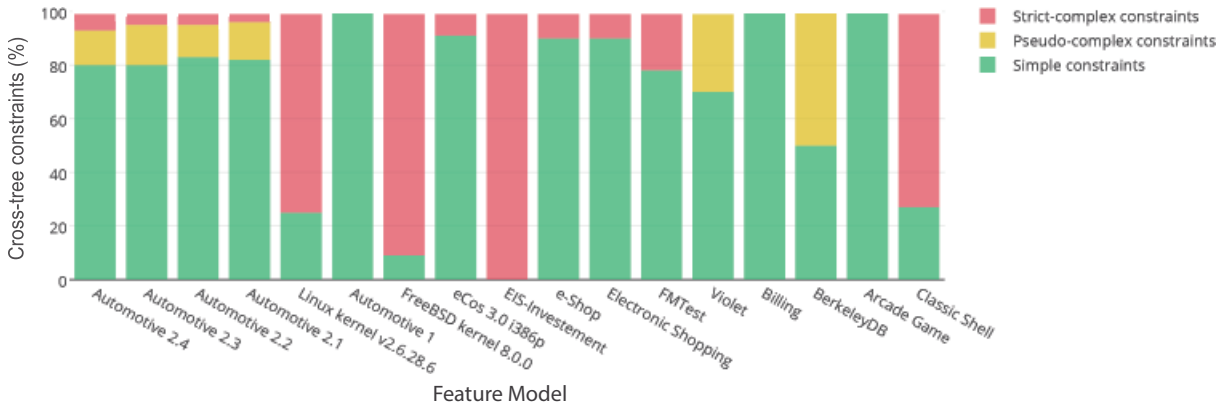


Figure 6.1.: Number of simple constraints, pseudo-complex constraints, and strict-complex constraints for each feature model.

It can be seen that 12 out of 17 feature models from our sample set utilize indeed strict-complex constraints. Exceptions are Automotive 1, Billing, and Arcade Game, who use only simple constraints. Moreover, BerkeleyDB and Violet can be trivially refactored to basic feature models by eliminating their pseudo-complex constraints. Two feature models from the S.P.L.O.T. repository, namely EIS-Investment and Classic Shell, have more strict-complex than simple constraints. The Linux kernel exaggerates the use of cross-tree constraints. Despite having only 6,889 features, Linux kernel has 80,715 cross-tree constraints, 60,269 of whom are strict-complex constraints.

Constraints can also be redundant or tautological. To decrease the number of constraints, a user may cause our implementation to automatically remove them (cf. Section 5.2). Figure 6.2 illustrates the fraction of redundant constraints we found in our evaluated feature models. Unfortunately, the larger feature models (i.e., all versions of Automotive, Linux kernel, eCos, and FreeBSD) ran into a timeout. The reason is that identifying redundant constraints requires a SAT analysis, which can be quite costly for larger feature models. This is also one reason why removing redundant and tautological constraints is optional in our implementation. Surprisingly, only three out of the nine remaining feature models were free from redundant constraints. FMTest has even over 50% redundant constraints.

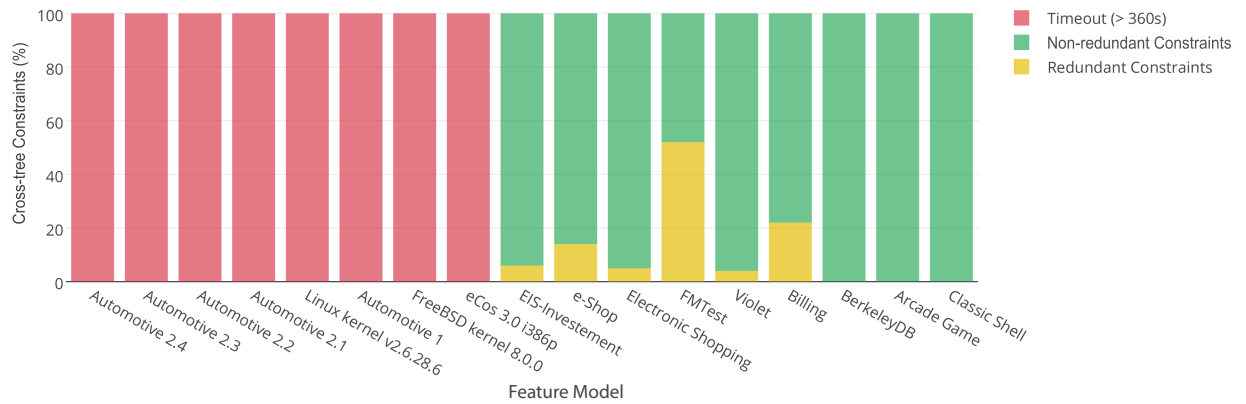


Figure 6.2.: Number of Identified Redundant Constraints

As already mentioned in the previous section, another important metric of a strict-complex constraint for our approach is its *length* (also called *width*) and clause density.

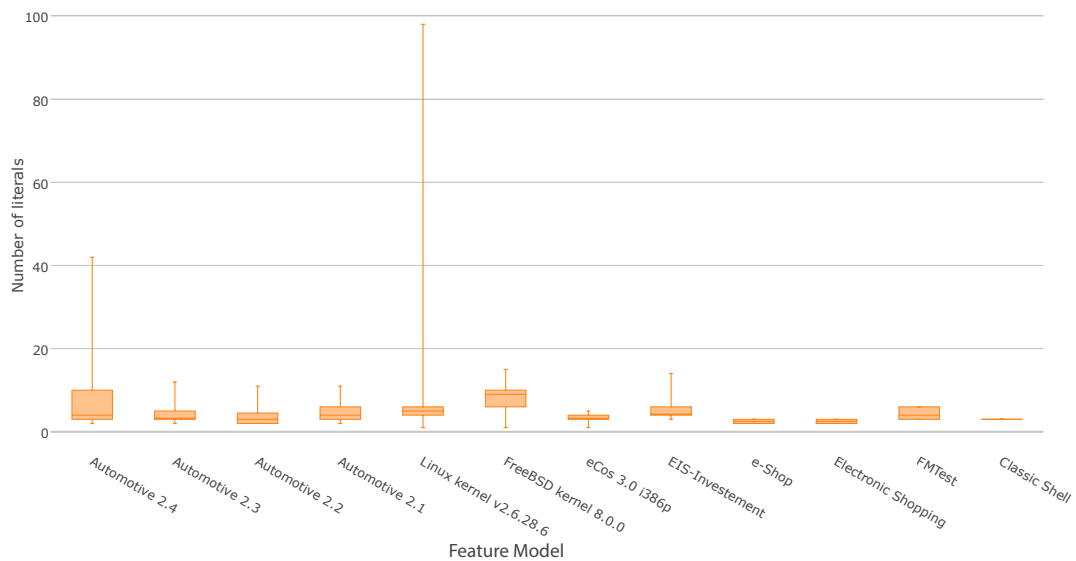


Figure 6.3.: Number in Literals of Strict-Complex Constraints for each Feature Model

The constraint width is the maximum number of literals in one clause and the clause density is the average number of literals per clause. The more literals a strict-complex constraint holds,

the bigger the corresponding abstract subtree becomes in terms of additional features and simple constraints. Figure 6.3 shows a box plot illustrating the width of strict-complex constraints for each feature model. The Linux kernel has strict-complex constraints with a width up to 98 literals. However, the average number of literals for each clause is approximately five.

### Discussion (RQ3.1 and RQ3.2)

As we can see in Figure 6.1, two-third of our evaluated feature models utilized strict-complex constraints. However, the Automotive versions, Linux kernel, eCos, and FreeBSD are the most important ones, as they represent our industrial cases. Except for Automotive 1, all of them use strict-complex constraints. Moreover, the larger feature models in the S.P.L.O.T. repositories like Classic Shell, EIS-Investment, and FMTest use a significant amount of strict-complex constraints. EIS-Investment even has almost only strict-complex constraints. Furthermore, only three feature models in total, namely Automotive 1, Billing, and Arcade Games, have merely simple constraints. There is thus reason to presume that the larger a feature model becomes (and therefore more complex), the more useful are complex constraints based on their expressive value (cf. Section 3.3). Six feature models were also using pseudo-complex constraints, mostly of the form  $A \Rightarrow a_1 \wedge \dots \wedge a_n$ , to shorten the construction of multiple simple constraints, which is yet another point for expressive usefulness of complex constraints. Feature models from S.P.L.O.T. and the imported DIMACS models (Linux kernel, eCos, and FreeBSD) do not have pseudo-complex constraints, as each constraint consist of only one clause in conjunctive normal form either by definition (S.P.L.O.T.) or by importing it into FEATUREIDE (DIMACS). Pseudo-complex constraints require at least a conjunction of two or more clauses (cf. Definition 4.5 on Page 53).

### 6.2.2. Performance Analysis

Recall that our elimination process can be divided into four major steps.

1. *Removing redundant and tautological constraints.* To decrease the size of processed constraints, we provide users the option to eliminate redundant constraints beforehand (cf. Section 5.1). Unfortunately, this step requires a SAT analysis. The question then arises as to how much time is consumed during the analysis of redundant constraints?
2. *Identifying and refactoring pseudo-complex constraints.* This step aims at decreasing the number of complex constraints and potentially the width (e.g., if the top level is a conjunction of simple and complex constraints), leaving only simple and strict-complex constraints.
3. *Preprocessing strict-complex constraints according to a given strategy.* Only strict-complex constraints are dealt with at this stage. Strict-complex constraints are either transformed into negation normal form (nnf strategy), conjunctive normal form (cnf strategy), or into the one which results in a minimum number of additional features and simple constraints (combined strategy).
4. *Generating abstract subtrees and composing them with the original feature model.* Given the set of processed strict-complex constraints from the last step. Each constraint results in an abstract subtree that is then conjoined with the original feature model.

In the following, we want to measure the average elapsed time in milliseconds to perform either of the four steps. Step 1 requires a SAT analysis and may therefore be expensive. Step 2 depends only on the number of constraints. Step 3 and step 4 depend on the number of strict-complex constraints and the chosen strategy. Hence, we measure the times for each strategy individually.<sup>1</sup>

### Overall Performance of Eliminating Complex Constraints

Before we evaluate the performance for each of the four steps individually, we depict the total time needed to eliminate complex constraints for each evaluated feature model (cf. Table 6.2) and strategy (NNF, CNF, combined) in Figure 6.4.

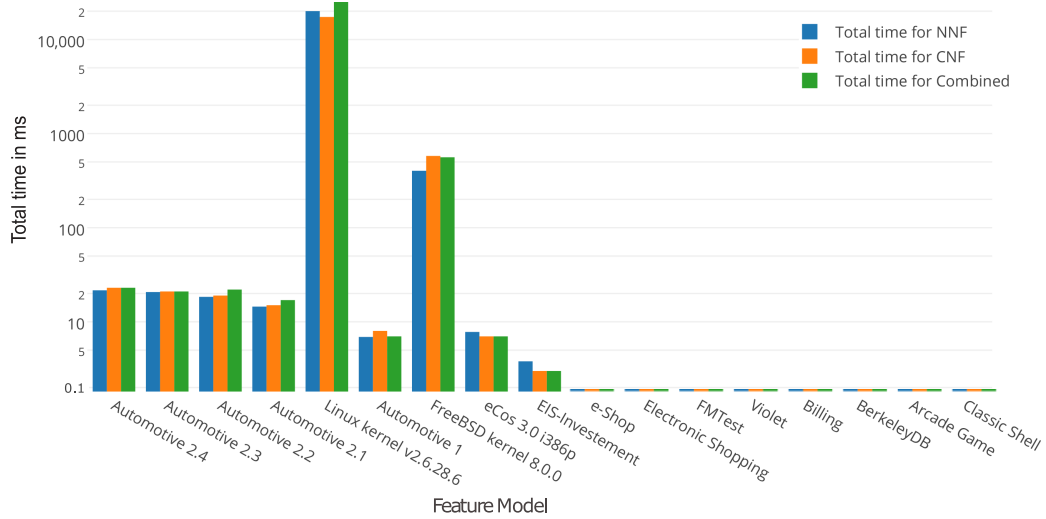


Figure 6.4.: Total Time Measured for Eliminating Complex Constraints (Incoherent Refactoring)

It can be seen that the Linux kernel takes the longest with approximately 20 seconds per strategy. Moreover, there seems to be no significant timing difference among the three strategies. This is surprising, as we assumed that the combined strategy would take considerably longer. However, except for the Linux kernel and the FreeBSD kernel, all other feature models needed less than 25 milliseconds for each strategy. The eight smallest features models were even close to 0 milliseconds for each strategy, which can be seen by the missing bars.

Recall that we provide a user option to preserve a bijection between old and new configurations by adding additional simple constraints to the abstract subtree (cf. Section 4.3.4). At Definition 4.13 on Page 61, we declared a refactoring producing a bijection between old and new configurations as *coherent*. The total time measured with a coherent refactoring for each strategy is illustrated in Figure 6.5.

<sup>1</sup>To extenuate outliers caused by fluctuation, we take 100 measurements for each step and feature model and use the average as the measured time.

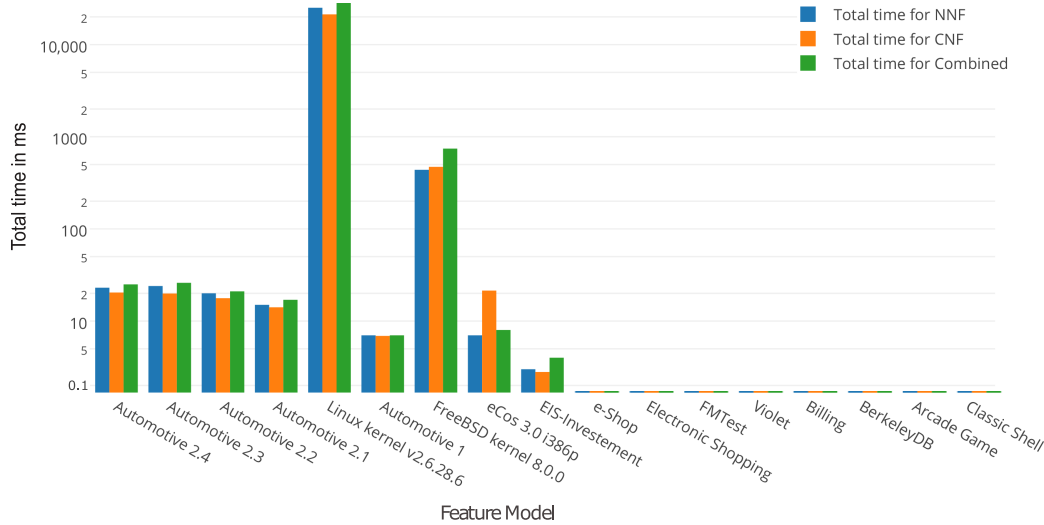


Figure 6.5.: Total Time Measured for Eliminating Complex Constraints (Coherent Refactoring)

Again, all three strategies produce times that are close together. Overall, the coherent refactoring performed slightly worse in total, but not to a noticeable degree.

### Removing Redundant and Tautological Constraints

Identifying redundant and tautological constraints is an integral part in FEATUREIDE (cf. Section 5.2). In Figure 6.2, we already depicted that six feature models of our evaluated sample set have redundant constraints (either redundant or tautological). Figure 6.6 illustrates the time needed to identify and remove them.

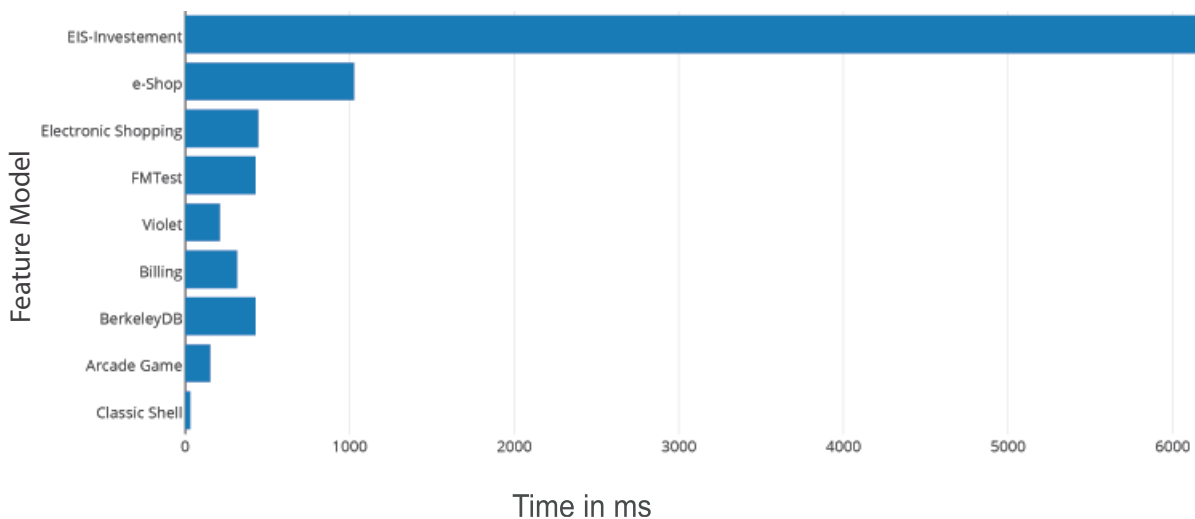


Figure 6.6.: Time Measured to Identify Redundant and Tautological Constraints.

Whereas the smaller feature models up to 366 features and 192 constraints (EIS-Investment) could be analyzed approximately within six seconds, the larger feature models (i.e., all versions of Automotive, Linux kernel, eCos, and FreeBSD) ran into an appointed timeout of 6 minutes and are therefore not depicted in Figure 6.6. Nevertheless, as this step is optional, we omit this step

in the following measurements, meaning that we use the original feature models with redundant constraints instead of the reduced versions.

### Refactoring Pseudo-Complex Constraints

In the following, we present the average times needed to identify and remove pseudo-complex constraints from feature models of our representative sample set (cf. Table 6.2). For each feature model, we measured the average time needed to run through all cross-tree constraints, converting each cross-tree constraint into conjunctive normal form, and checking if the transformed constraint is a conjunction of simple constraints. We set the time needed in this phase in relation to total time needed. As none of the three strategies is involved, we computed the average total time and average time to refactor pseudo-complex constraints over all instances we ran. The results are depicted in Figure 6.7.

Unsurprisingly, the more cross-tree constraints a feature model has, the more time is consumed during the process of identifying pseudo-complex constraints. The Linux kernel spends roughly 90% of its time in this phase. However, even with over 80,000 cross-tree constraints, the Linux kernel needs only 18 seconds in average. Unfortunately, the Linux kernel has no pseudo-complex constraints and therefore do not need to perform a split into simple constraints, so the time spent could be even worse.

As can be seen by Automotive 2.1-2.4, FreeBSD, and eCos, only a fractional amount of time compared to the Linux kernel is spent by these models. Automotive 2.3 has less constraints compared to Automotive 2.4, but slightly more pseudo-complex constraints, and therefore spends 2% more time in this phase in our experiments. All other feature models need less than one millisecond to refactor pseudo-complex constraints.

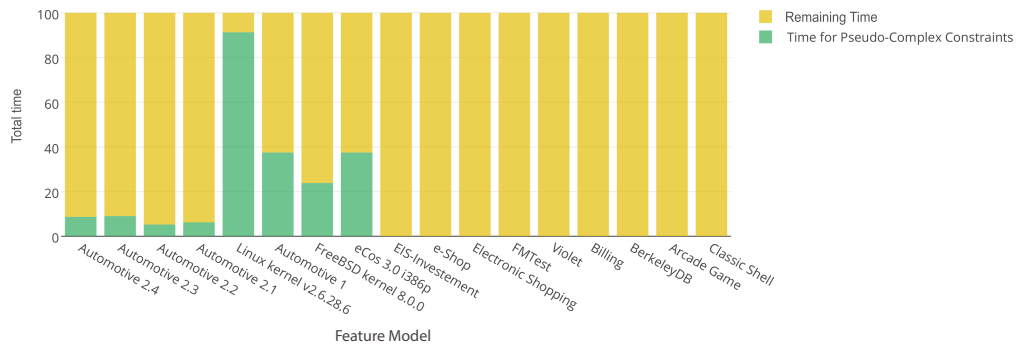


Figure 6.7.: Time Measured to Process Pseudo-Complex Constraints

### Preprocessing Strict-Complex Constraints

We now present the average times needed to transform all strict-complex constraint into either conjunctive normal form, negation normal form, or the combined option. For the sake of clarity and based on our observations of the total time measured, we again compute the average on all instances we ran, as there was no significant difference in time measured at any stage among the three strategies. We depict the results for each strategy in Figure 6.8, once again in relation to the total time measured. A more detailed overview is given in Table A.3. It can be seen that

EIS-Investments spends roughly 30% on average of the total time in this phase. The next one is FreeBSD with 20% and the Linux kernel with 8%. All other feature models do not have any strict-complex constraints (highlighted in red), or took less than one millisecond.

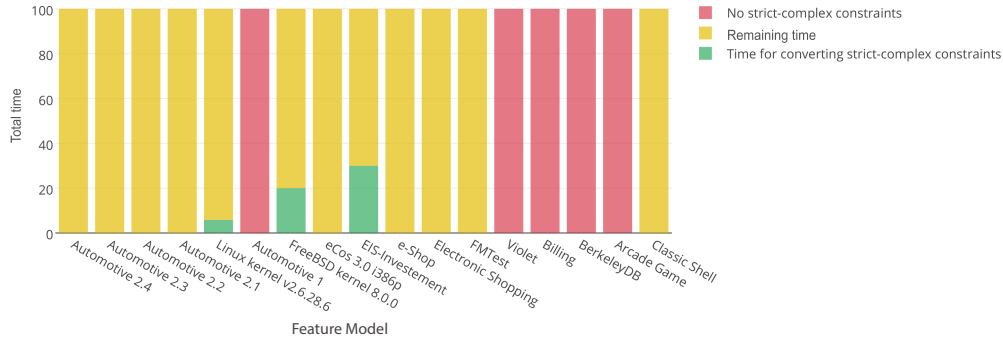


Figure 6.8.: Time Measured to Process Strict-Complex Constraints

### Generating and Adding Abstract Subtrees

Finally, we present the results of our time measurements for the construction and composing of an abstract subtree given the list of strict-complex constraints. The time needed for this not only depends on the number of strict-complex constraints, but obviously also on the constraint's width, as the number of literals and nesting of logical operators may result in a bigger abstract subtree. In Figure 6.9, we depict the results of this time measurement for each feature model and strategy, again in relation to the average total time. Surprisingly, the Linux kernel spends less than 8% of its time in this phase. Nevertheless, the absolute time is almost five times greater than the absolute time of FreeBSD.

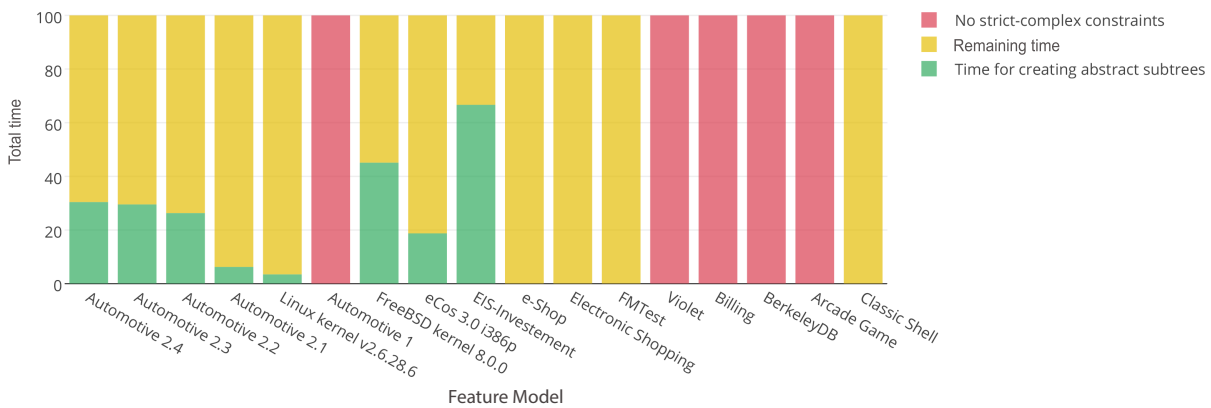


Figure 6.9.: Time Measured to Construct and Compose all Abstract Subtrees

### Memory Consumption

To discover bottlenecks resulting from poor implementation, and to find out whether a disproportionately amount of heap space in the refactoring is required, we conducted a benchmark where we monitored the heap allocation of our algorithm over time. We only used the Linux kernel as the feature model of choice, as all other feature models were processed too fast. The Linux



kernel should give a fair measure of the upper bound in terms of memory consumption to our implementation, as it takes by far the longest. We skipped the identifying of redundant and tautological constraints, as the analysis for the Linux kernel takes too long (cf. Figure 6.2).

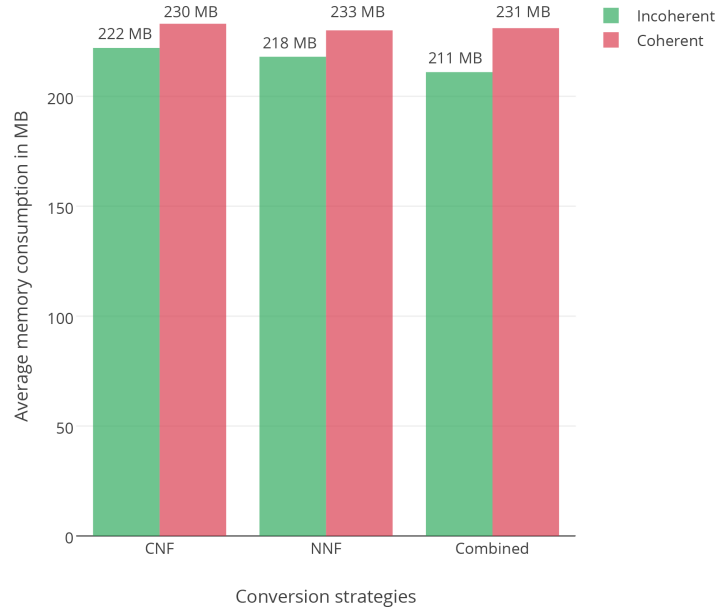


Figure 6.10.: Average Heap Allocation for the Linux Kernel for each Conversion Strategy

Figure 6.11 shows a bar diagram of the average memory consumption during either a coherent or incoherent refactoring of the Linux kernel for each strategy. It can be seen that the coherent refactoring allocates slightly more heap in the process which is not surprising considering the additional simple constraints. Figure 6.11 shows that heap memory is allocated in the beginning, when initialization of our implementation takes place, and then drops to a moderate 25% in the process of identifying pseudo-complex constraints. This process takes the longest for the Linux kernel (cf. Figure 6.8), but requires no additional heap allocation. The reason is that the Linux kernel has no pseudo-complex constraints, which would otherwise cause a spike in the heap allocation. The peak for heap allocation in instance illustrated in Figure 6.11 is at 430 MB. For runs with smaller feature models (e.g., all Automotive versions), the peak was below 100 MB. All in all, the resulting heap memory allocation for such large feature models can be considered fairly usual.

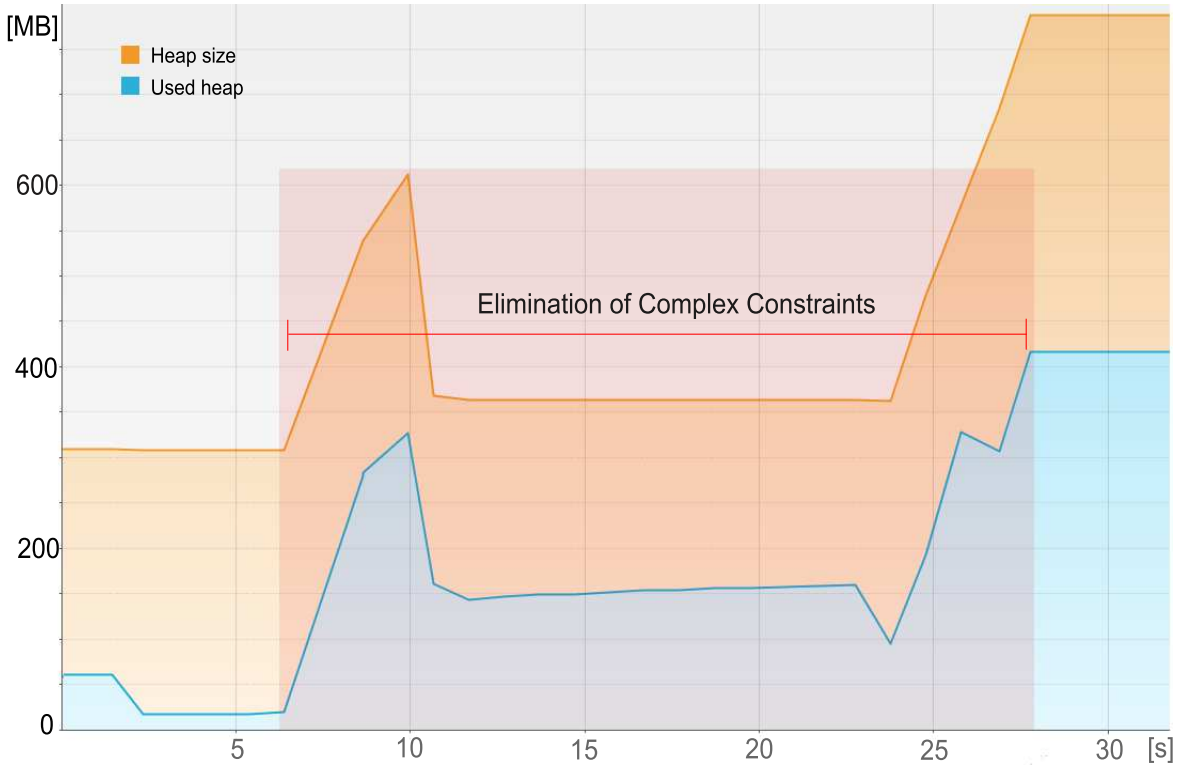


Figure 6.11.: Heap Allocation of a Refactoring of the Linux Kernel Using Conjunctive Normal Form

### Discussion (RQ4.1)

The refactoring of redundant and tautological constraints is optional in our implementation, as the required SAT analysis is quite disappointing for large feature models (cf. Figure 6.6). Moreover, a thoughtful modeling prevents the necessity of this refactoring step.

As visualized in Figure 6.8, and more elaborated in Table A.3, identifying and refactoring pseudo-complex constraints takes the longest in our implementation for larger feature models. However, skipping this refactoring may result in larger abstract subtrees. Furthermore, our implementation for refactoring pseudo-complex constraints requires general heap allocation that is also used in following computations. This allocation would then need to take place at another location, eventually resulting in equal computation time. Skipping this refactoring may therefore only benefit feature models that have no pseudo-complex constraints. Moreover, except for Linux kernel, all computations were remarkably fast (cf. Figure 6.4 and Figure 6.5). We found no significant differences when we compared the measured times of all three strategies, either with or without a coherent refactoring.

### 6.2.3. Scalability

Introducing new features and cross-tree constraints to feature models may require additional effort from automated analysis tools, potentially making them infeasible. It is therefore important to look at the increase in number of features and cross-tree constraints after performing a refactoring with our three strategies. Our evaluated results on additional features and cross-tree constraints

for each of the three strategies can be found in Table A.5. For a more convenient presentation, we only use the results from the combined strategy, which performed best, as a lower bound. We recognized that the number of additional feature and constraints is proportional to the number of strict-complex constraints. Figure 6.12 and Figure 6.13 highlight the increase in features and constraints after an incoherent and coherent refactoring. It can be seen that the Linux kernel has four times more cross-tree constraints and 60 times more features in total after an incoherent refactoring. The coherent refactoring leads to an increase of constraints by a factor of 6.5. The impact on the automotive versions were comparatively small. The resulted feature model of the FreeBSD was the worst. It had 85 times more features and 13.5 times more constraints after an coherent refactoring.

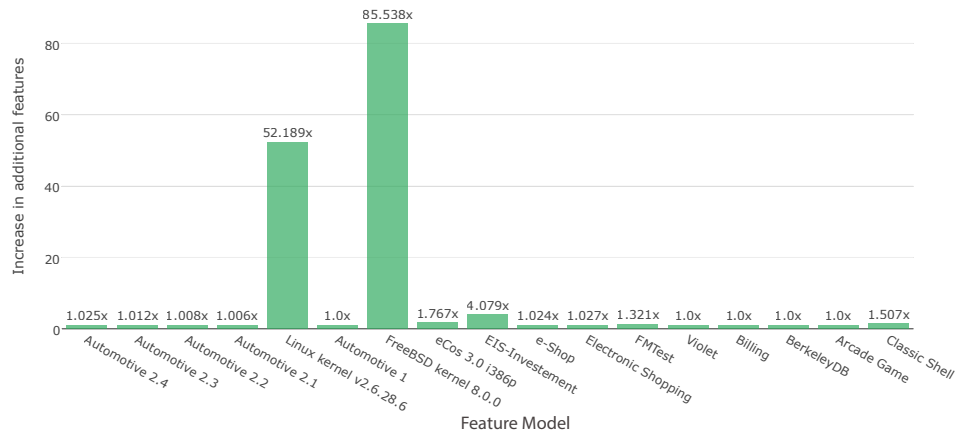


Figure 6.12.: Increase in Features after Refactoring using the Combined Method

Smaller feature models are less affected by our refactoring. However, Violet, for example, has no strict cross-tree constraints, but eight pseudo-complex constraints. The refactoring of those pseudo-complex constraints results in an increase of additional constraints by a factor of 3.3.

We can conclude that both, incoherent and coherent refactorings, may lead to a massive rise in cross-tree constraints to a point, where the analysis of a refactored feature model might be rendered infeasible alongside with other feature modeling applications (cf. Section 2.3).

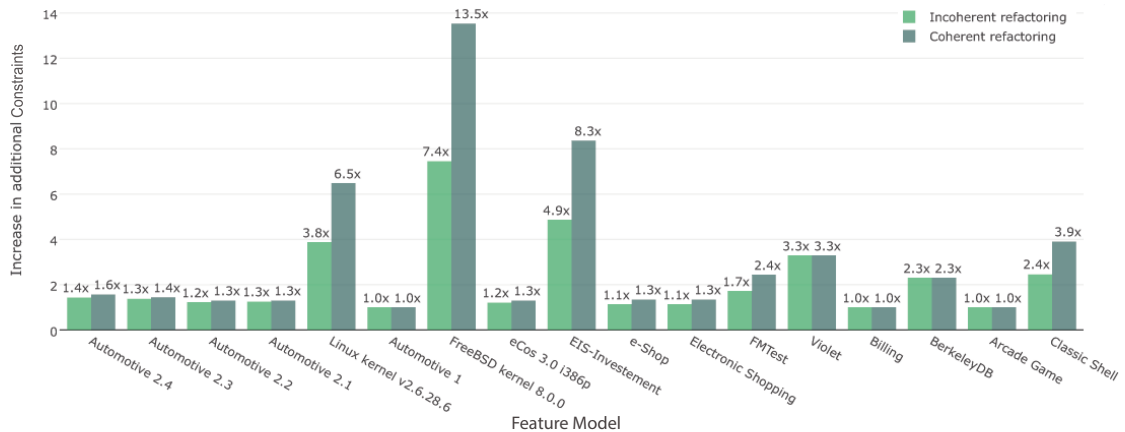


Figure 6.13.: Increase in Constraints after Refactoring using the Combined Method

### Discussion (RQ4.2)

Comparing the increase in additional features and cross-tree constraints using either *conjunctive normal form*, *negation normal form*, or the *combined strategy*, it can be seen in Table A.5 that, unsurprisingly, the combined strategy outproduces the other two strategies in terms of additional features and simple constraints (especially for Automotive 2.1-2.4). However, the results depend on the number and structure of strict-complex constraints. For instance, for feature models from the S.P.L.O.T. repositories, all strict-complex constraints are already in conjunctive normal form, which should cause all three strategies to produce the same results. However, as we load feature models in SXFM file format with FEATUREIDE, we identified an internal encoding of the cross-tree constraints that harms our strategy using negation normal form. For example, given a strict-complex constraint of the form  $\neg A \vee B \vee C \vee D$  in a SXFM-file, after loading this constraint into FEATUREIDE, the constraint is encoded into  $(\neg A \vee (B \vee (C \vee D)))$ . Using the negation normal form results in four or-groups instead of one, and thus leads to an increase of unnecessary features.

We already mentioned earlier that the expected increase in size after refactoring depends on number of strict-complex constraints. For our extreme case, the Linux kernel, with over 60,000 strict-complex constraints, this means a multiplication of cross-tree constraints by factor four and a multiplication of features by factor 60. For all other features models, except for EIS-Investment, the increase in additional constraints and features was fairly low compared to the original size. If our refactoring algorithm uses a coherent refactoring, the number of additional cross-tree constraints for the Linux kernel became twice as much compared to the incoherent approach, resulting in over 500,000 simple constraints in total. Our approach seems therefore suitable for feature models with only a low number of strict-complex-constraints. Moreover, the performance analysis revealed that the *combined strategy* in total time was surprisingly fast and should, based on our measurements, always be picked over the other two.

## General Conclusion and Implications

Overall, we can conclude that the results produced by our approach depend on the feature model's structure and number of strict-complex constraints it comprises. The Linux kernel has over ten times more cross-tree constraints than features, which indicates a miserable feature model structure and overuse of cross-tree constraints. There might be even a basic feature model or a *closer* feature model, meaning more feature model groups and less constraints, representing the same product line. However, our approach does not change the original feature model structure, but essentially only refactors cross-tree constraints.

Nevertheless, our refactoring algorithm looks promising for smaller feature models. Moreover, all computations, except for finding redundant and tautological constraints, were remarkably fast. The Linux kernel took the longest with  $\approx 20$  seconds. All other feature models from our sample set took less than 100 milliseconds in total. This indicates that there is a large timing window for further improvements and additional optimizations. For instance, a post procedure could determine and groups, produced by the negation normal form, that can be completely replaced by only one feature, as the selection of any feature of an and group with only mandatory features requires the selection of each feature of this group. This reduces the number of abstract features and should

not take longer than the general construction of abstract subtrees.

### 6.3. Threats to Validity

In the previous section, we measured the performance and scalability of our elimination algorithm for complex constraints. However, as for all experimental evaluations, certain threats may render our results less plausible. In the following, we address the potential threats harming our evaluation's validity.

The sample size for our evaluation with real-world feature models is small, as free large feature models are rare to find. We also incorporate artificial generated feature models in eight different sizes, which may lack the structure and complexity of real-world feature models. In addition, feature models from S.P.L.O.T. and the imported DIMACS models (Linux kernel, eCos, and FreeBSD) are limited in their diversity of complex constraints, as each constraint consists of only one clause in conjunctive normal form either by definition (S.P.L.O.T.) or by importing it into FEATUREIDE (DIMACS). There is thus reason to presume that the elimination of complex constraints produces different results when using a different sample set. However, the generated feature models were not randomly constructed but are sympathized with the complexity of real-world feature models (i.e., number of different decompositions and length of constraints). Moreover, the Automotive feature models represent large and real-world feature models that are not limited in their internal structure. In addition, we could show that there may exist feature models that cannot be tamed when refactored. As there are no real guidelines for modelers, real-world feature models can quite differ in their structure. In addition, most of our evaluated feature models are used in other evaluations as well.

Moreover, the feature models we use to answer the research questions are biased towards specific properties. All feature models are contradiction-free. In addition, the feature models from S.P.L.O.T. and DIMACS have no pseudo-complex constraints. The generated feature models are also biased. For every 10th feature, exactly one constraint is added. However, the internal implementation of our approach performs a simple refactoring. A contradicting feature model would result in a likewise contradicting feature model. The missing pseudo-complex constraints do not harm our evaluation either, as this step is only an optimization. A refactoring without identifying pseudo-complex constraints works as intended. The generated feature models represent more reliable use cases, where the majority of a feature model is modeled in its structure instead of its cross-tree constraints.

Furthermore, there could also be bugs and memory leaks in our implementation, falsifying our evaluated results. Nevertheless, we monitored memory consumption and CPU usage. The computations for identifying pseudo-complex constraints and refactoring strict-complex constraints were faster than expected. We additionally wrote several unit tests to ensure the correctness of checks for simple constraints, pseudo-complex constraints, and strict-complex constraints, and to ensure that we indeed perform a refactoring.

We rely on FEATUREIDE to load feature models from disk. This is especially interesting in the case of the SXFM format, as cross-tree constraints, when converted into the format of FEATUREIDE, are nested according to disjunction ( $\vee$ ). Our negation normal form then produces

larger abstract subtrees than necessary. Other implementations may therefore produce better results. However, we already concluded that using the *combined strategy* seems to be the overall solution and would, in this case, be equal to using the conjunctive normal form.

## 6.4. Summary

In Chapter 4, we proposed an algorithm for refactoring feature models with complex constraints to feature models with only simple constraints. In Chapter 5, we implemented our algorithm in FEATUREIDE to evaluate its practical significance. Our evaluation reveals that a higher number of strict-complex constraints may result in an enormous increase in size and number of simple constraints through our refactoring algorithm, eventually leading to a massive increase in effort for SAT-solvers and other feature modeling applications. For smaller feature models, the results were as expected. In addition, when using feature models with more than 1,000 strict-complex constraints, the total time needed increases significantly. Anyhow, the performance, except for identifying redundant and tautological constraints, was very fast, even for very large feature models.

# 7 Related Work

In the following, we discuss work that relate to the research on complex constraints and our refactoring solution to eliminate them.

## Expressive Power and Formal Semantics

The question of expressive power of feature models has been discussed before (Schobbens et al., 2007; Gil et al., 2010). We evaluated and highlighted that complex constraints add expressive value to feature modeling languages.

Schobbens et al. (2007) surveyed 12 feature modeling languages in total and, based upon them, proposed a general formal semantics capturing the most common feature model extensions. All surveyed languages use only simple constraints. However, their semantics includes directed acyclic graphs alongside with trees to overcome the limitation in expressiveness, whereas our proposed formal semantics uses only trees. Nevertheless, trees are more common in feature modeling, thus our proposed semantics seems easier to adopt in general and is much closer to the semantics of basic feature models. While they mention the decreased expressiveness of some languages, they do not explicitly discuss complex constraints. This thesis extends their prior research and highlights the differences in expressiveness with real numbers between basic feature models, feature models with complex constraints, and the usefulness of abstract features (Thüm et al., 2011). We also adopted and extended their survey with modern feature modeling languages that are richer in syntax and semantics.

## Eliminating Cross-Tree Constraints

We formalized an expressive complete sub-language to give a sound refactoring for eliminating complex constraints.

One attempt to eliminate cross-tree constraints is proposed by van den Broek and Galvao Lourenco da Silva (2009). They discuss the elimination of simple constraints by transforming a feature model to a *generalized feature tree*. A generalized feature tree allows features to occur multiple times in different places. This is obviously a greater restriction than our relaxation on the usage of abstract features. Moreover, they do not consider complex constraints. Cross-tree constraints are an integral and accepted part in feature modeling, as it can be seen in our survey that there is no prominent language discarding cross-tree constraints in general. Hence, we chose a different strategy and focused only on the elimination of complex constraints.

A different solution is proposed by Gil et al. (2010). There they prove that textual constraints can be eliminated by introducing a new set of features. Unfortunately, they restrict their approach to only textual constraints over two features. The handling of complex constraints is therefore questionable.

## Complex Constraints in the Literature

We surveyed 17 different feature modeling languages and identified nine of them only using simple constraints (Kang et al., 1990; Griss et al., 1998; Kang et al., 1998; Czarnecki and Eisenecker, 2000; Gulp et al., 2001; Riebisch et al., 2002; Eriksson et al., 2005; Benavides et al., 2005; Van Deursen and Klint, 2002).

Furthermore, we reviewed and discussed 26 publications on feature model applications and tools. Besides the assimilation of complex constraints, we particularly highlighted the absence on integration and discussion of complex constraints in five different areas, namely *automated analysis of feature models* (Segura et al., 2012), *synthesis of feature models* (She et al., 2011; Al-Msie 'deen et al., 2014; Haslinger et al., 2013; Lopez-Herrejon et al., 2015; Linsbauer et al., 2014), *generation of feature models as test data* (Guo et al., 2011; Segura et al., 2014), *product-line testing and analysis* (Shi et al., 2012; Ensan et al., 2012), and *optimal feature selection* (Benavides et al., 2005; White et al., 2009; Guo et al., 2011; White et al., 2014).

She et al. (2014) propose an algorithm to reverse engineer a feature model from an arbitrary propositional formula in either conjunctive normal form or disjunctive normal form. Their focus is on heuristically and automatically extracting a maximum number of groups as well as requires and excludes constraints. They acknowledge that a propositional rest may remain, depending on the complexity of the initial formula. Our evaluation on publicly available feature models complements their work and may reveal, based on the fraction of strict-complex constraints, which feature models tend to have a propositional rest and how large it is.

Mendonca et al. (2009) focus on the evaluation of SAT-based analysis for realistic feature models. They conclude that realistic feature models have a mix of binary and ternary clauses in conjunctive normal form. Our evaluation revealed that strict-complex constraints of larger feature models tend to have clauses with up to eight literals per clause in average (Automotive 2.4). Nevertheless, as they pointed out, it can be seen in our evaluation that realistic feature models indeed rely on complex constraints.

Berger et al. (2010) compare two variability modeling languages, CDL and KCONFIG, which are used in the operating system domain. They conduct an empirical study on the Linux kernel and provide evidence that real-world large-scale models need advanced concepts, compared to the concepts that FODA offers, to implement sufficient variability. They use cross-tree constraints beyond simple and complex constraints, as their constraints can have non-boolean properties (e.g., integers or strings). We think that the need for more complex constraints (e.g., non-boolean types, first-order logic) should be investigated in the future to complement our work.



# 8 Conclusion

Feature modeling is used in software product-line engineering to compactly express commonality and variability of a family of related software programs. Since feature models were first introduced in 1990, various notational extensions have been proposed by research to overcome the lack of expressiveness and allow a more precise representation of product lines. Using propositional calculus for cross-tree constraints, namely complex constraints, enable full expressiveness. In contrast, basic feature models know only two types of constraints between two features, called simple constraints. First, requires constraints can be used to always include a feature if another feature is selected. Second, excludes constraints are used when two features cannot be part of the same product. Surprisingly, basic feature models, despite their lack in expressiveness, prevail in many publications to today. This makes it worth to question the necessity of complex constraints and to think about a potential refactoring from complex to simple constraints.

We surveyed several textual and graphical feature modeling notations, and also got insight of the use of complex constraints in modern publications. Even though recent notations become more expressive through the use of group cardinalities or complex constraints, we highlighted that many publications on novel methods do not adapt accordingly and, hence, are tailored towards a small group of feature modeling languages. Moreover, practitioners working with complex constraints may not utilize these methods, resulting in a not intended limitation on both sides. Observing this problem, we dedicated ourselves in this thesis to fill this gap by contributing a refactoring from arbitrary feature models to basic feature models.

We emphasized the difference in expressiveness between feature models using complex constraints and feature models using only simple constraints. Unfortunately, this difference prevents a direct refactoring between both feature models. To overcome this limitation, we used the concept of abstract features, for which we proved complete expressiveness. Abstract features allow us to distinguish between program variants and configurations. Corollary, they allow us to add new features to a feature model without changing its product line.

We propose a sound refactoring algorithm. We can devise an equivalent feature model structure, consisting of only abstract features and simple constraints, using the negation normal form of a cross-tree constraint. This structure is eventually conjoined with the original feature model. The result is an equivalent basic feature model with abstract features. However, the practical significance needed to be evaluated empirically, for which we implemented our refactoring algorithm in the open-source Eclipse plug-in FEATUREIDE.

While our evaluation highlighted that feature models tend to have a significant amount of strict-complex constraints, and some feature models could be refactored sufficiently, the refactoring was devastating for larger feature models. The number of features of the Linux kernel feature model grew after refactoring from initially 6,889 features to 359,535 features, which is an increase by a factor of 52. Only the FreeBSD kernel was worse with an increase by a factor of 85, eventually consisting of almost 120,000 features.

Our main insight is that complex constraints are needed in the modeling process to increase applicability and expressiveness. First, complex constraints are already used by practitioners as our survey and evaluation revealed. Second, our algorithm illustrated the struggle to represent some product lines with only simple constraints. These refactored feature models would most definitely result in infeasibility for the majority of our reviewed methods that deal only with simple constraints.

# 9 Future Work

This thesis gave only a gentle entry into the world of complex constraints in feature modeling. We want to address the following concerns in future work.

## Improving our Algorithm through Partial Boolean Simplification

In Section 6.2.1, we measured the length of strict-complex constraints, and identified a correlation between number of strict-complex constraints, average length, and increase in size of the refactored feature model. Our refactoring algorithm can probably be improved and, thus, produce smaller feature models by reducing the length of strict-complex constraints.

In Section 3.2, we described how to transform a feature model into a propositional formula. The idea is to use this formula and then to logically minimize only cross-tree constraints. Logical minimization has the goal to eliminate as much literals as possible and, hence, to decrease the size of a propositional formula. Each complex constraint can be transformed into a *two-level-logic-minimization* problem Coudert and Sasao (2002). *Two levels* refer to the structure of the propositional formula, which is mostly in disjunctive normal form (first level is a disjunction of clauses ( $\vee$ ) and second level a conjunction of literals ( $\wedge$ )). The problem can then be solved with a minimization algorithm. Prominent examples are QUINE-MCCLUSCY- and the ESPRESSO-algorithm. The problem of logically minimizing cross-tree constraints while performing no change to the rest of the feature model has already been tackled by von Rhein et al. (2015) .

## Evaluating the Impact on Feature Model Applications

In Section 2.3, we highlighted that many publications propose methods without taking complex constraints into considerations. Some of them are highly cited and we assume they might be adopted more often than alternative options already integrating complex constraints.

However, it is yet unclear how to extend those methods so they can be used with complex constraints. For instance, methods utilizing SAT-solvers may be easier to prepare for complex constraints, as they typically transform a feature model into a propositional formula. Examples are analysis tools like BETTY (Segura et al., 2012). Other areas, as *optimal feature selection*, oftentimes try to solve the problem heuristically and propose an explicit transformation algorithm for each part of a feature model that is compliant with one of many methods in the operations research. Examples are evolutionary algorithms, for which even the creation of a structure for complex constraints allowing recombination, mutation, and selection might be difficult (Guo et al., 2011).



# A Evaluation Results

In Chapter 6, we used a small sample set of feature models to evaluate our approach. In the following, we give a more detailed overview with numbers on our results. As already mentioned in Section 6.1, we included also results on generated feature models of different sizes in terms of features (50, 100, 200, 500, 1,000, 2,000, 5,000, and 10,000) and constraints (5, 10, 20, 50, 100, 200, 500, and 1,000). Each category comprises 200 feature models. For a more convenient overview, we only present the average measurements on performance and scalability over the 200 feature models for each category.

■ Representative Sample Set .....	99
■ Statistical Properties of Evaluated Feature Models (cf. Section 6.1) .....	99
■ Time Measurements for Incoherent Refactoring (cf. Section 6.2.2) .....	100
■ Time Measurements for Coherent Refactoring (cf. Section 6.2.2) .....	101
■ Increase in additional Features and Constraints (cf. Section 6.2.3) .....	102
■ Generated Feature Models .....	103
■ Statistical Properties of Generated Feature Models .....	103
■ Time Measurements for Refactoring (Coherent and Incoherent) .....	104
■ Increase in Additional Features and Constraints .....	105

## Notation

We use the following symbols in this evaluation.

Symbol*	Description
$\Delta t_r$	Time needed to remove tautologies and redundancies (in milliseconds)
$\Delta t_p$	Time needed to refactor pseudo-complex constraints (in milliseconds)
$\Delta t_p^s$	Time needed to preprocess strict-complex constraints (in milliseconds)
$\Delta t_{AS}^s$	Time needed to construct an abstract subtree including composition with original feature model (in milliseconds)
$t_\Sigma^s$	Total time needed for our algorithm (in milliseconds) $t_\Sigma^s = \Delta t_p + \Delta t_p^s + \Delta t_{AS}^s$
$ N ^s$	The total number of features
$ \Phi ^s$	The total number of cross-tree constraints $ \Phi  =  \Phi _{simp} +  \Phi _{pseudo} +  \Phi _{strict}$
$ \Phi _{simp}$	The number of simple constraints
$ \Phi _{pseudo}$	The number of pseudo-complex constraints
$ \Phi _{strict}$	The number of strict-complex constraints

\*  $s$  is a replacement for either *cnf* (conjunctive normal form), *nnf* (negation normal form), or *comb* (combined method). For the number of features and constraints,  $s$  can also be empty (values of original feature model).

Table A.1.: Notation Used for Evaluation

The total time  $t_\Sigma^s$  omits  $\Delta t_r$ , the time needed to remove irrelevant constraints. The reason is that the required SAT-analysis is expensive for large feature models. Moreover, our algorithm has no influence on the performance of this step. We therefore decided to evaluate  $\Delta t_r$  in isolation.

Model name	$ N $	$ \Phi $	$ \Phi_{simp} $	$ \Phi_{pseud} $	$ \Phi_{strict} $	CTRC	#Redundant <sup>1</sup>	#Clauses	$\Delta$ Clause Density <sup>2</sup>	Constraint	Width
Automotive 2.4	18,616	1,369	1,102	190	77	8	*	1,823	7.0		42
Automotive 2.3	18,434	1,300	1,052	198	50	7	*	1,676	4.1		12
Automotive 2.2	17,742	914	765	113	36	6	*	1,050	3.7		11
Automotive 2.1	14010	666	550	99	17	6	*	788	4.9		11
Linux kernel v2.6.28.6	6,889	80,715	20,446	0	60,269	99	*	80,715	4.9		98
Automotive 1	2,513	2,833	2,833	0	0	51	*	2,833	0.0		0
FreeBSD kernel 8.0.0	1,397	14,295	1,310	0	12,985	93	*	14,295	8.1		15
eCos 3.0 i386p	1,245	2,478	2,265	0	213	99	*	2,478	3.5		5
EIS-Investment	366	192	1	0	191	93	7	192	4.9		14
e-Shop	326	21	19	0	2	2	10	21	2.5		3
Electronic shopping	291	21	19	0	2	12	1	21	2.5		3
FMTTest	168	46	36	0	10	29	24	46	4.3		6
Violet	101	27	19	8	0	66	1	89	0.0		0
Billing	88	59	59	0	0	66	13	59	0.0		0
BerkeleyDB	76	20	10	10	0	42	0	46	0.0		0
Arcade Game	65	34	34	0	0	52	0	34	0.0		0
Classic Shell	65	11	3	0	8	20	0	11	3.0		3

<sup>1</sup> Timeout (> 360s)<sup>2</sup> Average mean of all clauses from strict-complex constraints

Table A.2.: Statistical Properties of Evaluated Feature Models

Model name	$t_{\Sigma}^{cn,f}$	$t_{\Sigma}^{nn,f}$	$t_{\Sigma}^{comb}$	$\Delta t_p$	$\Delta t_p^{cn,f}$	$\Delta t_p^{nn,f}$	$\Delta t_p^{comb}$	$\Delta t_{AS}^{cn,f}$	$\Delta t_{AS}^{nn,f}$	$\Delta t_{AS}^{comb}$
Automotive 2.4	124	49	66	25	1	4	1	69	21	10
Automotive 2.3	172	39	103	4	0	0	2	9	20	86
Automotive 2.2	92	19	20	3	2	0	0	9	8	8
Automotive 2.1	17	18	21	2	0	0	0	6	6	7
Linux kernel v2.6.28.6	18,392	19,299	21,387	15,516	185	1,464	840	2,564	2,974	3,056
Automotive 1	10	9	9	5	0	0	0	0	0	0
FreeBSD kernel 8.0.0	759	560	1,055	226	152	35	263	349	363	420
eCos 3.0 i386p	12	10	11	5	0	0	0	3	2	2
EIS-Investment	5	5	5	0	0	0	1	2	3	2
e-Shop	0	0	1	0	0	0	0	0	0	0
Electronic Shopping	0	0	0	0	0	0	0	0	0	0
FMTTest	0	0	0	0	0	0	0	0	0	0
Violet	0	0	0	0	0	0	0	0	0	0
Billing	0	0	0	0	0	0	0	0	0	0
BerkeleyDB	0	0	0	0	0	0	0	0	0	0
Arcade Game	0	0	0	0	0	0	0	0	0	0
Classic Shell	0	0	0	0	0	0	0	0	0	0

Table A.3.: Time Measurements for Incoherent Refactoring



Model name	$t_{\Sigma}^{cnf}$	$t_{\Sigma}^{nnf}$	$t_{\Sigma}^{comb}$	$\Delta t_p$	$\Delta t_p^{cnf}$	$\Delta t_p^{nnf}$	$\Delta t_p^{comb}$	$\Delta t_{AS}^{cnf}$	$\Delta t_{AS}^{nnf}$	$\Delta t_{AS}^{comb}$
Automotive 2.4	261	68	35	29	1	2	2	202	45	12
Automotive 2.3	46	24	68	6	0	0	1	9	9	8
Automotive 2.2	21	23	22	3	0	0	0	8	9	9
Automotive 2.1	19	22	23	2	0	0	0	6	7	8
Linux kernel v2.6.28.6	26,404	23385	21,221	23,829	151	215	346	2,296	3,563	1,299
Automotive 1	8	281	7	4	0	0	0	0	0	0
FreeBSD kernel 8.0.0	512	1737	1136	134	47	31	672	299	1,455	307
eCos 3.0 i386p	9	9	9	4	0	0	0	2	2	2
EIS-Investment	4	5	6	0	0	0	1	2	3	3
e-Shop	0	0	0	0	0	0	0	0	0	0
Electronic Shopping	0	0	0	0	0	0	0	0	0	0
FMTest	1	0	0	1	0	0	0	0	0	0
Violet	0	0	0	0	0	0	0	0	0	0
Billing	0	0	0	0	0	0	0	0	0	0
BerkeleyDB	0	0	0	0	0	0	0	0	0	0
Arcade Game	0	0	0	0	0	0	0	0	0	0
Classic Shell	0	0	0	0	0	0	0	0	0	0

Table A.4.: Time Measurements for Coherent Refactoring

Model name	Incoherent						Coherent					
	$ N ^{cn,f}$	$ \Phi ^{cn,f}$	$ N ^{nn,f}$	$ \Phi ^{nn,f}$	$ N ^{comb}$	$ \Phi ^{comb}$	$ \Phi ^{cn,f}$	$ \Phi ^{nn,f}$	$ \Phi ^{comb}$	$ \Phi ^{cn,f}$	$ \Phi ^{nn,f}$	$ \Phi ^{comb}$
Automotive 2.4	19,212	2,132	19,209	1,971	19,098	1,970	2,379	2,138	2,131	2,379	2,138	2,131
Automotive 2.3	18,686	1,803	18,754	1,791	18,673	1,786	1,904	1,887	1,877	1,904	1,887	1,877
Automotive 2.2	17,896	1,125	17,950	1,125	17,892	1,119	1,199	1,203	1,191	1,199	1,203	1,191
Automotive 2.1	14,102	839	14,136	833	14,098	833	873	865	865	873	865	865
Linux kernel v2.6.28.6	359,535	312,822	359,535	312,822	359,535	312,822	523,594	523,594	523,594	523,594	523,594	523,594
Automotive 1	2,513	2,833	2,513	2,833	2,513	2,833	2,833	2,833	2,833	2,833	2,833	2,833
FreeBSD kernel 8.0.0	119,497	106,424	119,497	106,424	119,497	106,424	193,492	193,492	193,492	193,492	193,492	193,492
eCos 3.0 i386p	2,200	3,006	2,200	3,006	2,200	3,006	3,197	3,197	3,197	3,197	3,197	3,197
EIS-Investment	1,493	936	2,046	936	1,493	936	1,606	1,606	1,606	1,606	1,606	1,606
e-Shop	334	24	335	24	334	24	28	28	28	28	28	28
Electronic Shopping	299	24	300	24	299	24	28	28	28	28	28	28
FMTest	222	79	245	79	222	79	112	112	112	112	112	112
Violet	101	89	101	89	101	89	89	89	89	89	89	89
Billing	88	59	88	59	88	59	59	59	59	59	59	59
BerkeleyDB	76	46	76	46	76	46	46	46	46	46	46	46
Arcade Game	65	34	65	34	65	34	34	34	34	34	34	34
Classic Shell	98	27	106	27	98	27	43	43	43	43	43	43

Table A.5.: Increase in Features and Constraints After Refactoring

Model name	$ \Phi_{simp} $	$ \Phi_{pseud} $	$ \Phi_{strict} $	CTRC	#Clauses	Clause Density	Constraint Width
generated50	0.54	0.37	4.08	29.39	19.19	13.23	30.87
generated100	1.01	0.75	8.23	29.47	39.41	14.15	51.64
generated200	2.19	1.54	16.26	29.68	81.94	15.34	80.29
generated500	5.41	4.18	40.4	29.67	202.61	15.19	110.84
generated1000	10.68	8.46	80.86	29.67	408.91	15.45	148.81
generated2000	21.51	17.64	160.84	29.73	822.3	15.66	193.59
generated5000	56.29	45.2	398.5	29.7	2,056.09	15.88	271.03
generated10000	115.85	93.96	790.18	29.7	4,091.55	15.92	317.55

Table A.6.: Average Statistical Properties of Generated Feature Models

Model name	$t_{\Sigma}^{cnf}$	$t_{\Sigma}^{nnf}$	$t_{\Sigma}^{comb}$	$\Delta t_p$	$\Delta t_p^{cnf}$	$\Delta t_p^{nnf}$	$\Delta t_p^{comb}$	$\Delta t_{AS}^{cnf}$	$\Delta t_{AS}^{nnf}$	$\Delta t_{AS}^{comb}$	
Incoherent	generated50	0.54	0.41	0.47	0.07	0.03	0.05	0.1	0.36	0.26	0.28
	generated100	0.63	0.59	0.57	0.07	0.06	0.08	0.16	0.39	0.33	0.26
	generated200	1.02	0.98	1.31	0.15	0.13	0.15	0.55	0.55	0.5	0.43
	generated500	2.35	2.46	2.72	0.36	0.29	0.34	0.76	1.29	1.33	1.13
	generated1000	4.81	5.08	5.68	0.67	0.61	0.7	1.61	2.67	2.59	2.28
	generated2000	9.93	10.14	11.13	1.36	1.27	1.41	3.19	5.54	5.25	4.5
	generated5000	26.11	25.56	28.5	3.82	3.3	3.65	8.36	14.27	12.95	11.17
	generated10000	53.22	52.49	56.97	7.43	6.51	7.31	16.4	29.79	27.12	22.75
Coherent	generated50	0.56	0.46	0.45	0.07	0.04	0.05	0.14	0.38	0.29	0.2
	generated100	0.56	0.55	0.61	0.08	0.06	0.08	0.17	0.32	0.3	0.29
	generated200	1.2	1.09	1.21	0.17	0.15	0.17	0.4	0.69	0.56	0.45
	generated500	2.35	2.4	2.81	0.37	0.33	0.37	0.87	1.23	1.21	1.07
	generated1000	4.84	5.14	5.75	0.74	0.71	0.82	1.87	2.53	2.57	2.09
	generated2000	10.13	10.19	11.59	1.62	1.41	1.56	3.58	5.31	4.96	4.4
	generated5000	26.54	25.73	29.3	4.12	3.75	4.2	9.15	13.87	12.28	10.81
	generated10000	54.09	53.29	59.25	8.28	7.38	8.41	17.96	28.61	26.14	22.65

Table A.7.: Summary of Computed Times for Generated Feature Models

Model name	<i>Incoherent (<math>\emptyset</math>)</i>						<i>Coherent (<math>\emptyset</math>)</i>					
	$ N ^{cnf}$	$ \Phi ^{cnf}$	$ N ^{nnf}$	$ \Phi ^{nnf}$	$ N ^{comb}$	$ \Phi ^{comb}$	$ \Phi ^{cnf}$	$ \Phi ^{nnf}$	$ \Phi ^{comb}$	$ \Phi ^{cnf}$	$ \Phi ^{nnf}$	$ \Phi ^{comb}$
generated50	114.3	53.53	114.8	35.19	101.38	31.58	79.75	52.57	46.8	79.75	52.57	46.8
generated100	233.5	113.22	227.72	70.08	198.72	62.6	167.24	103.8	91.85	167.24	103.8	91.85
generated200	488.9	245.25	456.21	141.63	403.89	127.86	362.62	209.63	187.75	362.62	209.63	187.75
generated500	1,209.5	603.88	1,134.29	352.32	1,007.31	319.8	892.36	520.64	468.51	892.36	520.64	468.51
generated1000	2,443.17	1,228.92	2,276.86	709.65	2,023.35	644.38	1,814.21	1,046.85	942.92	1,814.21	1,046.85	942.92
generated2000	4,908.05	2,479.92	4,551.39	1,420.69	4,049.56	1,293.08	3,650.31	2,088.9	1,885.0	3,650.31	2,088.9	1,885.0
generated5000	12,297.52	6,232.95	11,321.03	3,527.47	10,089.18	3,214.59	9,158.48	5,176.08	4,674.97	9,158.48	5,176.08	4,674.97
generated10000	24,519.37	12,413.49	22,550.93	7,018.15	20,107.71	6,398.35	18,203.23	10,271.12	9,278.75	18,203.23	10,271.12	9,278.75

Table A.8.: Summary of Increase in Features and Constraints for generated Feature Models



# Bibliography

- Acher, M., A. Cleve, G. Perrouin, P. Heymans, C. Vanbeneden, P. Collet, and P. Lahire  
2012. On extracting feature models from product descriptions. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, Pp. 45–54. ACM. (Cited on pages 21 and 25.)
- Acher, M., P. Collet, P. Lahire, and R. B. France  
2011. Managing feature models with familiar: a demonstration of the language and its tool support. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, Pp. 91–96. ACM. (Cited on pages 17 and 19.)
- Al-Hajjaji, M., T. Thüm, J. Meinicke, M. Lochau, and G. Saake  
2014. Similarity-based prioritization in software product-line testing. In *Proceedings of the 18th International Software Product Line Conference-Volume 1*, Pp. 197–206. ACM. (Cited on pages 22, 23, and 25.)
- Al-Msie 'deen, R., M. Huchard, A.-D. Seriai, C. Urtado, and S. Vauttier  
2014. Reverse Engineering Feature Models from Software Configurations using Formal Concept Analysis. In *CLA 2014: Eleventh International Conference on Concept Lattices and Their Applications*, S. R. Karel Bertet, ed., volume 1252 of *CEUR-Workshop*, Pp. 95–106, Košice, Slovakia. Ondrej Krídlo. (Cited on pages 2, 21, 25, and 92.)
- Alturki, F. and R. Khedri  
2010. A tool for formal feature modeling based on bdds and product families algebra. In *WER*. (Cited on page 10.)
- Alves, V., R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena  
2006. Refactoring product lines. In *Proceedings of the 5th international conference on Generative programming and component engineering*, Pp. 201–210. ACM. (Cited on page 46.)
- Apel, S., D. Batory, C. Kästner, and G. Saake  
2013. *Feature-Oriented Software Product Lines*. Berlin, Heidelberg: Springer. (Cited on pages 6, 7, and 10.)
- Bak, K., Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wasowski  
2013. Clafer: unifying class and feature modeling. *Software & Systems Modeling*, Pp. 1–35. (Cited on pages 17 and 19.)
- Batory, D.  
2005. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Conference on Software Product Lines, SPLC'05*, Pp. 7–20, Berlin, Heidelberg. Springer-Verlag. (Cited on pages 1, 14, 15, 19, 20, and 36.)

- Batory, D., D. Benavides, and A. Ruiz-Cortés  
2006. Automated analysis of feature models: Challenges ahead. *Commun. ACM*, 49(12):45–47. (Cited on pages 6 and 20.)
- Bécan, G., M. Acher, B. Baudry, and S. B. Nasr  
2015. Breathing ontological knowledge into feature model synthesis: an empirical study. *Empirical Software Engineering*, Pp. 1–48. (Cited on pages 21 and 25.)
- Benavides, D., S. Segura, and A. Ruiz-Cortés  
2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615 – 636. (Cited on pages 1, 2, and 20.)
- Benavides, D., S. Segura, P. Trinidad, and A. Ruiz-Cortés  
2007. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. Pp. 129–134, Limerick, Ireland. Technical Report 2007-01, Lero. (Cited on pages 15, 19, 20, 25, and 72.)
- Benavides, D., P. Trinidad, and A. Ruiz-Cortés  
2005. Automated reasoning on feature models. In *Advanced Information Systems Engineering*, Pp. 491–503. Springer. (Cited on pages 1, 13, 19, 23, 25, and 92.)
- Berger, T., S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki  
2010. Variability modeling in the real: a perspective from the operating systems domain. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, Pp. 73–82. ACM. (Cited on page 92.)
- Boucher, Q., A. Classen, P. Faber, and P. Heymans  
2010. Introducing tvl, a text-based feature modelling language. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS’10), Linz, Austria, January*, Pp. 27–29. (Cited on pages 16 and 19.)
- Bovet, D. P. and M. Cesati  
2005. *Understanding the Linux kernel*. " O’Reilly Media, Inc.". (Cited on page 6.)
- Büning, H. K. and T. Lettmann  
1999. *Propositional logic: deduction and algorithms*, volume 48. Cambridge University Press. (Cited on pages 54, 55, 60, and 61.)
- Classen, A., Q. Boucher, and P. Heymans  
2011. A text-based approach to feature modelling: Syntax and semantics of tvl. *Science of Computer Programming*, 76(12):1130–1143. (Cited on page 8.)
- Classen, A., P. Heymans, and P.-Y. Schobbens  
2008. What’s in a feature: A requirements engineering perspective. In *Proceedings of the Theory and Practice of Software, 11th International Conference on Fundamental Approaches to Software Engineering, FASE’08/ETAPS’08*, Pp. 16–30, Berlin, Heidelberg. Springer-Verlag. (Cited on page 6.)



- Clements, P. and L. M. Northrop  
2001. *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. (Cited on page 5.)
- Coudert, O. and T. Sasao  
2002. Two-level logic minimization. In *Logic Synthesis and Verification*, Pp. 1–27. Springer. (Cited on page 95.)
- Czarnecki, K. and U. W. Eisenecker  
2000. *Generative Programming: Methods, Tools, and Applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co. (Cited on pages 1, 9, 12, 13, 19, 34, and 92.)
- Czarnecki, K. and A. Wasowski  
2007. Feature diagrams and logics: There and back again. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*, Pp. 23–34. IEEE. (Cited on page 45.)
- Das, S. K.  
2008. *Foundations of decision-making agents: logic, probability and modality*. World Scientific. (Cited on page 40.)
- Davis, M., G. Logemann, and D. Loveland  
1962. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397. (Cited on page 59.)
- Deelstra, S., M. Sinnema, and J. Bosch  
2004. Experiences in software product families: Problems and issues during product derivation. In *Software Product Lines*, Pp. 165–182. Springer. (Cited on page 23.)
- Deelstra, S., M. Sinnema, and J. Bosch  
2005. Product derivation in software product families: a case study. *Journal of Systems and Software*, 74(2):173–194. (Cited on page 23.)
- Ensan, F., E. Bagheri, and D. Gašević  
2012. Evolutionary search-based test generation for software product line feature models. In *Advanced Information Systems Engineering*, Pp. 613–628. Springer. (Cited on pages 23, 25, and 92.)
- Eriksson, M., J. Börstler, and K. Borg  
2005. The pluss approach: Domain modeling with features, use cases and use case realizations. In *Proceedings of the 9th International Conference on Software Product Lines*, SPLC’05, Pp. 33–44, Berlin, Heidelberg. Springer-Verlag. (Cited on pages 1, 13, 19, and 92.)
- Gacek, C. and M. Anastasopoulos  
2001. Implementing product line variabilities. In *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context*, SSR ’01, Pp. 109–117, New York, NY, USA. ACM. (Cited on page 6.)

- Ganter, B. and R. Wille  
2012. *Formal concept analysis: mathematical foundations*. Springer Science & Business Media. (Cited on page 21.)
- Gil, Y., S. Kremer-Davidson, and I. Maman  
2010. Sans constraints? feature diagrams vs. feature models. In *International Conference on Software Product Lines*, Pp. 271–285. Springer. (Cited on page 91.)
- Griss, M. L., J. Favaro, and M. d. Alessandro  
1998. Integrating feature modeling with the rseb. In *Proceedings of the 5th International Conference on Software Reuse, ICSR '98*, Pp. 76–, Washington, DC, USA. IEEE Computer Society. (Cited on pages 1, 11, 12, 13, 19, and 92.)
- Guo, J., J. White, G. Wang, J. Li, and Y. Wang  
2011. A genetic algorithm for optimized feature selection with resource constraints in software product lines. *J. Syst. Softw.*, 84(12):2208–2221. (Cited on pages 2, 22, 23, 25, 92, and 95.)
- Gurp, J. V., J. Bosch, and M. Svahnberg  
2001. On the notion of variability in software product lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture, WICSA '01*, Pp. 45–, Washington, DC, USA. IEEE Computer Society. (Cited on pages 1, 12, 19, and 92.)
- Harel, D. and B. Rumpe  
2000. Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff. Technical report, Jerusalem, Israel, Israel. (Cited on pages 28 and 32.)
- Haslinger, E. N., R. E. Lopez-Herrejon, and A. Egyed  
2013. On extracting feature models from sets of valid feature combinations. In *International Conference on Fundamental Approaches to Software Engineering*, Pp. 53–67. Springer. (Cited on pages 21, 25, and 92.)
- Kang, K. C., S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson  
1990. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute. (Cited on pages iii, 1, 6, 7, 10, 11, 19, 26, and 92.)
- Kang, K. C., S. Kim, J. Lee, K. Kim, G. J. Kim, and E. Shin  
1998. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168. (Cited on pages 1, 12, 19, and 92.)
- Kästner, C., T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel  
2009. FeatureIDE: A Tool Framework for Feature-Oriented Software Development. Pp. 611–614. Formal demonstration paper. (Cited on pages 1, 7, 13, 19, 20, and 25.)
- Liang, J. H., V. Ganesh, K. Czarnecki, and V. Raman  
2015. Sat-based analysis of large real-world feature models is easy. In *Proceedings of the 19th International Conference on Software Product Line*, Pp. 91–100. ACM. (Cited on page 36.)


- Linsbauer, L., R. E. Lopez-Herrejon, and A. Egyed  
2014. Feature model synthesis with genetic programming. In *International Symposium on Search Based Software Engineering*, Pp. 153–167. Springer. (Cited on pages 21, 25, and 92.)
- Lopez-Herrejon, R. E., L. Linsbauer, J. A. Galindo, J. A. Parejo, D. Benavides, S. Segura, and A. Egyed  
2015. An assessment of search-based techniques for reverse engineering feature models. *Journal of Systems and Software*, 103:353–369. (Cited on pages 21, 25, and 92.)
- Machado, L., J. Pereira, L. Garcia, and E. Figueiredo  
2014. Splconfig: Product configuration in software product line. (Cited on pages 23 and 25.)
- McMinn, P.  
2004. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156. (Cited on page 22.)
- Mendonca, M., A. Wąsowski, and K. Czarnecki  
2009. Sat-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*, Pp. 231–240. Carnegie Mellon University. (Cited on page 92.)
- Mendonça, M., M. Branco, and D. Cowan  
2009. S.P.L.O.T.: Software Product Lines Online Tools. Pp. 761–762. (Cited on pages 1, 15, 19, 20, and 25.)
- Meyer, M. H. and A. P. Lehnerd  
1997. *The power of product platforms*. Simon and Schuster. (Cited on page 9.)
- Nolt, J. E., D. A. Rohatyn, and A. Varzi  
1998. Schaum’s outline of theory and problems of logic. (Cited on page 37.)
- Perrouin, G., S. Sen, J. Klein, B. Baudry, and Y. Le Traon  
2010. Automated and scalable t-wise test case generation strategies for software product lines. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, Pp. 459–468. IEEE. (Cited on pages 22 and 25.)
- Pohl, K., G. Böckle, and F. J. v. d. Linden  
2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc. (Cited on pages 1, 5, 6, 9, and 10.)
- Riebisch, M., K. Böllert, D. Streitferdt, and I. Philippow  
2002. Extending feature diagrams with uml multiplicities. Pasadena, CA. (Cited on pages iii, 1, 12, 13, 15, 19, and 92.)
- Rosenmüller, M., N. Siegmund, T. Thüm, and G. Saake  
2011. Multi-Dimensional Variability Modeling. In *VaMoS*, Pp. 11–22, NY. ACM. (Cited on pages 16 and 19.)
- Ryssel, U., J. Ploennigs, and K. Kabitzsch  
2011. Extraction of feature models from formal contexts. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, P. 4. ACM. (Cited on page 25.)

- Schobbens, P.-Y., P. Heymans, J.-C. Trigaux, and Y. Bontemps  
2007. Generic semantics of feature diagrams. *Comput. Netw.*, 51(2):456–479. (Cited on pages 2, 3, 10, 12, 18, 23, 27, 28, 29, 37, and 91.)
- Segura, S., J. A. Galindo, D. Benavides, J. A. Parejo, and A. Ruiz-Cortés  
2012. Betty: benchmarking and testing on the automated analysis of feature models. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, Pp. 63–71. ACM. (Cited on pages 2, 21, 22, 25, 92, and 95.)
- Segura, S., J. A. Parejo, R. M. Hierons, D. Benavides, and A. Ruiz-Cortés  
2014. Automated generation of computationally hard feature models using evolutionary algorithms. *Expert Systems with Applications*, 41(8):3975–3992. (Cited on pages 2, 22, 25, and 92.)
- She, S., R. Lotufo, T. Berger, A. Włosowski, and K. Czarnecki  
2011. Reverse engineering feature models. In *Software Engineering (ICSE), 2011 33rd International Conference on*, Pp. 461–470. IEEE. (Cited on pages 21, 25, 76, 77, and 92.)
- She, S., U. Ryssel, N. Andersen, A. Wasowski, and K. Czarnecki  
2014. Efficient synthesis of feature models. *Information and Software Technology*, 56(9):1122–1143. (Cited on pages 21, 25, and 92.)
- Shi, J., M. B. Cohen, and M. B. Dwyer  
2012. Integration testing of software product lines using compositional symbolic execution. In *Fundamental Approaches to Software Engineering*, Pp. 270–284. Springer. (Cited on pages 23, 25, and 92.)
- Sun, J., H. Zhang, Y. Fang, and L. H. Wang  
2005. Formal semantics and verification for feature modeling. In *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, Pp. 303–312. IEEE. (Cited on pages 27 and 45.)
- Svahnberg, M. and J. Bosch  
2000. Issues concerning variability in software product lines. In *Software Architectures for Product Families*, Pp. 146–157. Springer. (Cited on page 6.)
- Thüm, T., S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake  
2012. Analysis strategies for software product lines. *School of Computer Science, University of Magdeburg, Tech. Rep. FIN-004-2012*. (Cited on page 22.)
- Thüm, T., D. Batory, and C. Kästner  
2009. Reasoning about edits to feature models. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, Pp. 254–264. IEEE. (Cited on pages 22, 25, 46, 56, and 76.)
- Thüm, T., C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich  
2014. Featureide: An extensible framework for feature-oriented software development. *Sci. Comput. Program.*, 79:70–85. (Cited on page 68.)
- Thüm, T., C. Kästner, S. Erdweg, and N. Siegmund  
2011. Abstract features in feature modeling. In *Proceedings of the 2011 15th International*

- Software Product Line Conference, SPLC '11*, Pp. 191–200, Washington, DC, USA. IEEE Computer Society. (Cited on pages 8 and 91.)
- Tseitin, G. S.  
1983. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, Pp. 466–483. Springer. (Cited on page 61.)
- van den Broek, P. and I. Galvao Lourenco da Silva  
2009. Analysis of feature models using generalised feature trees. (Cited on page 91.)
- Van Deursen, A. and P. Klint  
2002. Domain-specific language design requires feature descriptions. *CIT. Journal of computing and information technology*, 10(1):1–17. (Cited on pages iii, 14, 19, and 92.)
- von Rhein, A., A. Grebhahn, S. Apel, N. Siegmund, D. Beyer, and T. Berger  
2015. Presence-condition simplification in highly configurable systems. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, Pp. 178–188. IEEE Press. (Cited on pages 76 and 95.)
- White, J., B. Dougherty, and D. C. Schmidt  
2009. Selecting highly optimal architectural feature sets with filtered cartesian flattening. *Journal of Systems and Software*, 82(8):1268–1284. (Cited on pages 23, 25, and 92.)
- White, J., J. A. Galindo, T. Saxena, B. Dougherty, D. Benavides, and D. C. Schmidt  
2014. Evolving feature model configurations in software product lines. *J. Syst. Softw.*, 87:119–136. (Cited on pages 2, 23, 25, and 92.)
- Zanardini, D., E. Albert, and K. Villela  
2016. Resource–usage–aware configuration in software product lines. *Journal of Logical and Algebraic Methods in Programming*, 85(1):173–199. (Cited on pages 23 and 25.)







---

Technische Universität Carolo-Wilhelmina in Braunschweig  
Institut für Softwaretechnik und Fahrzeuginformatik

Mühlenpfordtstr. 23  
D-38106 Braunschweig