Otto-von-Guericke-University Magdeburg

School of Computer Science
Department of Technical & Operational Information Systems

Bachelor Thesis

Reasoning about Feature Model Edits

Author:

Thomas Thüm

June 20, 2008

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
Otto-von-Guericke-University Magdeburg
School of Computer Science
P.O. Box 4120, 39016 Magdeburg, Germany

Prof. Don Batory
The University of Texas at Austin
Department of Computer Sciences
2.124 Taylor Hall, Austin, Texas 78712, USA

Thüm, Thomas:

Reasoning about Feature Model Edits

Thüm, Thomas:
Reasoning about Feature Model Edits
Bachelor Thesis, Otto-von-Guericke-University Magdeburg, 2008.

Contents

List of Figures **vi**

List of Tables **vii**

List of Abbreviations **ix**

1 Introduction **1**

2 Background **3**

2.1 Feature Models 3

2.1.1 Feature Diagrams 3

2.1.2 Grammars 4

2.1.3 Propositional Formulas 4

2.2 Feature Model Refactoring 5

2.3 Feature Model Edits 6

2.4 AHEAD and FeatureIDE 7

3 Feature Model Types **9**

3.1 GUIDSL Feature Model 9

3.2 Property-based Transformations 12

3.2.1 Groups 13

3.2.2 Optional Features 13

3.2.3 Constraints 15

3.2.4 Several Groups 15

3.2.5 Directed Acyclic Graphs 16

3.2.6 Attributes 17

3.2.7 Cloning 19

3.3 Summary 21

4 Reasoning **23**

4.1 Operation sets 23

4.2 Reasoning with Propositional Formulas 24

4.3 Simplified Reasoning 28

4.4 Extension for Adding and Removing Features 29

4.5 Extension for Code-less Features 31

4.6 Summary 33

5 Evaluation **35**
5.1 Generating Feature Models 36
5.2 Performance and the Number of Features 37
5.3 Generating Edits on Feature Models 38
5.4 Performance and the Number of Edits 38
5.5 Performance and the Number of Children 40
5.6 On the Number of Children 41
5.7 Summary 41

6 Related Work **43**

7 Conclusion **45**

A Eliminate *And*-Groups without Mandatory Children **47**

B Tables on Calculation Time **51**

Bibliography **53**

List of Figures

2.1 Notations in Feature Diagrams 4

2.2 Feature Models Mapping to Equivalent Propositional Formulas 5

3.1 Meta Model for GUIDSL Feature Models 10

3.2 Elimination of Compound Features Associated with Code 12

3.3 A GUIDSL Feature Model Inconvertible into other Feature Models . . 12

3.4 Cardinalities in GUIDSL Feature Models 14

3.5 Redundant Representations with *Or* and *Alternative* [Rie03] 14

3.6 Elimination of Several Groups [CBB$^+$03] 16

3.7 A Feature Model with Multiple Parents [SHTB07] 17

3.8 Elimination of Multiple Parents 17

3.9 A Feature Model with Global Attributes 18

3.10 Elimination of Global Attributes 18

3.11 A Feature Model with a Cloneable Feature [CK05] 19

3.12 Elimination of Clonable Feature - First Variant 20

3.13 Elimination of Clonable Feature - Second Variant 20

4.1 Example for Reasoning using Propositional Formulas 26

4.2 Example were the Presented Approach is Incorrect 30

4.3 Example were the Dynamic Approach is Incorrect 31

4.4 A Worst Case Edit on a Worst Case Feature Model 32

5.1 Random Generated Feature Model 36

5.2 Calculation Time in Relation to the Number of Features 37

5.3 Calculation Time in Relation to the Number of Edits 39

5.4 Calculation Time in Relation to the Maximum Number of Children . . 40
5.5 Equivalent Feature Model to Figure 4.4 41

A.1 S is the root feature (Case 1) 47
A.2 S is optional (Case 2) 48
A.3 S is mandatory (Case 3) 48
A.4 S is contained in *Or* (or *Alternative*) (Case 4) 48

List of Tables

2.1 Mapping Feature Models to Grammars and Propositional Formulas . . 4
2.2 Categories of Feature Model Edits 6

4.1 Conversion of Feature Diagram Rules to CNF 27

5.1 Parameters used in Figure 5.2 38
5.2 Parameters used in Figure 5.3 39
5.3 Parameters used in Figure 5.4 40

B.1 Calculation Time in Relation to the Number of Features 51
B.2 Calculation Time in Relation to the Number of Edits 52
B.3 Calculation Time in Relation to the Maximum Number of Children . . 52

List of Abbreviations

AHEAD	Algebraic Hierarchical Equations for Application Design
BDD	Binary Decision Diagram
CNF	Conjunctive Normal Form
CSP	Constraint Satisfaction Problem
DAG	Directed Acyclic Graph
FM	Feature Model
SAT Solver	Satisfiability Solver

1. Introduction

Feature models are a well accepted means for expressing requirements in a domain on an abstract level. They are applied to describe variable and common properties of products in a software product line, and to derive and validate configurations of software systems. Their industrial importance is increasing rapidly [Rie03].

Software product lines are already adopted for many different domains. For instance, they are used to build automotive gasoline systems [STB$^+$04] and software for product lines of high-end televisions [Tre05] and mobile phones [vdLSR07].

A configuration is a combination of features. While a feature model identifies all valid configurations, a change to the feature model possibly adds and removes products from the software product line. Reasoning about feature model edits compares two given feature models concerning to their valid configurations.

Automated analysis of feature models focuses on properties of a feature model, e.g., if a feature model contains at least one product or how many valid configurations it describes [Man02, BTRC05, CK05, BRCTS06, Ben07]. Automated analysis only deals with feature models as statically constructs. But feature model are developed such as code and therefore they have to be edited. We propose that we need tool support for reasoning about feature model edits.

In 2005, Czarnecki et al. has introduced specializations of feature models, i.e., feature models that contain less products. They presented operations to specialize a feature model [CHE05a, CHE05b, KC05]. Operations on feature models that maintain the set of products or add new products to a software product line were presented by Alves et al. [AGM$^+$06].

The operations are only applied to a specific type of feature models. But as we will see in Chapter 3, many different types can be found in the literature. The main disadvantage of approaches using sound operations is that we cannot decide in which relation two given feature models are, if we do not have the edits that transform one into the other.

Sun et al. presented how equivalent feature models, i.e., that have all valid configurations in common, can be identified using first-order logic [SZLW05]. Janota and Kiniry used higher-order logic to identify if a feature model is a specialization of another, where both feature models have to be defined on the same set of features [JK07].

The latter approach has a strong restriction. It does not apply to edits where we add or delete features. Furthermore, we have found no empirical studies of the runtime for the given approaches using first-order and higher-order logic.

In this thesis, we present how equivalent feature models, as well as specializations and generalizations can be identified using satisfiability solvers. Therefore the input feature models does not need to be defined on the same set of features. Moreover, we analyze the runtime of our approach and show that it is practically computable even for large feature models with 1000 features.

Structure of the Thesis

The remainder of the thesis is structured as follows. Chapter 2 contains the necessary background to understand the following chapters. Feature models are introduced with different representations. In addition we familiarize the reader with refactorings, specializations and generalizations on feature models.

Chapter 3 present the feature model type that we use for our reasoning algorithm. We discuss different properties of feature model types that can be found in the literature and give transformations to the feature model type that we propose for reasoning.

We discuss the approach of operation sets in Chapter 4 in more detail. Then our approach that uses propositional formula is presented. We begin with a very simple idea and extend it so that it can be adopted to any pair of feature model of the type presented in Chapter 3.

If the presented approach scales is evaluated in Chapter 5. We analyze the runtime for random generated feature models. We experience an advantage of the feature model type selected for our approach.

Chapter 6 gives an overview on related work. Finally, we summarize our results and give a conclusion in Chapter 7.

2. Background

Software product lines bridges the gap between a fully custom-built single product and the mass production of software. By using a flexible system architecture - built out of a core, used by all members of the family and several variable components - the application engineer can develop new applications within a short time period and with less resources compared to conventional software development [SRP03].

Feature-oriented programming is an extension of the paradigm for object-oriented programming [Pre97] and it can be used to build software product lines in terms of features. *Features* are any prominent and distinctive aspects or characteristics that are visible to various stakeholders, e.g., end-users, domain experts or developers [KKL$^+$98].

A particular product line member is defined by a unique combination of features [KCH$^+$90]. A *software product line* is a set of products, whereas each product is a legal combination of features. We call a combination of features a *configuration*.

2.1 Feature Models

Feature modeling is a technique for managing commonalities and variabilities within a product line [CK05]. A *feature model* is a hierarchically organized set of features, that is used as a compact representation of all possible products.

Feature models were introduced in the Feature-Oriented Domain Analysis in 1990 [KCH$^+$90]. Since then different types of feature models were proposed. We discuss feature models more in detail in Chapter 3. In the following sections we give an overview on the most important representations of a feature model.

2.1.1 Feature Diagrams

A *feature diagram* is a graphical representation of a feature model [KCH$^+$90]. Every feature has a parent feature except for one feature that we call the *root feature*. Features without children are called *primitive* and features that have children are called *compound* [Bat05]. Semantically we want to express that whenever a feature is contained in a product, we will also find its parent in the same product.

Usually we distinguish between the three group types in that a feature is connected to its children (see Figure 2.1) [GFdA98, CE00, BLHM02, BSTRC06, CW07]. *And*-groups have mandatory (filled circle) and optional feature (empty cirle). Mandatory features are always selected when their parent is selected. The semantic of *Alternative*-groups is that whenever the parent is selected, we have to choose exactly one of its children. *Or* means that we have to choose at least one of the children.

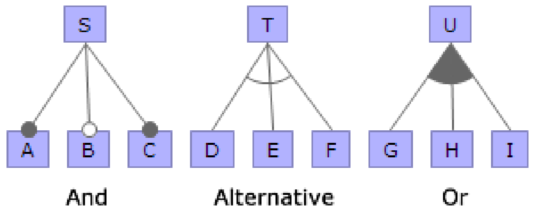


Figure 2.1: Notations in Feature Diagrams

A feature diagram may also contain so called cross-tree constraints [Ben07]. Such constraints may express that one feature requires another or that two feature mutually exclude each other. Cross-tree constraints are often drawn as dashed arrows in feature diagrams or written below the diagram.

2.1.2 Grammars

Grammars as a textual representation of feature models were introduced by Jong and Visser [BLHM02]. They mapped feature diagrams to grammars. The language defined by such a grammar belongs to the software product lines.

Every sentence equals to a product of the product line. It was proposed that a generated parser based on the grammar can identify valid configurations. In Table 2.1 we have listed grammar rules according to Figure 2.1.

Connection	Grammar	Propositional Formula
And	$S : A\ [B]\ C$	$(S \Rightarrow A \wedge C) \wedge (A \vee B \vee C \Rightarrow S)$
Alternative	$T : D \mid E \mid F$	$(T \Leftrightarrow D \vee E \vee F) \wedge \text{atmost1}(D, E, F)$
Or	$U : (G \mid H \mid I)+$	$(U \Leftrightarrow G \vee H \vee I)$

Table 2.1: Mapping Feature Models to Grammars and Propositional Formulas

2.1.3 Propositional Formulas

Propositional formulas can be seen as a logical representation of feature models. The idea is simple. For every feature we have a variable (usually with the same name) and assigning `true` to a variable means that the corresponding feature is selected. For all valid configurations the propositional formula has the boolean value `true` and for other configurations `false` [Bat05].

A feature diagram can be translated into a propositional formula using the rules given in Table 2.1. The propositional formulas for each group are connected with the logical and. Additionally we add the rule that the root feature is `true` in all products.

2.2 Feature Model Refactoring

As shown by Czarnecki et al. it is possible to transform a propositional formula to a feature model, but since there are multiple feature models that map to equivalent propositional formulas, the result is not necessarily unique [CW07]. An example for the propositional formula $A \wedge B$ is given in Figure 2.2.

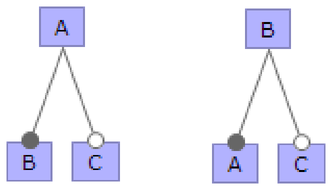


Figure 2.2: Feature Models Mapping to Equivalent Propositional Formulas

Equivalent propositional formulas identify the same software product line, since the they describe the same valid combinations of features. That means we can have different feature models that specify the same software product line. Therefore we call such feature model *equivalent* [SZLW05] and call an edit that result in an equivalent feature model a refactoring.

Definition 2.1. *A refactoring is an edit made to a feature model without changing the set of members that constitute its software product line.*

We have found a definition of refactoring in terms of software product lines in the literature that is slightly different [AGM$^+$06]. Among to their definition adding products to a software product line is also a refactoring. We are convinced that our definition is closer to the definition of code refactorings by Fowler [Fow00].

Definition 2.2. *Code refactoring is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.*

The definition of Alves et al. is inappropriate for two reasons. First, adding products seems far away from "make it easier to understand and cheaper to modify". Second, Fowler describes the two hat technique were we always should separate between refactoring and improving functionality, which we can apply for feature diagrams.

2.3 Feature Model Edits

There are more categories of edits that can be found in the literature. Specializations were introduced by Czarnecki et al. for the process of deriving configurations of a feature model [CHE05a, CHE05b, KC05]. Specializations on cardinality-based feature models were shown, that result in feature models where some products are deleted. We discuss cardinality-based feature models more in detail in Chapter 3.

Specializations are defined formal by Janota and Kiniry [JK07], but both feature models have to be defined on the same set of features. We allow feature models also to be defined on different sets of features. For instance, we can remove an optional feature to achieve a specialization.

Definition 2.3. *A specialization is an edit made to a feature model that deletes but does not add products to the software product line.*

In other situations we might need the reverse of a specialization. Alves et al. presented operations that maintain a software product line while adding new products, i.e., no products are deleted [AGM+06]. As it is the opposite to a specialization, we call those edits generalizations.

Definition 2.4. *A generalization is an edit made to a feature model that adds but does not delete products of the software product line.*

We presented definitions for refactorings, specializations, and generalizations. But since we can add products or not and we can delete products or not in an edit, there is one case missing that is shown in Table 2.2.

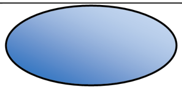
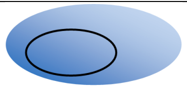
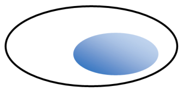
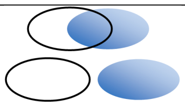
	No Products Added	Products Added
No Products Deleted	 Refactoring	 Generalization
Products Deleted	 Specialization	 Arbitrary Edit

Table 2.2: Categories of Feature Model Edits

Usually we are not interested in arbitrary edits, because there is no application for them. But we need a name for those edits that they can be identified. For the sake of completeness we also define arbitrary edits.

Definition 2.5. *An edit is called arbitrary if it is not a refactoring, specialization or generalization.*

2.4 AHEAD and FeatureIDE

AHEAD is an abbreviation for Algebraic Hierarchical Equations for Application Design. The AHEAD Tool Suite are tools for feature-oriented programming [BSR04]. The tools introduce a new language named Jak that is an extension of Java. Java is extended in order to express feature-oriented concepts.

The tools provide different steps in the feature-oriented development. Starting with tools that allow the user to create a valid configuration based on a feature model, those configurations can be used to compose Jak files and convert them to Java and compile them.

FeatureIDE is an Eclipse-based integrated development environment for feature-oriented programming [LAMS05]. It uses the AHEAD Tool Suite underneath to provide the functionality for feature-oriented programming. Furthermore it supports the process of designing feature models by a graphical editor and deriving configurations.

The reasoning algorithm presented in Chapter 4 is implemented in FeatureIDE. A view can be opened that calculates the category of all edits since the feature model is last saved, i.e., the user would know if the new feature model allows new products and if it forbids already existing ones.

3. Feature Model Types

This chapter introduces the GUIDSL Feature Model which is a feature model type used for our reasoning algorithm presented in the following chapter. We study different properties of feature models that we found in the literature and show how such feature models can be converted into a GUIDSL Feature Model.

This study is important because we want to ensure that we do not implement reasoning only for one particular feature model type, i.e., if there would be no transformation from other known feature model types, we should not use this type.

Kang et al. introduced feature models in the Feature-Oriented Domain Analysis in 1990 [KCH$^+$90]. As noticed by Schobbens et. al., afterwards, various extensions were introduced to compensate for a purported ambiguity and lack of precision and expressiveness [SHTB07]. We give references to these publications along with the properties they hold.

The remainder of this section is structured as follows. We present the GUIDSL Feature Model in Section 3.1 and specify its semantics using a meta model. Section 3.2 discusses several properties that distinguish different feature model types in the literature. We describe how feature models owning a particular property can be transformed into a GUIDSL Feature Model. Finally, we summarize our results in Section 3.3.

3.1 GUIDSL Feature Model

Batory presented a transformation between feature models and grammars [Bat05]. A tool named GUIDSL takes a grammar as input and provides a graphical user interface to create configurations. We examined allowed constructs in the grammar and analyzed the meaning in GUIDSL to build a meta model. We name the feature model type defined by this meta model *GUIDSL Feature Model*.

In the following, we explain the semantics of GUIDSL Feature Models in detail. The reason that we choose this feature model type for reasoning is that we are able to convert almost all feature model type we found in the literature into a GUIDSL Feature Model. But we found no feature model type, so that a GUIDSL Feature Model can be converted into a feature model of this type. The reason is given at the end of this section. Another advantage is recognized in Section 5.

Figure 3.1 presents the meta model for GUIDSL Feature Models. Each GUIDSL Feature Model has exactly one root feature. A root feature is always a compound feature. As introduced in Chapter 2 compound features do have children. Therefore we know that a GUIDSL Feature Model has at least two features.

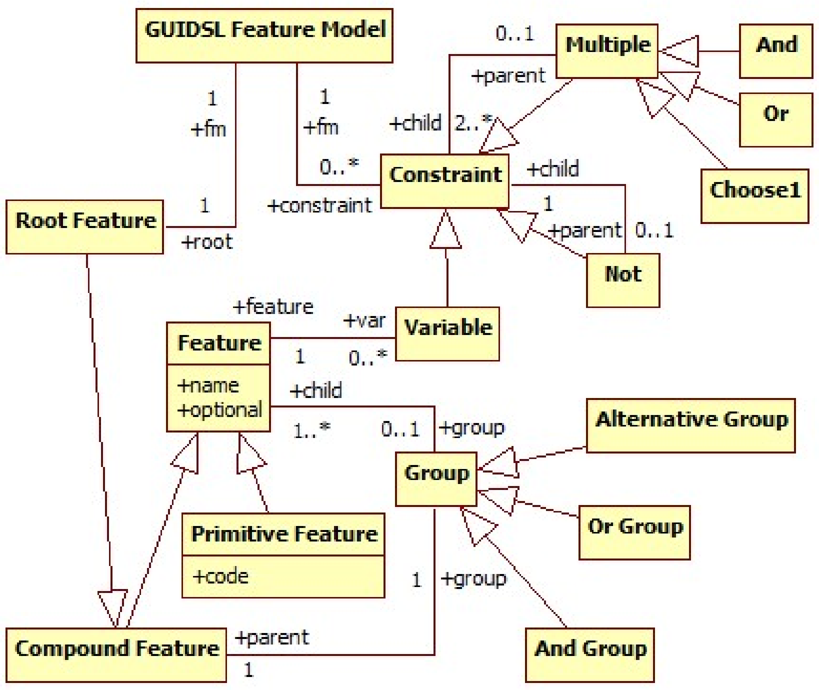


Figure 3.1: Meta Model for GUIDSL Feature Models

Compound and primitive features are features which own a unique name and a boolean value that specifies whether the feature is optional or mandatory. Each compound feature is the parent from exactly one group. A group is a *And*-, *Or*-, or *Alternative*-group.

An *Alternative*-group means that exactly one child is selected if the parent is selected. For *Or*-groups one or more children are in a product if the parent is also in the product. *And*-groups demand that all mandatory children are selected if the parent is selected. A group contains at least one child.

We give a restriction additionally to the meta model. All features that are not contained in an *And*-group are mandatory, i.e., the root and features contained in *Alternative*- or *Or*-groups cannot be optional.

We discussed the structure of features and give a detailed description of constraints. A GUIDSL Feature Model can have any number of constraints. Where we define constraints recursively.

(i) A variable X is a constraint.

(ii) If X is a constraint, then $\neg X$ is a constraint, too.

(iii) If $X_1, X_2, ..., X_n$ with $n > 1$ are constraints, then $(X_1 \wedge X_2 \wedge ... \wedge X_n)$, $(X_1 \vee X_2 \vee ... \vee X_n)$, and *choose1*$(X_1, X_2, ..., X_n)$ are constraints, too.

While *And* and *Or* are known from propositional calculus, *Choose1* means that one constraint is `true` and all others are `false`.

Apart from fact that primitive features have no children there is a second difference between primitive and compound features. Only primitive features are associated with code, because GUIDSL only permits primitive features in configurations. The reason can be found in the connection to grammars.

Grammars distinguish terminal and non-terminal symbols. The sentences of the language defined by the grammar only contain terminal symbols, since all non-terminal symbols are replaced by terminals.

We call compound features in GUIDSL Feature Models *code-less* features to express that they are not associated with code. We draw code-less features in a different color, that it can easily be seen which type is used at a specific diagram.

Hence all feature models we found in the literature do not deal with code-less features, we discuss in this section how other feature models can be transformed into feature models, where all compound features are code-less.

Schobbens et al. presented an algorithm to convert a feature model without code-less features into one where compound features are not allowed to contain code [SHTB07]. For every compound feature S we add two code-less features P and Q, where Q and S are children of P in an *And* and all children of S are moved to Q. The connection type that S had before is set to Q. See Figure 3.2 for an example. Why the *Or*-group is rendered slightly different in the right feature diagram is explained in Section 3.2.4.

The reason that we implemented reasoning on GUIDSL Feature Model is, that there is no transformation for the opposite direction. Figure 3.3 shows a simple example of a feature model with a code-less feature that has no counterpart in the other representation. The reason is, that the root in a feature model without code-less features occurs in all products and the example has no such feature. Both A and B do not occur in all products.

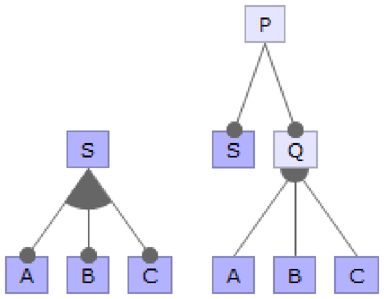


Figure 3.2: Elimination of Compound Features Associated with Code

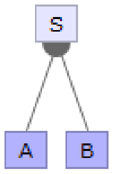


Figure 3.3: A GUIDSL Feature Model Inconvertible into other Feature Models

3.2 Property-based Transformations

This section discusses properties that we have abstracted to distinguish different feature model types that can be found in the literature. The properties do not belong to the graphical notation, i.e., how an *Or*-group is rendered at the feature diagram. We only deal with properties that have an influence on the semantic and the software product lines that can be expressed by the feature model type. We have structured our abstracted properties into seven sections.

1. Groups: Which connections are allowed between a parent and its children?
2. Optional Features: Where in the feature model are optional features allowed?
3. Constraints: What kinds of constraints are permitted?
4. Several Groups: Is it approved that a feature can have more than one group?
5. Directed Acyclic Graphs: Can a feature have more than one parent?
6. Attributes: Are constraints defined on numerical values?
7. Cloning: Is it possible to clone features in the configuration process?

We give references where particular properties occur and show how feature models with a special property can be transformed into GUIDSL Feature Models.

3.2.1 Groups

Kang et al. are known for the first feature models. The presented a feature model type that contains *And*- and *Alternative*-groups [KCH+90]. All following publications may reference to this feature model type, but they also introduce *Or*-groups. We discussed already the semantics of all three group types and hence they are all allowed in GUIDSL Feature Models there is no need for a transformation.

We found different names for *Alternative*-groups such as *One-of*-group [dJV02], *Mutex*-group [ZZM04], and *Xor*-group [GFdA98, vGBS01, CBB+03, Rie03, SRP03, BHST04, CK05, KC05, CKK06, CW07, SHTB07, WSB+08]. We disapprove to other names than *Alternative*, because *Alternative* is first published and there is no need for a new name. In particular the name *Xor*-group is chosen inauspiciously, since the logical *Xor* has a different meaning for three or more features, e.g., $1 \oplus 1 \oplus 0 \oplus 1 \equiv 1$.

Groups were first generalized with cardinalities by Riebisch et. al [RBSP02]. We assign the group cardinality $<n..m>$ to a group with k children, where $n \geq 0$, $n <= m$, and $m \leq k$. Therefore we already know that the group cardinalities $<1..1>$, $<1..k>$, and $<k..k>$ as *Alternative*-, *Or*-, and *And*-groups. Group cardinalities are widespread in literature on feature models, because several authors have adopted this idea [Rie03, SRP03, BHST04, AMS04, BSTRC05, CK05, CHE05a, CHE05b, KC05, BRCTS06, CKK06, JK07, WSB+08].

Hence we do not have such all these general $<n..m>$-groups in GUIDSL Feature Models, we need to present a transformation. Let Y be a feature that has k children $X_1, ..., X_k$ and the group cardinality $<n..m>$. The resulting feature model has only *And*-groups where all features are optional. Furthermore we add the constraint

$$\bigwedge_{\{i_1,...,i_{k-n+1}\}\subseteq\{1,...,k\}} (\neg Y \vee X_{i_1} \vee ... \vee X_{i_{k-n+1}})$$

if $n > 0$ and if $m < k$ we also add the constraint

$$\bigwedge_{\{i_1,...,i_{m+1}\}\subseteq\{1,...,k\}} (\neg X_{i_1} \vee ... \vee \neg X_{i_{m+1}}).$$

Note that GUIDSL Feature Models allow arbitrary constraints with $\wedge$, $\vee$, and $\neg$. An example for the group cardinality $<2..2>$ and a group with three children is given in Figure 3.4.

3.2.2 Optional Features

Optional features in *And*-groups are allowed in all publications and so for GUIDSL Feature Models, too. But some publications also allow features to be optional beyond *And*-groups, i.e., in *Or*- and *Alternative*-groups [CE00, vDK02, dJV02, CBB+03, ZZM04].

This property was first presented by Czarnecki and Eisenecker involved with necessary normalizations to prevent redundancies at the feature diagram. The reason is that whether one or more optional features appear in *Or*- and *Alternative*-groups are optional

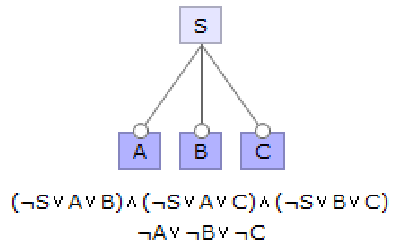


Figure 3.4: Cardinalities in GUIDSL Feature Models

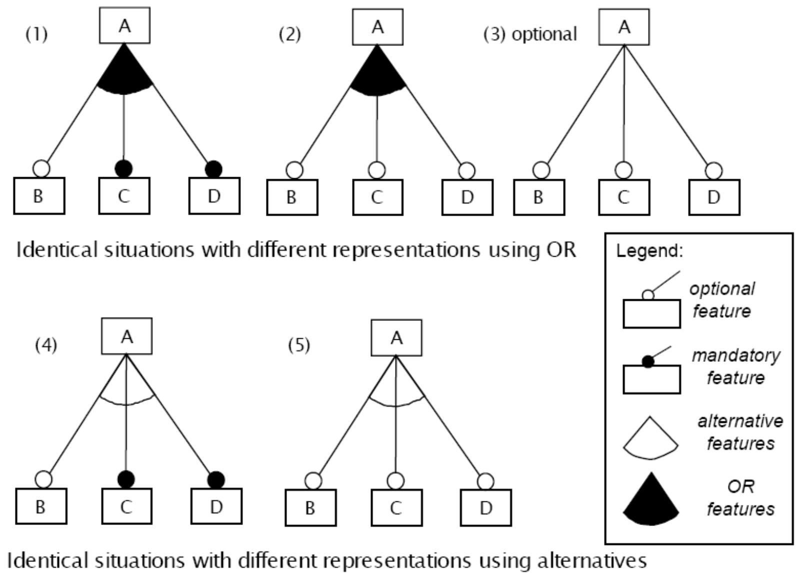

(1)
A
B
C
D
(2)
A
B
C
D
(3) optional
A
B
C
D
Identical situations with different representations using OR
Legend:
optional feature
mandatory feature
alternative features
OR features
(4)
A
B
C
D
(5)
A
B
C
D
Identical situations with different representations using alternatives

Figure 3.5: Redundant Representations with *Or* and *Alternative* [Rie03]

and which is not decisive. If there is an optional feature within such a group, then it is also allowed that the parent is selected, but none of its children. These equivalent feature diagrams were also proposed by Riebisch (see Figure 3.5).

Furthermore Riebisch presented that an *Or*-group with an optional feature is equivalent to an *And*-group where all children are optional. This is already a transformation into a GUIDSL Feature Model, because an *And*-group where all children are optional is allowed in GUIDSL Feature Models.

There are proposals that only permit the normalized feature models, i.e., either all or no optional feature occur in such groups [LAMS05, SZLW05]. Moreover all feature models that have group cardinalities are able to express the normalized versions, because *Alternative*-groups with optional features are equivalent to <0..1>-groups. Therefore all *Alternative*-groups that contain optional features can be transformed into GUIDSL Feature Models as shown in Section 3.2.1.

3.2.3 Constraints

Constraints were introduced as composition rules [KCH+90, KKL+98, AMS04, JK07] and also referenced to as additional constraints [Beu03, WLS+05, KC05, CKK06, CW07], non-grammar constraints [Bat05], logical constraints [AGM+06], and cross-tree constraints [BRCTS06, BSTRC06, TBRC+06, TBKC07, WSB+08]. We use the term constraint, since it it mostly used.

The feature model type presented by Kang et al. involves two different constraints [KCH+90]. Given two features A and B, we can express that A requires B or that A and B mutually exclude each other (A mutex B). Because A excludes B is equivalent to B excludes A, mutually is often omitted. Depends [BSTRC05, TBRC+06] and includes [BRCTS06, BSTRC06] is sometimes used instead of requires. We can also find incompatible-with for excludes [AMS04].

These both constraints are usually supported by feature models. Therefore we give not references to all these feature model types. Since A requires $B \equiv \neg A \vee B$ and A excludes $B \equiv \neg A \vee \neg B$ we know that these simple constraints can be expressed in GUIDSL Feature Models. Feature model types that do not support constraints [vGBS01, dJV02, CHE05a, CHE05b] do not have to be converted due to this property.

Aside from Batory [Bat05] also other publications allow arbitrary [CE00, RBSP02, Beu03, SRP03, CK05, Bat05, AGM+06, CKK06, CW07, TBKC07] or self-defined constraints [ZZM04]. All these constraints can be converted into GUIDSL constraints, since $\{\wedge, \vee, \neg\}$ is expressive complete as known from propositional calculus, i.e., we can express every boolean function on a set of variables.

3.2.4 Several Groups

GUIDSL Feature Models only allow one group per feature. To be more precise, for every compound feature. But we have found 14 publications with examples [CE00, dJV02, CBB+03, ZZM04, SZLW05, LAMS05, WLS+05, CHE05a, CHE05b, AGM+06, CW07] or meta-models [Beu03, BSTRC05, KC05] where more than one group type with the same parent is permitted.

Cao et al. were the first that have proposed that there are different notations of feature models [CBB+03]. They called it mixture of feature representations and presented a transformation to eliminate such occurrences (see Figure 3.6).

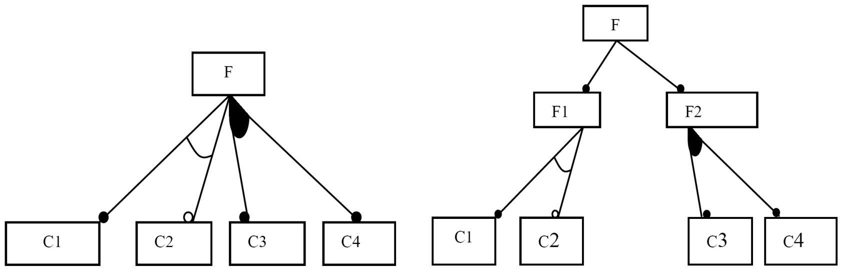


Figure 3.6: Elimination of Several Groups [CBB+03]

Note that in general, this transformation is not a refactoring, because we add two features that are contained in all products which contain F. But this is a refactoring if we assume that $F1$ and $F2$ are code-less features, i.e., they are not intended to appear in products. Since $F1$ and $F2$ are compound features and in GUIDSL Feature Models all compound features are code-less, this is the transformation of several groups into GUIDSL Feature Model.

We argued that it is no restriction for GUIDSL Feature Models to forbid several groups. But Bontemps et al. presented a formal semantic for FODA feature model, i.e., the first feature models presented by Kang et al. in 1990, that forbids several groups [BHST04]. We think this is a strong restriction since most people have interpreted FODA feature models different.

The generic semantic for feature models by Schobbens et al. is based on this previous semantic and was aimed to give a semantic to the most published and used feature models [SHTB07]. They presented a strong theory, but since it does not apply to the most of our referenced literature, we cannot use it to give transformations generically.

Whenever we deal with feature models where several types are not allowed, we draw a half circle for *Or*- and *Alternative*-groups. We recommend that people dealing with those feature models use this notation in future publications, so that readers can easily identify the feature model type. Again, this is important because the above transformation is not a refactoring, if we do not deal with code-less features.

3.2.5 Directed Acyclic Graphs

The GUIDSL Feature Model guarantees that a feature has at most one parent. But there are feature model types in the literature that also allow multiple parents [GFdA98, KKL+98, CE00, RBSP02, Beu03, BHST04, ZZM04, CW07, SHTB07], i.e., feature models are no longer trees, they are directed acyclic graphs (see Figure 3.7).

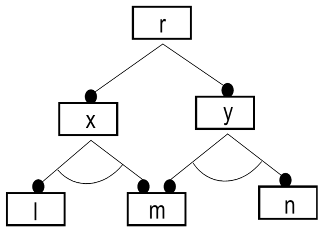


Figure 3.7: A Feature Model with Multiple Parents [SHTB07]

We presented in Section 2.1.3 that a feature model can be converted into a propositional formula. This is done by generating a propositional formula for every group, that represents exactly the same restriction. An algorithm that converts a feature model with multiple parents into a GUIDSL Feature Model, could work as follows.

If a feature model contains a feature with more than one parent, we remove one of these connections and add the according propositional formula as constraint. It can happen that a another child of this connection has no longer a parent. In this case we add this feature as an optional feature to its old parent. We repeat this procedure until all features have at most one parent.

In Figure 3.8 the result of the algorithm used for the above example is shown. Note that the primitive features might also have to be eliminated before or afterwards. For simplicity this is not done with our example.

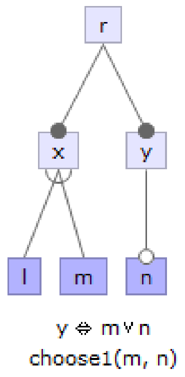


Figure 3.8: Elimination of Multiple Parents

3.2.6 Attributes

The first publications arguing that attributes (also called properties or parameters) are a useful extension to feature models were proposed in 2003 [Beu03, SRP03]. They state that any number of attributes can be associated with a feature. We give a conceived example in Figure 3.9, where the Java version of several libraries is gathered.

An *attribute* is any characteristic of a feature that can be measured. We distinguish between *basic attributes* that are directly related to a feature and *derived attributes* that are calculated from other attributes. The *domain* of an attribute is the space of possible

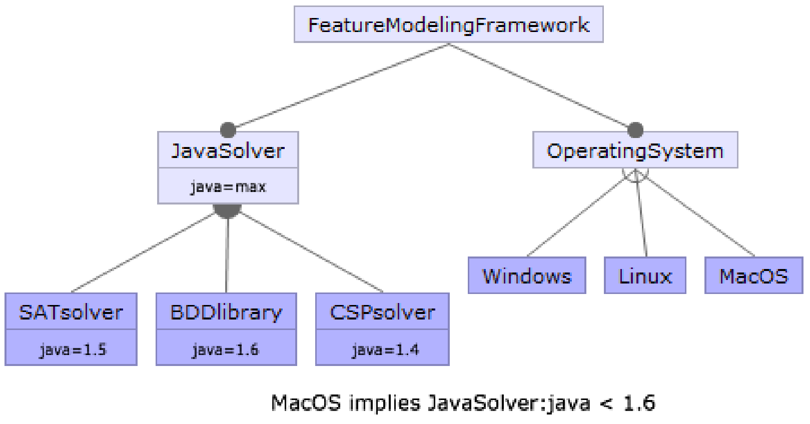


Figure 3.9: A Feature Model with Global Attributes

values of an attribute. A domain can be discrete (e.g., integer, boolean, enumeration) or continuous (e.g., real) [BTRC05].

In our example in Figure 3.9 the Java version of specific solvers are basic attributes. The Java version for `JavaSolver` is derived. It is the maximum of the values of its children. The domain of these attributes is the enumeration $\{1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6\}$. A constraint is given which only allows Java versions that are already available for Mac OS, if the feature `MacOS` is selected.

The example can be transformed into a GUIDSL Feature Model (see Figure 3.10). Therefore we have to convert all constraints defined on attributes. In this case we know that `BDDlibrary` cannot be selected if `MacOS` is selected. All other configurations defined by the feature diagram are possible. In addition, we delete all attributes from the feature model, because they have no influence on further analysis.

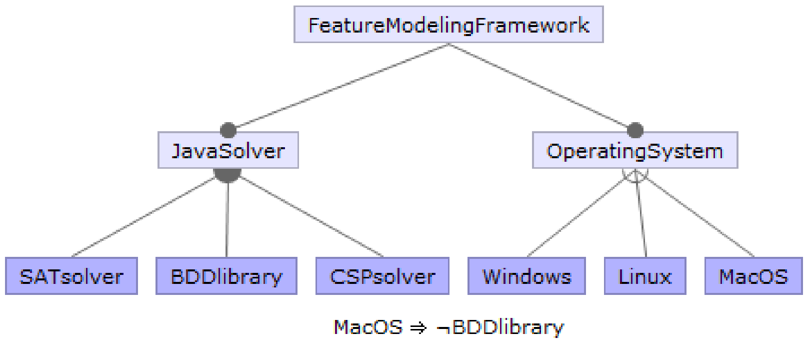


Figure 3.10: Elimination of Global Attributes

Beuche distinguishes between global and local attributes. The value of a *global* attribute is stored together with the feature inside the feature model and is considered to be part of the feature model. A *local* feature attribute value is stored in a configuration, so the value may be different in another configuration [Beu03].

All attributes in our example are global, because they are specified in the feature model. We can eliminate all global attributes as shown our example. Therefore we can also convert feature models from all publications where attributes are meant to be global [AMS04, BTRC05, BRCTS06, Ben07, JK07, WSB$^+$08].

Czarnecki et al. has published feature models that contain attributes which can be either local or global [CK05, CHE05a, CHE05b, KC05, CKK06]. It depends if the value is already set in the feature model or not. If not, only a domain is given. Usually it is not possible to convert local attributes into GUIDSL Feature Model, because these attributes are needed for the configuration process.

For example, one could extend our example in Figure 3.9 by a user defined Java version which is used to compile the feature modeling framework. We cannot eliminate the attribute-based constraint, since we do not know the compile version in the modeling process.

Note that in general there is a main gap between local attributes and automated analysis of feature models. For instance, what is the number of products defined by a feature model that contains a local attribute A with the domain integer? Should we count all possible configurations (i.e., $A = 1, A = 2, A = 3, \ldots$)?

3.2.7 Cloning

Czarnecki et al. presented a feature model type that introduces feature cardinalities [CK05, CHE05a, CHE05b, KC05, CKK06]. A feature cardinality is assigned to every feature. Optional feature have the feature cardinality [0..1], features in groups except *And*-groups and mandatory have the cardinality [1..1]. In general, [m..n] denotes that a feature can occur in a configuration between m and n times. The feature cardinality [m..$*$] means that there is no upper bound on the number of cloned features (see Figure 3.11).

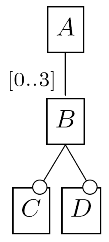


Figure 3.11: A Feature Model with a Cloneable Feature [CK05]

We present two transformations to GUIDSL Feature Models. Both variants convert the feature cardinality [m..n] into a group cardinality <m..n>. Because group cardinalities does not allow $*$, we replace $*$ in this case by the number of features in the new created group. B denotes the feature whose feature cardinality needs to be eliminated.

The first transformation creates all configurations of the subtree rooted by B (see Figure 3.12). Note that this is only possible, if there is an upper bound on the number of configurations. In general, it is not possible if we have local attributes. But since we already know that local attributes cannot be eliminated (see Section 3.2.6), we are able to use this transformation.

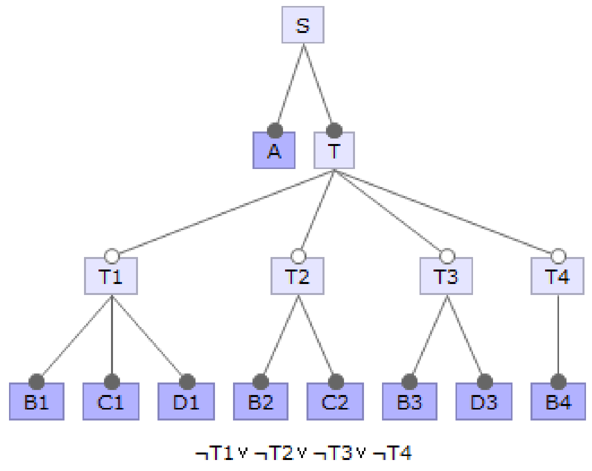


Figure 3.12: Elimination of Clonable Feature - First Variant

First, we have created code-less features S and T using the algorithm given in Section 3.1. Second, we created new subfeatures $T_1, ..., T_4$ and added all possible configurations. Third, we added a constraint for the upper bound $n = 3$. We would also need to add an constraint for the lower bound, but $m = 0$ in this case.

The second transformation creates a new group that contains n-times the subtree of B as in the original feature model (see Figure 3.13). This transformation is only possible if $n \neq *$. Contrary to the above transformation, we do not need the constraint for the upper bound as there are only n children. But in addition we need constraints to ensure that all T_i represent different configurations.

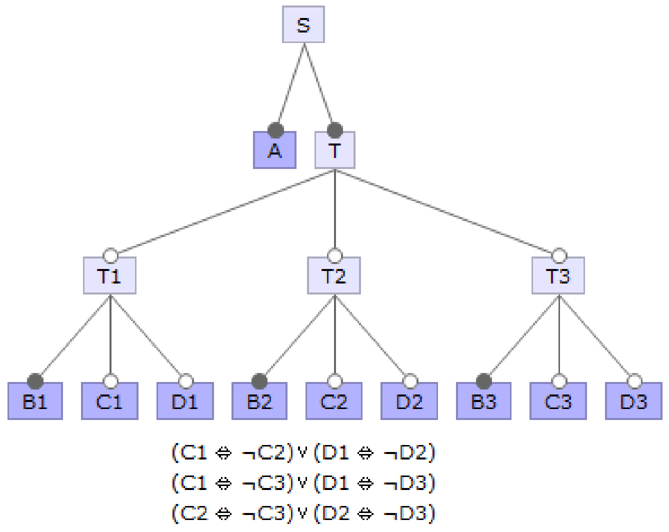


Figure 3.13: Elimination of Clonable Feature - Second Variant

Hence the transformation of feature models with clonable features into GUIDSL Feature Model is possible if we do not allow local attributes. The first transformation is used.

If the upper bound of the feature cardinality is not $*$, we might choose the second transformation in case it smaller.

3.3 Summary

We presented a feature model type named GUIDSL Feature Model. A meta-model was presented for a precise semantics. We noticed that GUIDSL Feature Model is the only type we know that supports the notion of code-less features. Therefore this feature model type can express software product lines that cannot be expressed in any other feature model type.

We analyzed 35 publications that introduced new feature models, presented formal semantics or techniques for automated analysis. Seven properties were abstracted that distinguish several feature model types from each other. We shown representative feature model types for every feature and how they are different from GUIDSL Feature Model.

For the feature model types of 30 publications we presented transformation into GUIDSL Feature Models [KCH+90, GFdA98, KKL+98, CE00, vDK02, vGBS01, RBSP02, dJV02, CBB+03, Beu03, Rie03, SRP03, BHST04, AMS04, ZZM04, SZLW05, BSTRC05, LAMS05, WLS+05, BTRC05, AGM+06, BRCTS06, BSTRC06, TBRC+06, Ben07, CW07, JK07, SHTB07, TBKC07, WSB+08].

We are able to transform the cardinality-based feature models presented in the left 5 publications, if they do not contain local variables [CK05, CHE05a, CHE05b, KC05, CKK06]. But we have claimed that this is not only a problem of our approach. Until know there are is no automated analysis support for feature models with local attributes.

The main advantage of our property-based transformations is, that it can be used to convert feature model types that are not yet published into GUIDSL Feature Models, if they are just new combinations of the shown properties. Therefore GUIDSL Feature Model can now be used for automated analysis and reasoning, since other representations can be converted by tools.

4. Reasoning

Edits on a feature model produce a new feature model. We are interested how the semantics of the old feature model changes, i.e., its software product line. Given two feature models, we want to decide whether the edit is an refactoring, specialization, generalization or an arbitrary edit.

This section is structured as follows. First, we show an existing approach that uses sound operation sets to ensure that a transformation is a refactoring or a generalization. We analyze if it is suitable for reasoning about feature model edits.

The following sections cover the reasoning based on propositional formula. In Section 4.2 we show how satisfiability solvers (SAT solver) can be used to examine semantic changes of feature models. It is a very intuitive approach that seems easy to implement and not practically computable for large feature models. As far as our experience goes both speculations are wrong.

In Section 4.3 we present a technique that we call simplified reasoning. Through the decrease of its time and space complexity it is now practical to reason for large feature models.

The implementation is not straightforward. Problems arise if both feature models do not have all features in common, e.g., if we add an optional feature. We present solutions separately for features that contain code (Section 4.4) and for code-less features (Section 4.5).

4.1 Operation sets

In 2006, Borba et. al. has presented a technique to generally identify refactorings and generalizations[1] for feature models [AGM$^+$06]. A set of four sound operations is

[1]Note that our generalization is equal to their refactoring and what we call refactoring is a bi-refactoring according to their definition.

used to identify refactorings. They have a second set containing 12 operations that are generalizations.

All operations are written generally, that they cover many particular cases. So whenever we use only the operations of one of those sets to change a feature model, we automatically know how the software product line changes. Specialization operations are not needed because they are the reversed generalization operations.

The authors have not shown the completeness of those operation sets. This means that whenever we make changes that do not correspond to any of those operations, we do not know anything about the consequences for the product line.

In fact, we have found different operations (refactorings and generalizations) that are not contained in the given sets. We refer again to Figure 2.2 on Page 5 where a mandatory feature is switched with its parent. This refactoring is not presented in [AGM$^+$06].

At least this shows us that it is not obvious to build a complete set of operations to cover all refactorings and generalizations. A proof might be complicated, too. But given a sound complete set for refactorings and generalizations, how could we realize those operations in an editor?

First, we could implement an editor with different modes (e.g., a refactoring mode). For each mode we have a set of available operations. If the change a user wants to make to a feature model is not one of the primitive operations, the user has to determine which combinations will achieve the edit. Possibly there is no such combination. But that the user does not find a combination, does not mean that there is no combination of primitive operations. Hence it would not tell us that the edit is not allowed in this mode (e.g., it is no refactoring).

Second, we can imagine that a theorem prover would be able to compute, whether changes to a feature model are a combination of basic operations from one of the given sound operation sets.

Hence we do not have complete operation sets, we consider a way using propositional formula in the following sections.

4.2 Reasoning with Propositional Formulas

In this section, we discuss reasoning based on propositional formulas. As introduced in Section 2.1.3, we can transform a feature diagram into a propositional formula. Together with the cross-tree constraints, we get a representation of the feature model in terms of propositional formulas.

We need some symbols to describe our algorithms compactly. In the following $P(f)$ denotes the propositional formula representing the feature model f. We also assign a software product line $L(f)$ to a feature model f, that contains a product, if and only if the propositional formula is `true`, where a feature variable is `true`, if and only if it is contained in the product.

We have defined refactoring, specialization, generalization and arbitrary edit based on set relations. To examine the relation of two software product lines $L(f)$ and $L(g)$, we only need to define how $L(f) \subseteq L(g)$ is represented in propositional formula $P(f)$ and $P(g)$. The boolean value of $L(f) \subseteq L(g)$ and $L(g) \subseteq L(f)$ tells us, in which relation $L(f)$ and $L(g)$ are.

A simple approach for feature models f and g that contain the same set of features can be found in the literature [JK07]. A basic idea they discuss is the following.

$$(L(f) \subseteq L(g)) \equiv (P(f) \Rightarrow P(g)) \tag{4.1}$$

This formula follows our intuition. $P(f) \Rightarrow P(g)$ means that $P(f)$ contains all constraints of $P(g)$ and maybe more. Therefore all products of the according software product line $L(f)$ are contained in $L(g)$.

We want to compute whether $P(f) \Rightarrow P(g)$ is valid for all allocations of boolean values to variables using a SAT solver. SAT solvers are able to compute whether a given formula is satisfiable or unsatisfiable. Since we are interested if a propositional formula X is valid, we check if its negation $\neg X$ unsatisfiable. The following equation shows, how we can compute $\neg X$ with $X = P(f) \Rightarrow P(g)$ [TBKC07].

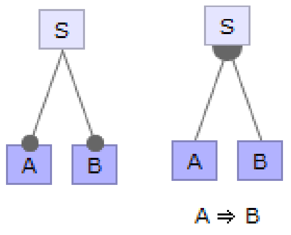
$$\neg(P(f) \Rightarrow P(g)) \equiv \neg(\neg P(f) \vee P(g)) \equiv P(f) \wedge \neg P(g) \tag{4.2}$$

SAT solvers usually accept a conjunction of several logical operators that are based on literals. But we are not able to put in interleave logical operators, e.g., $\neg(A \wedge (C \vee D))$. Therefore we have to convert propositional formulas into *conjunctive normal form (CNF)*. There are three steps to do and every step has to ensure the semantics.

1. Eliminate all logical operators except AND, OR and NOT.
2. Relocate all NOT operations to the lowest level.
3. Convert all parts where an AND is on a lower level than an OR.

Example. Given two feature models f and g we want to compute the relation of their software product lines $L(f)$ and $L(g)$. The feature models we consider are shown in Figure 4.1.

First, we are interested whether all products of $L(f)$ are contained in $L(g)$. Concerning Equation 4.1 and 4.2 we convert $\neg(P(f) \Rightarrow P(g))$ into CNF using the above discussed steps. In the last step we have removed all clauses that are always true, i.e., that contain a variable positive and negative. Furthermore we have removed literals that already occur in a disjunction.



(a) FM f (b) FM g

Figure 4.1: Example for Reasoning using Propositional Formulas

$$
\begin{aligned}
& \neg(P(f) \Rightarrow P(g)) \\
\equiv\; & P(f) \wedge \neg P(g) \\
\equiv\; & (S \wedge (S \Rightarrow (A \wedge B)) \wedge ((A \vee B) \Rightarrow S)) \\
& \wedge \neg(S \wedge (S \Leftrightarrow (A \vee B)) \wedge (A \Rightarrow B)) \\
\overset{1}{\equiv}\; & (S \wedge (\neg S \vee (A \wedge B)) \wedge (\neg(A \vee B) \vee S)) \\
& \wedge \neg(S \wedge ((\neg S \vee (A \vee B)) \wedge (\neg(A \vee B) \vee S)) \wedge (\neg A \vee B)) \\
\overset{2}{\equiv}\; & (S \wedge (\neg S \vee (A \wedge B)) \wedge ((\neg A \wedge \neg B) \vee S)) \\
& \wedge(\neg S \vee ((S \wedge (\neg A \wedge \neg B)) \vee ((A \vee B) \wedge \neg S)) \vee (A \wedge \neg B)) \\
\overset{3}{\equiv}\; & S \wedge (\neg S \vee A) \wedge (\neg S \vee B) \wedge (\neg A \vee S) \wedge (\neg B \vee S) \\
& \wedge(\neg S \vee S \vee A \vee B \vee A) \wedge (\neg S \vee S \vee A \vee B \vee \neg B) \\
& \wedge(\neg S \vee S \vee \neg S \vee A) \wedge (\neg S \vee S \vee \neg S \vee \neg B) \\
& \wedge(\neg S \vee \neg A \vee A \vee B \vee A) \wedge (\neg S \vee \neg A \vee A \vee B \vee \neg B) \\
& \wedge(\neg S \vee \neg A \vee \neg S \vee A) \wedge (\neg S \vee \neg A \vee \neg S \vee \neg B) \\
& \wedge(\neg S \vee \neg B \vee A \vee B \vee A) \wedge (\neg S \vee \neg B \vee A \vee B \vee \neg B) \\
& \wedge(\neg S \vee \neg B \vee \neg S \vee A) \wedge (\neg S \vee \neg B \vee \neg S \vee \neg B) \\
\equiv\; & S \wedge (\neg S \vee A) \wedge (\neg S \vee B) \wedge (\neg A \vee S) \wedge (\neg B \vee S) \\
& \wedge(\neg S \vee \neg B \vee A) \wedge (\neg S \vee \neg B)
\end{aligned}
$$

A SAT solver can be asked if the last or the second last formula is satisfiable and it will answer, that the given formula is unsatisfiable. For this simple example it can easily be proven. We know that S has to be `true`. But if S is *true*, B has to be *true* due to $(\neg S \vee B)$ and *false* due to $(\neg S \vee \neg B)$, what is impossible.

Since $\neg(P(f) \Rightarrow P(g))$ is unsatisfiable, we know that $L(f) \subseteq L(g)$. In an analogous manner we can show that $L(f) \supseteq L(g)$ is not correct. Therefore we know that $L(f) \subset L(g)$ and the edit on feature model f to achieve feature model g is a generalization. □

Complexity. When we map a feature diagram to a propositional formula and add also the cross-tree constraints, we get a big conjunction of smaller propositional formulas.

Converting it into a CNF only consist of converting all its parts into CNF and do the above defined step 4 for the whole formula. Table 4.1 shows how to convert rules that are generated from our feature diagram.

Connection	Propositional Formula	Conjunctive Normal Form
And	$(S \Rightarrow A \wedge C) \wedge$ $(A \vee B \vee C \Rightarrow S)$	$(\neg S \vee A) \wedge (\neg S \vee C) \wedge$ $(\neg A \vee S) \wedge (\neg B \vee S) \wedge (\neg C \vee S)$
Alternative	$(T \Leftrightarrow D \vee E \vee F) \wedge$ $\mathit{atmost1}(D, E, F)$	$(\neg T \vee D \vee E \vee F) \wedge$ $(\neg D \vee T) \wedge (\neg E \vee T) \wedge (\neg F \vee T) \wedge$ $(\neg D \vee \neg E) \wedge (\neg D \vee \neg F) \wedge (\neg E \vee \neg F)$
Or	$(U \Leftrightarrow G \vee H \vee I)$	$(\neg U \vee G \vee H \vee I) \wedge$ $(\neg G \vee U) \wedge (\neg H \vee U) \wedge (\neg I \vee U)$

Table 4.1: Conversion of Feature Diagram Rules to CNF

In CNF we get longer propositional formulas whereas each disjunction does not contain more variables than participated at a connection (number of children plus one). Therefore we look at the number of generated formula more in detail. k denotes the number of children in a connection.

The number of disjunctions when we convert an *And* to CNF is $\Theta(k)$, because we have between k and $2k$ disjunctions of two literals. For *Or* we have exactly $k + 1$ disjunctions and get $\Theta(k)$, too. The conversion of *atmost1* is more expensive. For every two variables we have to generate one rule containing two litarals. Hence we have exactly $k + 1 + \frac{1}{2}\binom{k}{2} = \frac{1}{4}k^2 + \frac{3}{4}k + 1$ disjunctions, i.e., $\Theta(k^2)$.

Overall for a feature diagram with c connections with at most m children we can get up to $O(c \cdot m^2)$ disjunctions in a CNF. Let n denote the number of features. Then we know that $m < n$ and $c \leq n - m$. Thus we know that $O(n^2)$ is an upper bound for the number of disjunctions for all tree constraints in CNF. □

What about cross-tree constraints? As mentioned in Chapter 3 we want to consider arbitrary propositional formula as cross-tree constraints. It is complicated to give an upper bound, but we give an example to show how expensive it can be.

Example. We consider the following cross-tree constraint and its CNF.

$$\begin{aligned}
& (A \wedge \neg B) \vee (\neg C \wedge \neg D) \vee (\neg E \wedge F \wedge G) \\
\equiv\ & (A \vee \neg C \vee \neg E) \wedge (A \vee \neg C \vee F) \wedge (A \vee \neg C \vee G) \\
& \wedge (A \vee \neg D \vee \neg E) \wedge (A \vee \neg D \vee F) \wedge (A \vee \neg D \vee G) \\
& \wedge (\neg B \vee \neg C \vee \neg E) \wedge (\neg B \vee \neg C \vee F) \wedge (\neg B \vee \neg C \vee G) \\
& \wedge (\neg B \vee \neg D \vee \neg E) \wedge (\neg B \vee \neg D \vee F) \wedge (\neg B \vee \neg D \vee G).
\end{aligned}$$

In this example we have $2 \cdot 2 \cdot 3 = 12$ disjunctions in a conjunction after the conversion into CNF. Let x be the number of conjunctions in a disjunction and each of the conjunctions contains at most m literals, then we get up to $O(m^x)$ disjunctions as a result. $\square$

Limitations. We have encountered that cross-tree constraints are usually much easier to convert, because their structure is not so far away from a CNF. But we have to be aware of the fact, that there might be constraints that cannot be converted into CNF practically. In the following we treat cross-tree constraints as practical convertible into CNF.

Are we able to convert the part $\neg P(g)$ into CNF? Practically it is not possible. We have tried this for all our feature models and in every case the default Java heap space was not enough to compute it. The problem is that we get a big disjunction of smaller propositional formula after step 2 (de Morgan's Theorem).

It is exactly the same problem as shown in the above example. But there is one big difference: we cannot assume that $P(g)$ is practical convertible into CNF. The reason is, that feature diagrams always consist of rules that are conjunctive related. Its negation will ever be a disjunction of all its negated rules. In the following section we will show an approach to avoid the conversion of $\neg P(g)$ into CNF. $\square$

In this section, we have shown how to reason on feature model edits using propositional formula. The calculation of a CNF that is associated to a feature model takes at most $O(n^2)$ time for tree constraints and exponential time for cross-tree constraints, where n denotes the number of features. For reasoning as shown in this section we also need the CNF of the negated propositional formula, what needs exponential time and is practical not possible.

4.3 Simplified Reasoning

There is a simple way to reduce the costs, mentioned in the previous section. Whenever we talk about feature models f and g, we assume that they have some differences but they also may have a lot of rules that are identical. We can rewrite $P(f)$ and $P(g)$ by

$$\begin{aligned} P(f) &= p_f \wedge c \\ P(g) &= p_g \wedge c \end{aligned}$$

where c are the identical rules and p_f are rules that are contained in $P(f)$ but not in $P(g)$. $P(f) \Rightarrow c$ is always true, because $P(f) \wedge \neg c \equiv p_f \wedge c \wedge \neg c$ is unsatisfiable. Combined with 4.2 we get the following equivalence.

$$\neg(P(f) \Rightarrow P(g)) \equiv P(f) \wedge \neg p_g \tag{4.3}$$

This simply reduces the amount of n to $n' \leq n$, but we still have this fast growing function and it is only possible to make small changes to the feature model f. The propositional formula p_g is a conjunction of n' rules $R_1, R_2, ..., R_{n'}$. Together with 4.3 we state the following simplification.

$$\begin{aligned}\neg(P(f) \Rightarrow P(g)) &\equiv P(f) \wedge \neg(R_1 \wedge R_2 \wedge ... \wedge R_{n'}) \\ &\equiv P(f) \wedge (\neg R_1 \vee \neg R_2 \vee ... \vee \neg R_{n'}) \\ &\equiv (P(f) \wedge \neg R_1) \vee (P(f) \wedge \neg R_2) \vee ... \vee (P(f) \wedge \neg R_{n'}) \quad (4.4)\end{aligned}$$

That means we ask a SAT solver up to n'-times if $P(f) \wedge \neg R_i$ is satisfiable. If only one of these conjunctions is satisfiable, then $\neg(P(f) \Rightarrow P(g))$ is satisfiable, too. Because this means that $P(f) \Rightarrow P(g)$ is not valid, we can stop if we find a satisfiable conjunction.

How expensive is the conversion of each $\neg R_i$ into CNF? It is easy if we assume that p_g is already in CNF. We can transform p_g into CNF as described in the last chapter. Hence all R_i are disjunctions of literals, what makes it easy to convert.

We have presented an algorithm called simplified reasoning. Contrary to reasoning that is already mentioned in the literature, this is a new strategy to reduce the costs. Compared to the complexity of the afore shown reasoning it is no longer necessary to compute the CNF of a negated feature model, what makes it possible to compute even for larger feature models.

4.4 Extension for Adding and Removing Features

Until now we only talked about feature models that have the same features. But what if we add or delete features? The next chapter will give an example that we need to add additional informations, to the model that does not have a feature the other feature model has.

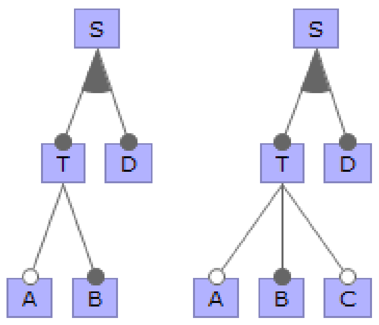
It seems obvious that 4.1 holds, but this statement is just valid if g was only produced by changes on the feature model f that do not change the set of contained features. i.e., we cannot create or delete features to produce g out of f. In the following we give an counter example for not yet convinced readers.

Example. Figure 4.2 contains a feature model f were we have added an optional feature C to get feature model g. We know that this is a generalization because g contains all products of f and more.

Lets look at $P(f)$ and $P(g)$ to see, if $P(f) \Rightarrow P(g)$ holds.

$$\begin{aligned}P(f) &= (S \wedge (S \Leftrightarrow (T \vee D)) \wedge (T \Rightarrow B) \wedge ((A \vee B) \Rightarrow T)) \\ P(g) &= (S \wedge (S \Leftrightarrow (T \vee D)) \wedge (T \Rightarrow B) \wedge ((A \vee B \vee C) \Rightarrow T))\end{aligned}$$

$P(f) \Rightarrow P(g)$ holds if and only if the following four formulas hold.



(a) FM f (b) FM g

Figure 4.2: Example were the Presented Approach is Incorrect

$$\begin{aligned} P(f) &\Rightarrow S \\ P(f) &\Rightarrow (S \Leftrightarrow (T \vee D)) \\ P(f) &\Rightarrow (T \Rightarrow B) \\ P(f) &\Rightarrow ((A \vee B \vee C) \Rightarrow T) \end{aligned}$$

We assign the value `true` to the variables S, D and C and the value `false` to T, A and B, to show that $P(f) \Rightarrow P(g)$ is not valid. The fourth formula is `false` for these values. This means that $P(f) \Rightarrow P(g)$ is not valid.

But we know that the transformation from f to g is a generalization, which requires $P(f) \Rightarrow P(g)$. Therefore the presented approach does not work for feature models, which have different sets of features. $\square$

We will now show an improved approach that works for Figure 4.2, too. The idea is simple: A feature that is not contained in a feature model cannot be selected. We have to add this information to the propositional formula. The problem exists because we put both propositional formulas together in one. Every feature that is not contained in a feature model, does not occur in its propositional formula. That means it can be selected, while it is not contained in the feature model.

For every feature A that is contained in g but not in f we add the new clause $\neg A$ to the conjunction $P(f)$ resulting in $P'(f)$. Analogously we add a new clause $\neg B$ to the conjunction $P(g)$ for every feature B that is contained in f but no longer in g resulting in $P'(g)$.

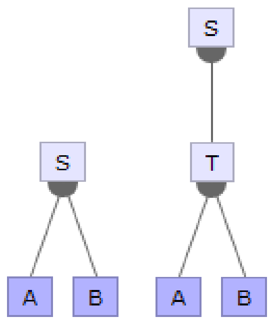
$$(L(f) \subseteq L(g)) \Leftrightarrow (P'(f) \Rightarrow P'(g)) \tag{4.5}$$

The algorithms reasoning and simplified reasoning have to be extended, if we allow edits that create or delete features. We simply add an additional constraint $\neg F$ to a model for every feature F that is not contained in the feature model, but in the second one.

4.5 Extension for Code-less Features

The algorithm presented in the previous section returns the correct results for two given feature models that are only based on features, that are associated with code. But what happens if we create or delete code-less features, whose selection or deselection does not matter? The next section gives an example why we should not treat code-less features as described in the previous section.

Example. Figure 4.3 shows a feature model f, where A and B are *Or*-connected. We have produced another feature model g by creating a new abstract feature T. It is obvious that $L(f) = L(g)$.



(a) FM f (b) FM g

Figure 4.3: Example were the Dynamic Approach is Incorrect

The corresponding adapted propositional formula are as follows.

$$\begin{aligned} P'(f) &= (\neg T \wedge S \wedge (S \Leftrightarrow (A \vee B))) \\ P'(g) &= (S \wedge (S \Leftrightarrow T) \wedge (T \Leftrightarrow (A \vee B))) \end{aligned}$$

In the same manner as above we show that $P'(g) \nRightarrow P'(f)$. The following formula is `false` when we assign the value `true` to all variables.

$$P'(g) \Rightarrow \neg T$$

One could think that it is a better idea to add T instead of $\neg T$, but it does not work either. The reason is simply that code-less features can also be optional. You can also produce a counter example, if you only add extra rules for added and deleted code-containing features in $P(f)$ and $P(g)$. □

Code-less features do always have children. If they would not have children, they would be useless for us and we could delete them. For this reason we can (usually) compute the value of a code-less feature given the boolean values of its children. We call a propositional formula that computes the value of a given code-less feature its *definition*. The idea discussed in the following is simple: We replace every occurrence of a code-less feature by its definition.

We refer again to Figure 2.1 on Page 4. The definition for *Alternative* and *Or* is identical. The parent is true, if and only if one or more of its children are selected, e.g., $T \Leftrightarrow (D \vee E \vee F)$ and $U \Leftrightarrow (G \vee H \vee I)$. For *And* it is the conjunction of all mandatory features, e.g., $S \Leftrightarrow (A \wedge C)$, if it has at least one mandatory feature.

What happens if an *And*-feature S has no mandatory child at all? In this case we cannot tell the boolean value of S, if all its children are deselected. And this means we have no unambiguous definition. We present an algorithm to eliminate such groups in Chapter A. However, this algorithm can only be applied if we have no cross-tree constraints. Therefore we assume that these groups do not occur in our feature models. Note that these groups are not necessary and can be replaced manually, while some cross-tree constraints might have to be altered.

How big is the resulting propositional formula in the worst case? A definition contains at most all code-containing features. The code-less feature can occur $O(n)$ times in all tree constraints. The cross-tree constraints can contain a code-less feature at most once for each disjunction. If a disjunction contains a feature more than once it is ever true and we can delete it (e.g., $S \vee \neg S \vee \ldots$) or we can delete all duplicates before replacing it with its definition (e.g., $S \vee S \vee \ldots$). Let k denote the number of disjunctions, then the propositional formula can get up to $O(k \cdot n^2)$ new literals in all disjunctions. The number of disjunctions remains the same.

Example. The worst case could be that we have a feature model with one code-less feature T and we insert an code-less feature S that gets all children of T (see Figure 4.4).

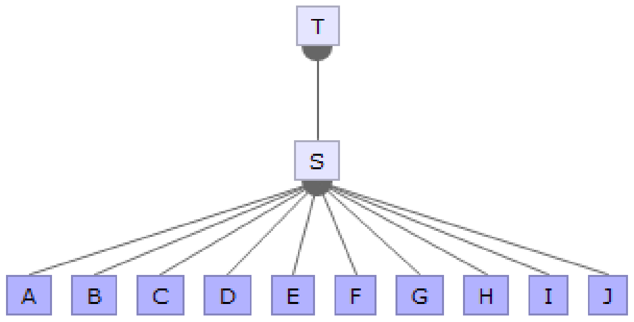


Figure 4.4: A Worst Case Edit on a Worst Case Feature Model

Additionally we insert the new feature S in every cross-tree constraint. This might not make sense, because S is already contained in almost all tree constraints, but it can happen. This example can give the reader an idea how unusual and rare the worst cases are. □

Notice that we only have to replace code-less features by there definition if they are not contained in both models or if the definition in one model is different from the second one. That can also save calculation time.

4.6 Summary

The present chapter analyzes existing and presents new techniques for reasoning about feature model edits. Reasoning is the process were we assign one or more edits to a feature model either to a refactoring, specialization, generalization or arbitrary edit.

The existing approach of operation sets for each type is not much useful for reasoning as long as the operation sets are not complete. We have also shown that it is not recommendable to allow only particular operations, because the user has to match the edit he wants to make to a composition of the given operations, which can be difficult.

We discussed reasoning based on propositional formulas. It was presented how SAT solvers can be used to reason on feature model edits. For this approach it is necessary to convert the negation of a propositional formula according to a feature model into CNF. This is practically not possible.

Reasoning using the fact that one feature model implies another one can be simplified. We presented an algorithm called simplified reasoning that avoids the exponential conversion of the negation of a feature model to CNF. Our algorithm runs in $O(n^2)$, given n features.

We have also shown how to handle features that are not contained in both models. For every code-containing feature that is not existing in one feature model we add a constraint which forbids the selection of that particular feature. Code-less features that are not contained in the second model are replaced by a propositional formula that calculates their boolean value.

Because we have reduced the problem of reasoning on feature model edits on the satisfiability problem, we still have an exponential run-time in the worst case. We have shown an example of a worst case feature model and it does not look like a popular feature model. In the next section we will analyze the run-time empirically and we will see that the run-time for common and even for large feature models is satisfactory.

5. Evaluation

In Chapter 3, a wide analysis on existing feature models is accomplished. Since work on the automated analysis and reasoning should not only be based on a particular feature model type, GUIDSL Feature Model and transformations from wide range of feature models into this feature model type were presented.

Chapter 4 has introduced a technique for reasoning about feature model edits based on propositional formulas. The latter were chosen that off-the-shelf tools like satisfiability solvers can be used for efficient calculation.

We have shown that the problem, whether a feature model contains all products of a second one, can be conveyed into a satisfiability problem. The latter is, given a set of variables and a propositional formula in CNF, whether the given propositional formula is satisfiable.

It was not possible to give a general upper bound for the transformation into a satisfiability problem, since we allow arbitrary propositional formulas as constraints. To convert these into CNF has an exponential complexity in the worst case.

Furthermore, we can give no polynomial upper bound for solving if a propositional formula is satisfiable, because this problem is known as *NP-complete*. Hence an empirical analysis is indispensable to evaluate the presented algorithm.

The technique presented in Chapter 4 is implemented using the SAT solver Sat4J. We tested the implementation with several feature models from practical examples. After editing the feature model in multiple steps the result, whether the edit made to the feature model is a refactoring, specialization, generalization or an arbitrary edit, was calculated within a second. However, to make sure that the tested feature models are not only special cases, we decided to test our reasoning approach on generated feature models.

5.1 Generating Feature Models

We implemented an algorithm to generate arbitrary feature models that takes as input the number of features and the number of children that a feature can have at most. The algorithm for generation was implemented in such a manner that the following conditions hold.

1. The features that get children are random chosen.
2. The count of children a feature has, is calculated randomly and is between zero and the input for the maximum count of children.
3. That a feature has an *And*-, *Or*- or *Alternative*-group to its children has the same probability.
4. For features that are in an *And*-group it is equiprobable that it is mandatory and that it is optional.

An example generated model with 20 features and at most five children can be found in Figure 5.1. But as a feature model consists of a feature diagram and constraints, we describe in the following how constraints are created.

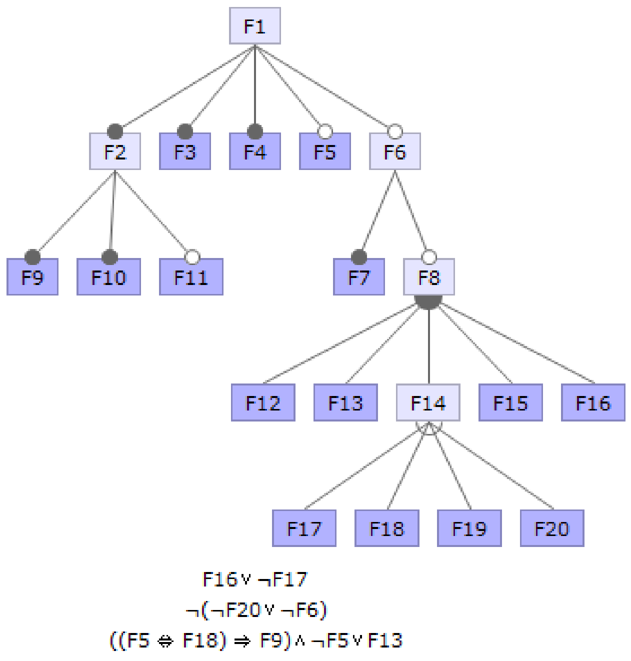


Figure 5.1: Random Generated Feature Model

A second method generates random constraints. There are two parameters for the calculation. First, the desired number of constraints is a parameter. Second, the maximum number of variables, on which a constraint is defined. We used the parameters three and five in Figure 5.1. The method fulfills the following conditions.

1. The constraints are constructed using $\neg, \wedge, \vee, \Rightarrow$, and $\Leftrightarrow$.
2. The output contains between 1 and the maximum desired count of literals, i.e., variables or negated variables.
3. A constraint is added if the feature model still contains at least one product, otherwise a new constraint is generated instead.

Our algorithm presented in Chapter 4 takes two feature models f, g as input and calculates if an edit on f resulting in the feature model g is a refactoring, specialization, generalization, or an arbitrary edit.

5.2 Performance and the Number of Features

We can use both methods to generate feature models. The resulting random generated feature models were taken as input to our algorithm to analyze the runtime in relation to the number of features in a feature model. We compare the runtime for different categories of edits in Figure 5.2. The exact values in milliseconds can be found in Table B.1 on Page 51.

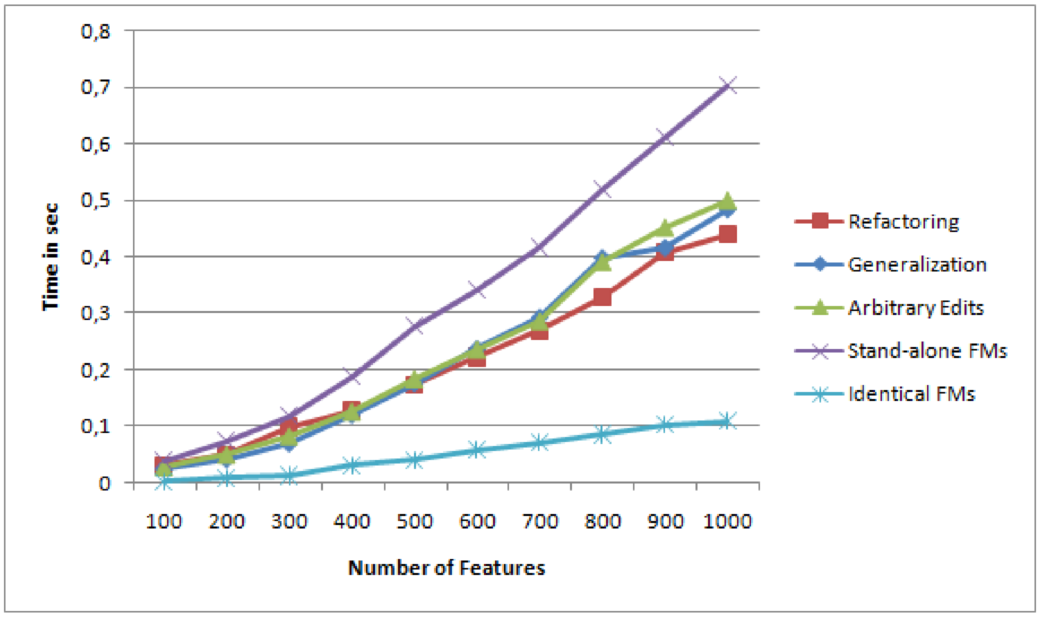


Figure 5.2: Calculation Time in Relation to the Number of Features

For instance, we compared two stand-alone feature models, i.e., two independent generated feature models. That means that the output can be each of the possible outputs defined above. The second row that can be measured is that the input consists of two identical feature models. In this case the output is always that the edit is a refactoring.

Since we use random generated models we need a number of iterations to get average values. We decided to repeat the calculation until the average has changed in the last five steps (stability constant) less than 0.1 percent (precision constant). A summary of all used parameters can be found in Table 5.1.

Parameter	Value
Number of Features	$100, 200, ..., 1000$
Maximum Number of Children	10
Number of Constraints	10
Maximum Number of Literals	5
Number of Edits	10
Stability Constant	5
Precision Constant	0.001

Table 5.1: Parameters used in Figure 5.2

5.3 Generating Edits on Feature Models

Both already presented value rows are not typical inputs for our algorithm. Usually we take a feature model make some edits and want to know of which category the edit is. Therefore it was necessary to implement methods that edit a given feature model by a specified number of simple edits (number of edits), which will be discussed in the following.

We decided to implement such methods for refactorings, generalizations and arbitrary edits. Note that there is no need to write such a method for specialization, since if f is a generalization of g, g is a specialization for f. Because our algorithm works symmetric we have not measured specializations in our empirical analysis.

As refactorings and generalizations we have implemented the operations presented by Alves et. al. [AGM+06]. For arbitrary edits we implemented the following operations.

1. Create or delete a primitive feature, i.e., a feature without children.
2. Change the type of a group, i.e., to a *And*-, *Or*, or *Alternative*-group that is different from the type before.
3. Make a mandatory feature optional or vice versa.
4. Create a new constraint with the above defined parameters or delete a constraint.

In Figure 5.2, the runtime increases in all cases almost linear with the number of features. It can be seen that stand-alone feature models take the most time for comparison and identical feature models can be compared seven times faster. All categories are almost in the same range, so that it is not relevant which edit me make to a feature model.

5.4 Performance and the Number of Edits

We are also interested how much more time is needed for more edits at the feature model. To answer this question, we analyzed the runtime in relation to the number of edits (see Figure 5.3).

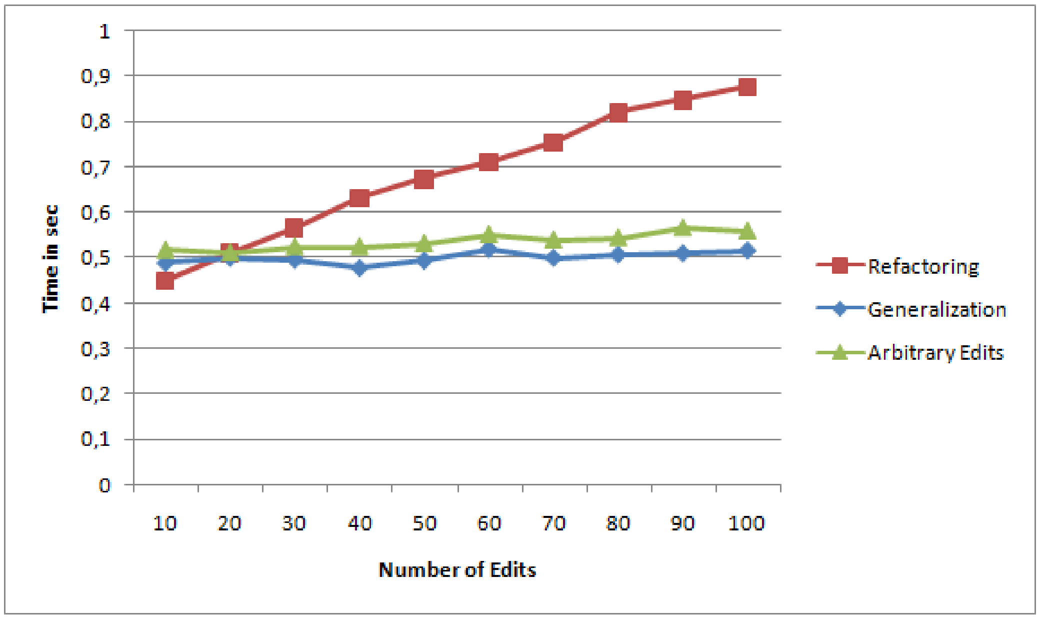


Figure 5.3: Calculation Time in Relation to the Number of Edits

Since we do not have a number of edits for stand-alone or identical feature models, these rows are not measured. The values can be found in Table B.2 on Page 52 and the used parameters are in Table 5.2.

Parameter	Value
Number of Features	1000
Maximum Number of Children	10
Number of Constraints	10
Maximum Number of Literals	5
Number of Edits	$10, 20, ..., 100$
Stability Constant	5
Precision Constant	0.001

Table 5.2: Parameters used in Figure 5.3

The results are surprisingly. While we have ten-times more refactorings applied to a feature model the runtime is only approximately twice as high. For generalizations and arbitrary edits is the result is much better, because the number of edits seems not to have an influence on the runtime and all results are calculated in about half a second.

The reason for the constant time is that the algorithm aborts for generalizations if one product is found that is added and all changes in the feature model add products. The time for arbitrary edits is a little bit higher since also a product must be found that was deleted. The calculation for refactorings is more time-consuming, hence we have to check all changed rules, which increases linear with the number of edits.

5.5 Performance and the Number of Children

In the previous chapter we presented an example feature model that is hard to convert to CNF. The feature model had only one compound feature that has all other features as children. Therefore we decided also to analyze the performance in relation to the maximum number of children. Parameters and values are given in Table 5.3 and in Table B.3 on Page 52. Note that we used feature models with 100 instead of 1000 features.

Parameter	Value
Number of Features	100
Maximum Number of Children	$10, 20, ..., 100$
Number of Constraints	10
Maximum Number of Literals	5
Number of Edits	10
Stability Constant	5
Precision Constant	0.001

Table 5.3: Parameters used in Figure 5.4

In Figure 5.4 it can be seen that we have non-linear increasing functions on the number of children. We have shown in Chapter 4 that our algorithm has a quadratic complexity to convert *Alternative*-groups into CNF. This the reason that the calculation for identical feature models is faster than for generalizations and generalizations are faster to compute than the other edits.

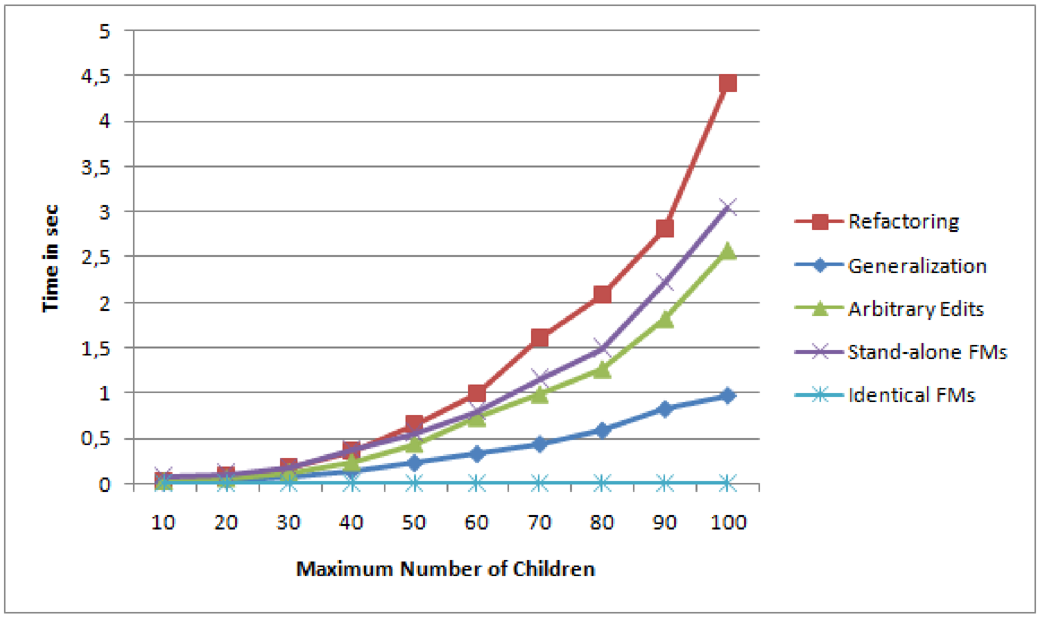


Figure 5.4: Calculation Time in Relation to the Maximum Number of Children

For identical feature models none of both input feature models has to be converted into CNF. A generalization removes constraints from the feature model. To show that the constraints are not any more contained in the new feature model, the latter has to be converted into CNF. Refactorings are usually hard to compute, since both feature models have to be converted into CNF. Notice that for arbitrary edits and stand-alone feature models the algorithm might also have to compute the CNF for both models, but not in all cases.

This result is very bad since we already need up to five seconds in average to compute the relation of two given feature models on 100 features. Note that a maximum number of 100 children in a feature models with 100 features is no restriction. But now we can show the great advantage of GUIDSL Feature Models which allow code-less features.

5.6 On the Number of Children

Every GUIDSL Feature Model can be converted into a GUIDSL Feature Model with at most k children for each feature by adding new code-less features. For instance, we have converted the GUIDSL Feature Model in Figure 4.4 on Page 32 to the GUIDSL Feature Model in Figure 5.5, which contains at most 3 children for each feature.

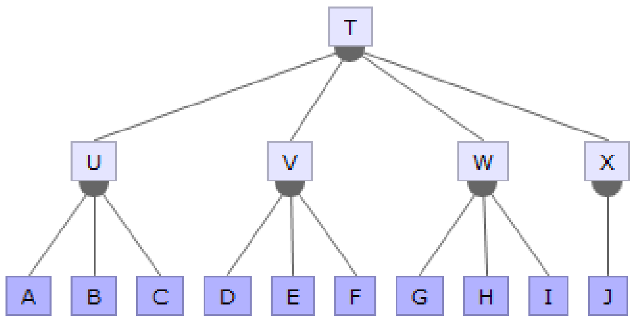


Figure 5.5: Equivalent Feature Model to Figure 4.4

Therefore the already used upper bound of ten children for the calculations in Figure 5.2 and Figure 5.3 is no restriction. An editor, for instance, could allow a maximum number of ten children without loss of expressiveness.

5.7 Summary

Chapter 3 presented GUIDSL Feature Models and how other feature model types can be converted into feature models of the former type. This work enables us to implement automated analysis and reasoning for a variety of feature model types by implementing it on a GUIDSL Feature Model to which other can easily be converted.

In Chapter 4, an algorithm for reasoning about feature model edits is proposed based on GUIDSL Feature Models. The algorithm has a exponential complexity in the worst case, since (i) we use SAT solvers and (ii) we allow arbitrary constraints that might be hard to convert into CNF.

Therefore, this chapter has analyzed the runtime for generated feature models and generated edits on those feature models. The empirical analysis is done separately for the different categories of edits, such as refactorings, generalization and arbitrary edits. Specializations do not have to be analyzed, since the result would be the same as for generalizations.

We experienced a linear growth in relation to the number of features. Whereas the results for all categories of edits are very close to each other. Comparing stand-alone, i.e., independently generated feature models, is the most costly.

The number of edits made to a feature models, seems to have no influence on the runtime in case of generalizations and arbitrary edits. A slow-growing linear function describes the calculation time for refatorings.

The maximum number of children that a feature can have in the feature model has a strong influence on the calculation time. For all categories of edits we get a growths faster than linear, whereat refactorings need the most calculation time followed by arbitrary edits and generalizations. But we have shown that limiting the number of features in a GUIDSL Feature Model is not a restriction in terms of expressiveness.

Unfortunately, we cannot use all feature models as input so far, since we only allow feature models where every *And*-group has at least one mandatory feature. This condition already holds for all feature model types that we found in the literature, since the do not have code-less features and the transformation creates one mandatory feature in each *And*-group. At least a procedure to refactor a GUIDSL Feature Model so that it no longer contains is given in Chapter A. This algorithm can only be applied manually since some constraints may have to be altered.

6. Related Work

Generic Semantic

Schobbens et al. presented a generic semantic for feature models [SHTB07]. They propose that feature models should be interpreted as a quadruple that specifies (i) whether the feature model is a tree or a DAG, (ii) the supported group types as well as (iii) the allowed graphical and (iv) textual constraints.

We think that the differentiation between the latter two is not very useful, since it is just a difference in notation and the generic meta model should be more abstracted. Furthermore, they cannot be applied for the most feature models surveyed in this thesis, since they do not allow the following constructs in their meta model.

(i) Groups defined on cardinalities.

(ii) Restriction in which groups optional features are allowed.

(iii) Several groups are not supported, since they decided to consider groups as symmetrical functions.

(iv) Attributes and constraints on attributes.

(v) Cloning of features, i.e., feature cardinalities.

Note that we observed all these properties of feature model types in Section 3. A very useful methodology is that they allow to specify a subset of the set of features to indicate code-less features for all feature model types.

Sound Operations for Feature Model Edits

Czarnecki et al. has introduced specializations of cardinality-based feature models [CHE05a, CHE05b, KC05]. Among to their notation specialization is a feature model that contain less products in relation to a second feature model. Operations were presented to create a specialization of a feature model, e.g., selecting an optional feature.

Alves et al. published operations that correspond to refactorings as well as generalizations [AGM$^+$06]. As we have shown in our work, the set of operations is not complete. Although, we could validate all these operations as proper using our tool.

Reasoning about Feature Model Edits

Sun et al. presented a formal semantics for feature models using first-order logic and stated that the theorem prover Z/EVES can be used for analysis [SZLW05]. Furthermore, they stated how equivalent feature models can be identified with the condition that both feature models are defined on the same set of features. Note that we can have equivalent feature models that have not all features in common, e.g., if we have code-less features or if one feature is dead in one feature model.

We give another comment on this proposal. The constraint $F3 \wedge F4 \Rightarrow F2$ is not equal to $(F3 \Rightarrow F2) \wedge (F4 \Rightarrow F2)$. Since the brackets can be applied it two different ways it is either $(F3 \wedge F4) \Rightarrow F2 \equiv \neg F3 \vee \neg F4 \vee F2$ or $F3 \wedge (F4 \Rightarrow F2) \equiv F3 \wedge (\neg F4 \vee F2)$, but both are different from $(F3 \Rightarrow F2) \wedge (F4 \Rightarrow F2) \equiv (\neg F3 \vee F2) \wedge (\neg F4 \vee F2)$.

Janota and Kiniry proposed reasoning on feature models using higher-order logic [JK07]. A meta model was presented to describe feature models with higher-order logic. They give a formal definition of a specialization that is more liberal, because it also captures refactorings.

The main disadvantage of their approach is that both feature models have to be defined on the same set of features. Therefore, if we add an optional feature to a feature model, we cannot ask their theorem if the edit is a specialization. This is a very strong restriction. They also do not deal with code-less features.

7. Conclusion

Feature models are used to specify the members of a software product line. While the automated analysis aims on analyzing feature models statically, reasoning on feature model edits is about changes on feature models. The previous results on reasoning are not sufficient.

In this thesis the first algorithm was presented to compare any feature models and therefore to classify edits made from one feature model to another. Previous presented approaches deal with sound operations sets to maintain the set of products or add products [AGM$^+$06] and to remove products [CHE05a, CHE05b, KC05].

Other approaches that use first- or higher-order logic have only presented how to identify if two feature models do allow the same set of products [SZLW05] or if one has less products [JK07], with the restriction that both feature models are defined on the same set of features.

We propose that feature models are not static and we need tool support, e.g., if new products are going to be added or if it is necessary that all current products will remain unchanged. That this work is also relevant for the configuration process can be seen by several publications on specializations [CHE05a, CHE05b, KC05, JK07].

Our presented algorithm can be adopted to almost all feature models as shown in Chapter 3. The only exception are feature models with attributes that get their value within the configuration process and on which the feature model contains constraints. Further work should issue on combining such attributes and automated analysis and reasoning. The main contributions of our proposal are the following.

1. The algorithm is implemented on a feature model type, where almost all feature models can be transformed into.

2. We use a simplification on propositional formulas that we do not have to compute the CNF of a negation of a feature model.

3. Our algorithm is not restricted to feature models defined on the same set of features.

4. We allow code-less features in our feature model type, that can help to reduce the runtime costs.

5. An empirical analysis has shown that the algorithm scales for feature models with more than 1000 features and for more than 100 edits.

The proposed algorithm was implemented and validated with already classified operations known in the literature. The calculation time for feature model with up to 1000 is usually lower than one second. Hence the implemenation can be used to classify edits on-the-fly, while the user edits the feature model. By this, a domain analyst get support for refactorings, generalizations and specializations.

A. Eliminate *And*-Groups without Mandatory Children

We mentioned in Section 4.5 that *And*-groups, where all child-features are optional have no unambiguous definition. We present how a feature model without cross-tree constraints can be refactored, that all code-less features have a definition.

All *And*-groups without mandatory children were replaced by *Or*-groups. In the following S denotes the parent feature and A, B, C are representative all children of S. We have to ensure that the S can be empty, i.e., that even if S is selected in the original feature model none of the children has to be selected. We make S kind of optional, to ensure the semantics. Therefore we have to consider all possible cases.

The first case is that S is the root feature, i.e., S has no parent. In this case we additionally remove the constraint $S \wedge \ldots$ from the propositional formula (see Figure A.1).

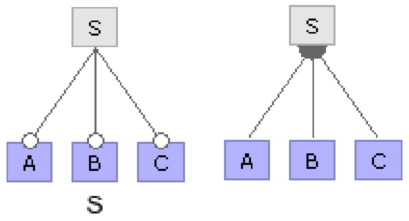


Figure A.1: S is the root feature (Case 1)

The second case happens if the parent connection of S is an *And* and S is an optional feature (see Figure A.2). We do not have to do anything, because S can already be empty.

The third case is common to the second case except that S is mandatory. We make S optional (see Figure A.3). But this can result in a new *And* without mandatory

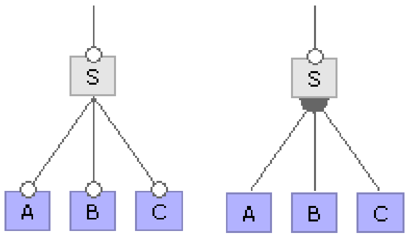


Figure A.2: S is optional (Case 2)

children, that we have eliminate this also. Be aware that we at least came on step closer to the root and we cannot come into an infinite loop.

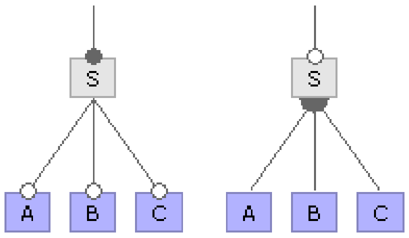


Figure A.3: S is mandatory (Case 3)

The fourth case is where S is contained in an *Or* or an *Alternative*. For both connections we have to recurse for the parent T of S to ensure, that the T can be empty. Figure A.4 shows the calculated refactoring before it is called recursive for T. Note that T can lead us to every case even if there is no connection shown in the figure.

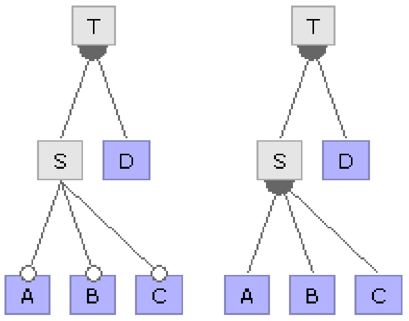


Figure A.4: S is contained in *Or* (or *Alternative*) (Case 4)

The reason for applying those refactorings is to get a definition for each code-less feature. Be aware of that a previously calculated definition might contain a code-less feature that we want to eliminate. This could result in a propositional formula where code-less features are not eliminated properly.

Therefore we apply the definition replacing algorithm first to the definitions itself, until all definitions are free of code-less features. This algorithm terminates because definitions are based on the feature diagram (tree) and we cannot have circles. Given the so calculated definitions we can eliminate all code-less features. The resulting formula can be used for reasoning as described in the previous chapters.

What is the complexity of the refactorings? To refactor a feature model as presented we need at most $O(n)$ recursions, if n is the number of features, because we can have at most $O(n)$ connections. The CNF of the resulting feature model has $O(r)$ disjunctions, where r is the number of disjunctions before refactoring. The reason is that *And* and *Or* both produce CNF clauses linear in the number of features in each connection (shown in Section 4.2).

B. Tables on Calculation Time

	Number of Features									
	100	200	300	400	500	600	700	800	900	1000
Refactoring										
Time	31	49	98	127	173	222	268	328	406	438
Iterations	308	209	241	181	281	290	222	285	338	245
Generalization										
Time	25	42	70	121	176	237	293	398	416	484
Iterations	237	100	134	193	281	308	301	379	326	351
Arbitrary Edits										
Time	28	50	83	126	184	236	286	391	452	500
Iterations	204	115	205	204	285	290	231	338	368	356
Stand-alone FMs										
Time	39	74	118	189	277	340	416	520	611	703
Iterations	201	178	155	295	330	325	348	293	236	173
Identical FMs										
Time	2	7	13	30	40	58	70	85	103	108
Iterations	255	137	144	211	173	282	198	182	199	70

Table B.1: Calculation Time in Relation to the Number of Features

	Number of Edits									
	10	20	30	40	50	60	70	80	90	100
Refactoring										
Time	451	512	567	633	674	712	754	820	848	877
Iterations	286	292	242	222	282	231	248	236	169	176
Generalization										
Time	488	499	495	477	493	518	499	506	509	515
Iterations	350	330	308	220	295	277	305	337	315	269
Arbitrary Edits										
Time	517	511	525	525	532	552	540	545	567	560
Iterations	281	326	299	307	279	292	272	288	212	256

Table B.2: Calculation Time in Relation to the Number of Edits

	Maximum Number of Children									
	10	20	30	40	50	60	70	80	90	100
Refactoring										
Time	27	76	185	359	663	1000	1611	2077	2820	4416
Iterations	186	391	466	639	725	996	638	995	1000	1000
Generalization										
Time	21	45	84	149	234	332	440	588	821	968
Iterations	178	321	639	634	661	949	958	817	975	988
Arbitrary Edits										
Time	27	57	125	238	442	730	987	1265	1822	2580
Iterations	148	286	464	597	646	353	635	656	640	640
Stand-alone FMs										
Time	83	110	177	378	567	801	1160	1497	2222	3052
Iterations	672	439	393	327	594	750	829	741	865	858
Identical FMs										
Time	2	2	2	2	2	2	2	2	3	3
Iterations	325	290	220	364	269	358	418	256	466	457

Table B.3: Calculation Time in Relation to the Maximum Number of Children

Bibliography

[AGM+06] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos José Pereira de Lucena. Refactoring Product Lines. In *GPCE '06: Proceedings of ACM SIGPLAN 5th International Conference on Generative Programming and Component Engineering*, pages 201–210, 2006.

[AMS04] Timo Asikainen, Tomi Männistö, and Timo Soininen. Using a Configurator for Modelling and Configuring Software Product Lines based on Feature Models. In *SPLC '04: Proceedings of Workshop on Software Variability Management for Product Derivation, Software Product Line Conference*, 2004.

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC '05: Proceedings of the 9th International Software Product Lines Conference*, pages 7–20, 2005.

[Ben07] David Benavides. *On the Automated Analysis of Software Product Lines Using Feature Models - A Framework for Developing Automated Tool Support*. PhD thesis, University of Seville, Spain, June 2007.

[Beu03] Danilo Beuche. *Composition and Construction of Embedded Software Families*. PhD thesis, Otto-von-Guericke University Magdeburg, Germany, December 2003.

[BHST04] Yves Bontemps, Patrick Heymans, Pierre-Yves Schobbens, and Jean-Christophe Trigaux. Semantics of Feature Diagrams. In *Proceedings of Workshop on Software Variability Management for Product Derivation (Towards Tool Support)*, August 2004.

[BLHM02] Don Batory, Roberto E. Lopez-Herrejon, and Jean-Philippe Martin. Generating Product-Lines of Product-Families. In *ASE '02: Proceedings of Automated Software Engineering Conference*, pages 81–92, 2002.

[BRCTS06] David Benavides, Antonio Ruiz-Cortés, Pablo Trinidad, and Sergio Segura. A Survey on the Automated Analyses of Feture Models. *JISBD '06: Jornadas de Ingeniería del Software y Bases de Datos*, 2006.

[BSR04] Don Batory, Jacob N. Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[BSTRC05] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. Using Java CSP Solvers in the Automated Analyses of Feature Models. In *GTTSE '05: Proceedings of Generative and Transformational Techniques in Software Engineering*, pages 399–408, 2005.

[BSTRC06] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. A First Step Towards a Framework for the Automated Analysis of Feature Models. In *Proceedings of Managing Variability for Software Product Lines: Working With Variability Mechanisms*, 2006.

[BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Using Constraint Programming to Reason on Feature Models. In *SEKE '05: Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering*, July 2005.

[CBB+03] Fei Cao, Barrett Bryant, Carol Burt, Zhisheng Huang, Rajeev Raje, Andrew Olson, and Mikhail Auguston. Automating Feature-Oriented Domain Analysis. In *SERP '03: Proceedings of International Conference on Software Engineering Research and Practice*, pages 944–949, June 2003.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[CHE05a] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing Cardinality-Based Feature Models and their Specialization. *Software Process: Improvement and Practice*, 10:7–29, 2005.

[CHE05b] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged Configuration through Specialization and Multi-Level Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[CK05] Krzysztof Czarnecki and Chang Hwan Peter Kim. Cardinality-Based Feature Modeling and Constraints: A Progress Report, October 2005.

[CKK06] Krzysztof Czarnecki, Chang Hwan Peter Kim, and Karl Trygve Kalleberg. Feature Models are Views on Ontologies. In *SPLC '06: Proceedings of the 10th International Conference on Software Product Lines*, pages 41–51, 2006.

[CW07] Krzysztof Czarnecki and Andrzej Wasowski. Feature Diagrams and Logics: There and Back Again. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference*, pages 23–34, 2007.

[dJV02] Merijn de Jonge and Joost Visser. Grammars as Feature Diagrams. In *ICSR '02: Proceedings of Workshop on Generative Programming*, pages 23–24, 2002.

[Fow00] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, February 2000.

[GFdA98] Martin L. Griss, John Favaro, and Massimo d' Alessandro. Integrating Feature Modeling with the RSEB. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, 1998.

[JK07] Mikolas Janota and Joseph Kiniry. Reasoning about Feature Models in Higher-Order Logic. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference*, pages 13–22, 2007.

[KC05] Chang Hwan Peter Kim and Krzysztof Czarnecki. Synchronizing Cardinality-Based Feature Models and Their Specializations. In *ECMDA-FA '05: Proceedings of the 1st European Conference on Model Driven Architecture - Foundations and Applications*, pages 331–348, 2005.

[KCH$^+$90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.

[KKL$^+$98] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Gerard Jounghyun Kim, and Euiseob Shin. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[LAMS05] Thomas Leich, Sven Apel, Laura Marnitz, and Gunter Saake. Tool Support for Feature-Oriented Software Development - FeatureIDE: An Eclipse-Based Approach. In *ETX '05: Proceedings of OOPSLA Eclipse Technology eXchange Workshop*, October 2005.

[Man02] Mike Mannion. Using First-Order Logic for Product Line Model Validation. In *SPLC '02: Proceedings of the Second Software Product Line Conference*, pages 176–187, 2002.

[Pre97] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer, 1997.

[RBSP02] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending Feature Diagrams with UML Multiplicities. In *IDPT '02: Proceedings of the 6th World Conference on Integrated Design & Process Technology*, June 2002.

[Rie03] Matthias Riebisch. Towards a more precise definition of feature models. In M. Riebisch, J. O. Coplien, and D. Streitferdt, editors, *Modelling Variability for Object-Oriented Product Lines*, pages 64–76. BookOnDemand Publ. Co, 2003.

[SHTB07] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic Semantics of Feature Diagrams. *Computer Networks*, 51(2):456–479, 2007.

[SRP03] Detlef Streitferdt, Matthias Riebisch, and Ilka Philippow. Details of Formalized Relations in Feature Models Using OCL. In *ECBS '03: Proceedings of 10th IEEE International Conference on Engineering of Computer–Based Systems, Huntsville, USA. IEEE Computer Society*, pages 45–54, 2003.

[STB$^+$04] Mirjam Steger, Christian Tischer, Birgit Boss, Andreas Müller, Oliver Pertler, Wolfgang Stolz, and Stefan Ferber. Introducing PLA at Bosch Gasoline Systems: Experiences and Practices. In *SPLC '04: Proceedings of the 3rd Software Product Line Conference*, pages 34–50, 2004.

[SZLW05] Jing Sun, Hongyu Zhang, Yuan Fang Li, and Hai Wang. Formal Semantics and Verification for Feature Modeling. In *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 303–312, June 2005.

[TBKC07] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe Composition of Product Lines. In *GPCE '07: Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, pages 95–104, 2007.

[TBRC$^+$06] Pablo Trinidad, David Benavides, Antonio Ruiz-Cortés, Sergio Segura, and Miguel Toro. Explanations for Agile Feature Models. In *APLE '06: Proceedings of the 1st International Workshop on Agile Product Line Engineering*, 2006.

[Tre05] Tim Trew. Enabling the Smooth Integration of Core Assets: Defining and Packaging Architectural Rules for a Family of Embedded Products. In *SPLC '05: Proceedings of the 9th International Software Product Lines Conference*, pages 137–149, 2005.

[vDK02] Arie van Deursen and Paul Klint. Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[vdLSR07] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[vGBS01] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. On the Notion of Variability in Software Product Lines. In *WICSA '01: Proceedings of Working IEEE/IFIP Conference on Software Architecture*, 2001.

[WLS+05] Hai Wang, Yuan F. Li, Jing Sun, Hongyu Zhang, and Jeff Pan. A Semantic Web Approach to Feature Modeling and Verification. In *SWESE '05: Proceedings of Workshop on Semantic Web Enabled Software Engineering*, November 2005.

[WSB+08] Jules White, Douglas C. Schmidt, David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated Diagnosis of Product-line Configuration Errors in Feature Models. In *SPLC '08: Proceedings of the 12th International Software Product Line Conference*, September 2008.

[ZZM04] Wei Zhang, Haiyan Zhao, and Hong Mei. A Propositional Logic-Based Method for Verification of Feature Models. *Formal Methods and Software Engineering*, pages 115–130, 2004.

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 20. Juni 2008