Bachelor Thesis
in Computer Science

Next-Generation Feature Models with Pseudo-Boolean SAT Solvers

Sebastian Henneberg

Advisor:
Dr.-Ing. Sven Apel

October 28, 2011

Abstract

Feature models are an important artifact in software product line engineering. They describe commonality and variability of all product line members. This thesis proposes the use of attributes and additional constraints in feature modeling to extend expressiveness and usability. Therefore, new grammars were built to extend traditional feature models by optional integer attributes and additional constraints. We found a mapping that converts extended feature models into pseudo-boolean satisfiability (PBSAT) instances. This allows reasoning of feature models using a PBSAT solver. We took different feature model analysis operations from several authors to show applicability of the PBSAT representation. This required adaptations and led to extensions of known algorithms. We analyzed the scalability of our proposed adaptations by an evaluation of different real-world feature models.

Contents

1 Introduction **7**

2 Background **10**
2.1 Feature models 10
2.2 SAT and PBSAT 11
2.3 Automated analysis 14
2.4 FeatureIDE 15

3 Extended feature models **17**
3.1 Attribute grammar 17
3.2 Constraint grammar 18
3.3 Generated parser 19

4 Mapping feature models to logic **21**
4.1 Feature diagram 21
4.2 Cross-tree constraints 22
4.3 Additional pseudo-boolean equations and inequalities 23

5 Analysis operations **26**
5.1 Domain engineering 26
5.1.1 Validation 27
5.1.2 Feature properties 28
5.1.3 Simplification 33
5.1.4 Edits 36
5.2 Application engineering 40
5.2.1 Propagation of selections 40

6 Evaluation **44**
6.1 Runtime behaviour 45
6.2 Simplification potential 45

6.3 Pre-processing impact 46

7 Conclusion **48**

7.1 Contributions 49

7.2 Further work 49

Bibliography **51**

List of Figures

2.1 Feature model of a mobile phone product line 10
2.2 Steps of automated resasoning 14
2.3 Screenshot of the feature modeling view in FeatureIDE 15

3.1 Grammar of the attribute file 18
3.2 Sample attribute file 18
3.3 Grammar of the constraint file 19
3.4 Sample constraint file 19
3.5 Alternative representation of sample constraint file 19

4.1 Subtree of a feature diagram 23
4.2 Sample attribute file according to 4.1 23

5.1 A *valid* and a *void feature model* 27
5.2 A feature model that contains features with different properties . . . 28
5.3 A feature model with emphasised *core features* 29
5.4 A feature model with emphasised *dead features* 30
5.5 A feature model with emphasised *variant features* 31
5.6 A feature model with emphasised *false optional features* 32
5.7 *Simplification* of a feature model with *atomic sets* 33
5.8 A *feature model edit* that is classified as *generalization* 36
5.9 Types of edits based on set inclusions or rather logical implications . 37
5.10 A *user selection* leads to a *propagation* 43
5.11 The propagation of 5.10 represented as graph 43

List of Tables

2.2 Feature diagram mappings to propositional logic 12

4.2 Feature diagram mappings to PBSAT 24
4.3 Additional constraint mapping according to 4.1 and 4.2 25

5.2 The four different types of BCP clauses 41
5.4 The four different types of BCP inequalities 42

6.1 Mean runtime of validation, property computation and propagation 45
6.2 Results of the simplification (atomic set) computation 46
6.3 Mean runtime of the simplification (atomic set) algorithm 47

List of Algorithms

1 Check validity of feature model 28
2 Compute set of core features 29
3 Compute set of dead features 30
4 Compute set of variant features 31
5 Compute set of false-optional features 33
6 Compute the atomic sets 35

1 Introduction

Software product lines are a recent research topic in software engineering. It is about producing a set of related software with similar core functionality but different peculiarity. Product lines are widely known from big manufacturer's of cars and computers or even fast-food restaurants. They all have in common that they offer a collection of related products which satisfy different customer needs. Product lines often enable customers to individualize products.

Apart from the advantage of smaller portfolios, product lines reduce costs and help to unify the development process. Especially larger software systems benefit from this technology. Software product lines accelerate development because of intense reuse of all software artifacts like components, frameworks, tests and documentation. Fewer developers and lower investments are major benefits concerning to resources. Furthermore, clients receive their products earlier and have more options to customize the resulting software.

Variability models are an important design artifact in software product-line engineering. They capture commonalities and variabilities of product lines to describe valid combinations of the available components. Several variability models have been proposed in the past. A well-known kind are *feature models* which organize different functionality or requirements in *features*.

Feature models are usually divided in two separate parts. The first part is the *feature diagram* that organises features in a tree-structure and provides basic relations. The second part are *additional constraints* that enable to define further relations between features. Additional constraints are often called *cross-tree constraints* since they do not rely on the tree-structure of the feature diagram. However, semantics of feature models are fix and precise. This allows to map feature models to different logical representations. A widely used representation is propositional logic since it allows to use *satisfiability* (SAT)[Bat05] solvers for reasoning. Other authors propose the

use of *description logic* (DL)[WLS+07] or *constraint programming* (CP)[BTRC05b]. In this thesis, we want to examine the feasibility of pseudo-boolean satisfiability (PBSAT) problems. PBSAT instances are systems of linear inequalities with two-value variables. Surprisingly, PBSAT instances can be handled similar to SAT instances in some cases. This thesis checks feasibility, discovers limitations and applies analysis based on feature models in PBSAT representations.

Chapter 2 provides explanations for the different topic-related terms in this thesis. Feature models will be introduced in detail using graphical examples. The similarities and differences between SAT and PBSAT will be examined. Furthermore, concepts of automated analysis and tool support in FeatureIDE will be presented.

Apart from mapping to logic representations, feature models have been extended by more sophisticated constructs like additional rules or attributes. Moreover, feature models can be specified using context-free languages. We combined both approaches and introduced attributes and additional constraints to extend features models. The limitations of the feasibility study and the resulting grammars are outlined in chapter 3.

The mapping of feature models to specific logic representations allows to reason with off-the-shelf solvers. Reasoning allows to check the integrity of the model using satisfiability algorithms. But feature model analysis also includes various more complicated operations. *Benavides et al.* [BSRC10] collected all known operations and compared the use of different reasoning engines. We recognized that pseudo-boolean satisfiability were not used yet. This motivated us to look for a mapping from extended feature models to pseudo-boolean satisfiability instances. Chapter 4 describes in detail the mapping procedure and emphasises advantages of the pseudo-boolean representation.

Understanding semantics or imagining all represented variants of product lines becomes difficult even for very small feature models. This is where tool support comes in to help developers comprehending. Additionally, automated analysis keeps track of changes and gives meaningful advices during refactoring. In chapter 5, we explain important analysis operations by definitions and graphical examples. Then we show the feasibility of these operations using a PBSAT solver. Therefore, we had to adapt most of them into the pseudo-boolean context. The algorithms are

divided in two separate sections because the analysis takes place in different stages of the development.

We created many test cases to verify the correctness of the adapted algorithms. We also collected feature models from the FeatureIDE repository[1] to apply the proposed analysis. Chapter 6 explains our evaluation which we used to measure time consumption in dependency to feature model size. The results of this evaluation are elaborated and commented.

We finally conclude this thesis in chapter 7 by summarising the different research steps and emphasising our contributions to the software product-line community. Additionally, we mention several related topics to investigate that build on top of the results from this thesis.

[1] `https://faracvs.cs.uni-magdeburg.de/projects/tthuem-FeatureIDE/browser/trunk/featuremodels`

2 Background

The background chapter helps to understand important terms and concepts of this thesis. Feature models as well as automated reasoning will be explained. SAT and PBSAT instances will be introduced and compared. In addition, we present important conversions that will be used in later chapters.

2.1 Feature models

As already mentioned in the introduction, feature models are a specialization of variability models. Feature models describe a set of similar products using features to encapsulate special customer needs like additional functionality or individual requirements. The feature model captures relations between features and hence defines legal combinations of them. The hierarchy and basic relations are provided by the feature diagram. Besides, feature diagrams also visualize components and give impressions of the complexity and size. Additional constraints are used to complement missing relations. Figure 2.1 shows a feature model using both, a feature diagram and additional constraints. The latter denoted below the feature diagram using propositional logic.

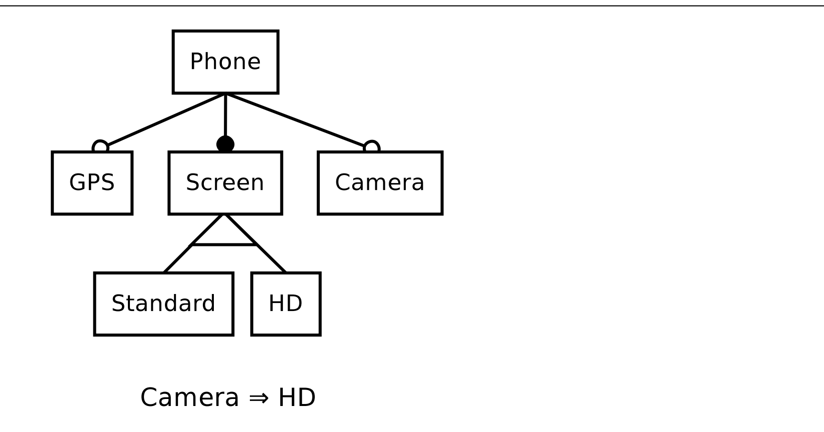


Figure 2.1: Feature model of a mobile phone product line

The complete feature model can be represented using propositional formulas. The tree-structure of the feature diagram can be mapped to propositional logic as shown in table 2.2. The four presented relations and the obligatory selected root feature are known from basic feature models. Cardinality-based feature models are not supported by the GUIDSL[1] format we use. But our proposed mapping is able to express cardinality-based feature models. We will introduce custom cardinality groups later. We shortened the propositional logic representation by using implications and high-level functions like $atLeast(n, X)$ and $atMost(n, X)$. Both functions limit the number of `true`-assigned variables in the set X by a lower or an upper bound n. An equivalent but very verbose representation in conjunctive normal form (CNF)[KK06] is shown by the equations 2.1 and 2.2.

$$atLeast(n, \underbrace{\{x_1, \cdots, x_i\}}_{=X}) \equiv \bigwedge_{C=\mathcal{P}_{=n+1}(X)} \bigvee_{l \in C} \neg l \tag{2.1}$$

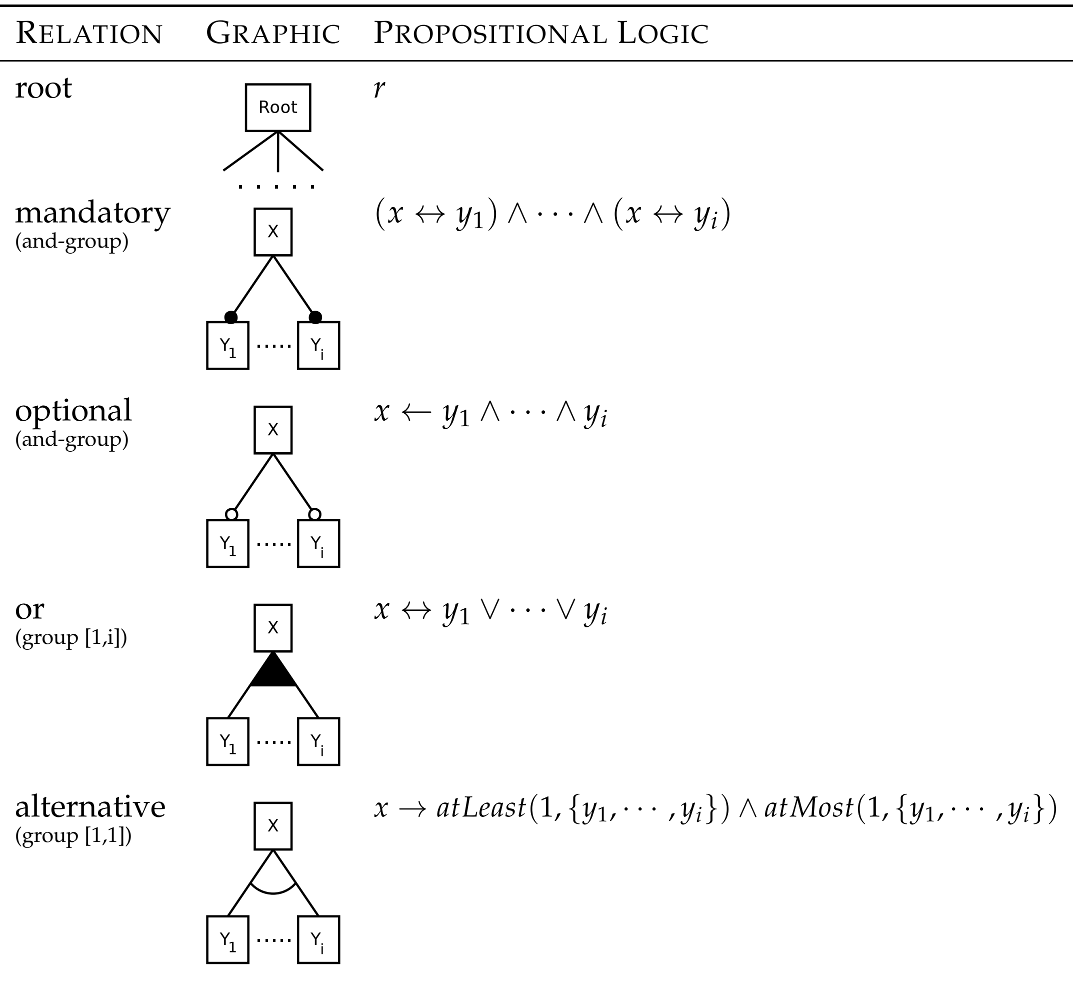
$$atMost(n, \underbrace{\{x_1, \cdots, x_i\}}_{=X}) \equiv \bigwedge_{C=\mathcal{P}_{=i-n+1}(X)} \bigvee_{l \in C} l \tag{2.2}$$

2.2 SAT and PBSAT

Feature models can be expressed in very different ways. The usual representations are logic-based. Propositional logic[Bat05] is the most common type since it enables straightforward transformation[BSRC10] as visualized in table 2.2. The propositional formula describing the feature model can easily be converted using known rules of boolean algebra. Therefore, conversion into different forms like the CNF are possible. Particularly the CNF is very important for analysis of feature models. It is used as input for SAT solvers and hence allows to determine legal feature combinations. We just have to create a mapping from features to boolean variables and then verify satisfiability using the SAT solver. Notice that the number of satisfying assignments in the CNF is equal to the number of variants because features are directly represented by the boolean variables.

The idea of PBSAT solvers is similar to SAT solvers. Both try to find satisfying assignments for boolean variables in a given problem instance. The major differ-

[1] `http://www.cs.utexas.edu/~schwartz/ATS/fopdocs/guidsl.html`



Relation	Graphic	Propositional Logic
root	Root	r
mandatory (and-group)	X, Y_1 … Y_i	$(x \leftrightarrow y_1) \wedge \cdots \wedge (x \leftrightarrow y_i)$
optional (and-group)	X, Y_1 … Y_i	$x \leftarrow y_1 \wedge \cdots \wedge y_i$
or (group [1,i])	X, Y_1 … Y_i	$x \leftrightarrow y_1 \vee \cdots \vee y_i$
alternative (group [1,1])	X, Y_1 … Y_i	$x \rightarrow atLeast(1, \{y_1, \cdots, y_i\}) \wedge atMost(1, \{y_1, \cdots, y_i\})$

Table 2.2: Feature diagram mappings to propositional logic

ence is the input format. PBSAT uses a system of linear inequalities with integer coefficients. The variables are pseudo-boolean because they are either 0 (`false`) or 1 (`true`). A sample instance is shown in equation 2.3. It represents the phrase: "If and only if a is `true`, exactly two of b, c and d have to be `true` too". Otherwise, the equation can not hold.

$$2a - 1b - 1c - 1d = 0 \tag{2.3}$$

The PBSAT representation allows to negate boolean variables as in propositional logic. We denote negated variables by an $\overline{\text{overline}}$. Combining this possibility with the property of pseudo-boolean variables allows to replace variables as shown in equation 2.4[BHP08].

$$\overline{x} = 1 - x \tag{2.4}$$

The replacement of equation 2.4 enables to transform all variables positive/negative or all coefficients positive/negative. We exemplary show how to flip all coefficients positive. Therefore, we apply the replacement three times in 2.3 and get equation 2.5.

$$2a + 1\overline{b} + 1\overline{c} + 1\overline{d} = 3 \tag{2.5}$$

We introduced PBSAT as system of linear **inequalities** but obviously presented just linear equations yet. But it is trivial to express an equation in a conjunction of inequalities. The conjunction 2.6 shows an equivalent statement as equations 2.3 using inequalities instead of equations. We will use equations as much as possible throughout this thesis to simplify and shorten representations by keeping in mind that the PBSAT solver just accepts linear inequalities.

$$2a - 1b - 1c - 1d \leq 0 \quad \wedge \quad 2a - 1b - 1c - 1d \geq 0 \tag{2.6}$$

Notice that we place variables always on the left-hand side of equations or rather inequalities. The integer number on the right-hand side of the equations or inequality is called the *degree*.

PBSAT allows to express a lot problems more compact than propositional logic. In the worst case, the PBSAT representation is as verbose as the propositional logic instance in CNF because we are able to map each CNF clause by a *cardinality restriction*[CK03]. The equivalence 2.7 shows the mapping.

$$(a \vee \neg b \vee c) \quad \Leftrightarrow \quad 1a + 1\overline{b} + 1c \geq 0 \tag{2.7}$$

The referenced literature calls pseudo-boolean equations and inequalities *constraints*. Unfortunately, the word constraint is already used to describe additional rules in feature models. That is why we call pseudo-boolean equations or inequalities *restrictions* throughout this thesis to avoid misunderstandings. The general mathematical notation of arbitrary pseudo-boolean restriction can be seen in equation 2.8. We will use this notation later to explain necessary conversions.

$$\sum_{i=1}^{n} a_i x_i = d \tag{2.8}$$

2.3 Automated analysis

Automated analysis plays an important role during feature modeling and product derivation. It supports developers with useful information about the product line and the represented variants. The collected information and the applied analysis depends on the concrete tool. Some of them just generate statistical information like homogeneity or the number of potential variants. Other tools focus on validation and simplification of feature models as well as classification of feature model edits. However, automated analysis provides different information to improve feature models. This can be done by providing meaningful advices and reporting warnings and errors. Integrated in feature model editors or development environments, automated analysis supports faster development and causes fewer problems. Figure 2.2 illustrates the usual work flow of automated analysis.

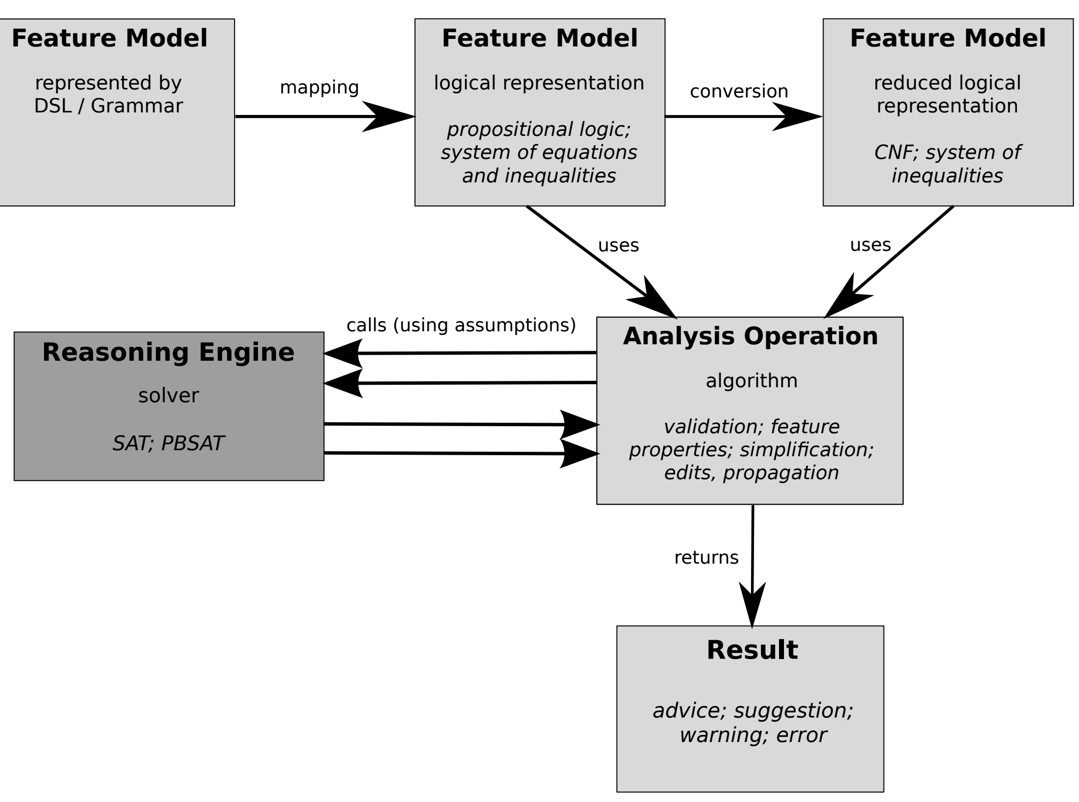


Figure 2.2: Steps of automated resasoning

2.4 FeatureIDE

FeatureIDE[2] is an open-source, integrated development environment (IDE) for feature-oriented software development. It provides feature modeling support and several established extensions for program composition. It is the only tool that delivers concrete support for program composition. Other tools like *pure::variants* from pure-systems[3] focus on variant management which also supports product line development for non-software projects. Certainly, no extensions for intelligent program composition are delivered with pure::variants. Both tools have in common that they are build on top of the Eclipse RCP[4].

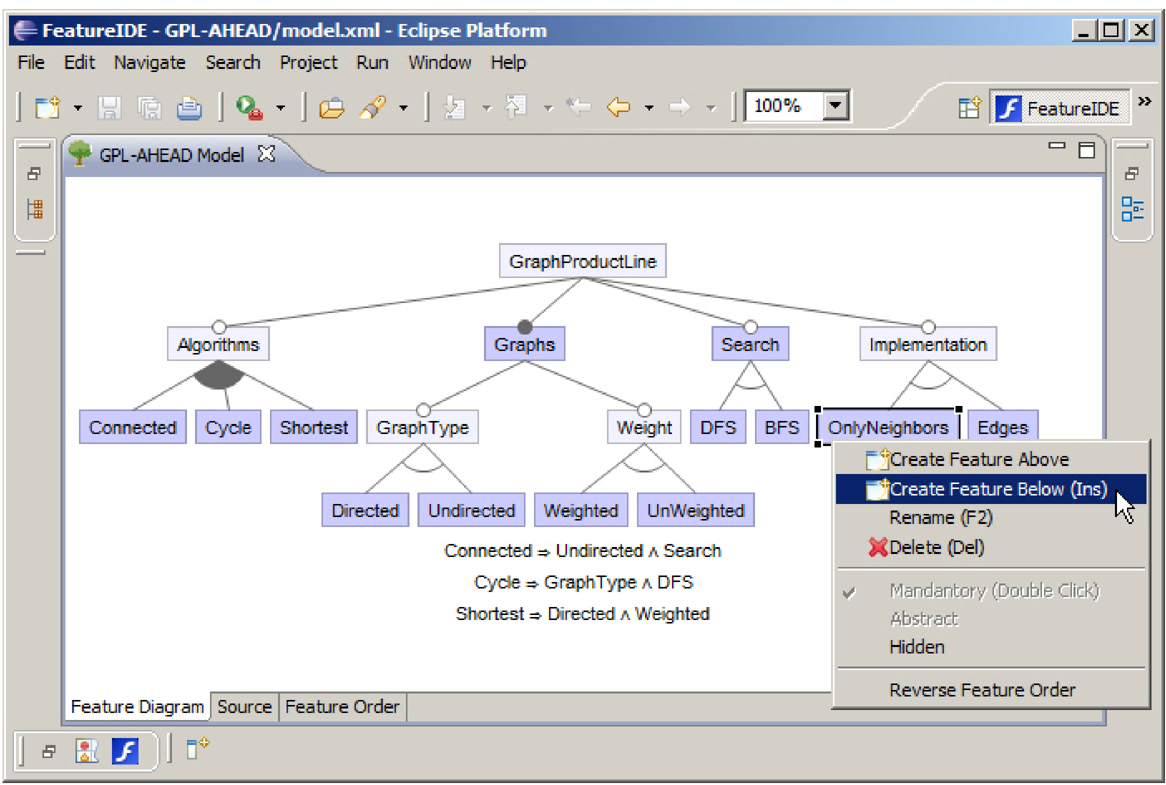


Figure 2.3: Screenshot of the feature modeling view in FeatureIDE

There are more projects that support feature modeling and/or automated analysis. But most projects concentrate either on feature modeling or automated analysis. We looked for a tool that combines both to improve development and enhance user

[2] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/
[3] http://www.pure-systems.com/
[4] http://www.eclipse.org/home/categories/rcp.php

experience. FeatureIDE fulfils this requirements. Therefore, our proposed formats and algorithms are fully compatible with FeatureIDE. Furthermore, we plan to extend the FeatureIDE platform by the results of this thesis. Figure 2.3 shows the feature model editor of FeatureIDE. It provides a comfortable editing interface that gives immediately feedback if the user modifies components such that inconsistencies arise.

3 Extended feature models

Traditional feature models are usually represented by propositional formulas [Bat05, BSRC10]. Feature diagrams are used to create a hierarchy and visualize the connection of features. Cross-tree constraints complement the missing expressiveness and define relations between features in different subtrees of the feature diagram. *Extended feature models* provide different ways to encode even more information into feature models [BTRC05a]. We focussed on optional attributes which have to be attached to features. The attached attributes are able to describe non-functional properties as well as behaviour or compatibility of features. Our feasibility analysis proved that we are able to express attributes in the integer domain using a PBSAT solver. Since attributes do not have pre-defined semantics, we added additional constraints that are able to refer features and feature attributes. These additional constraints are represented as pseudo-boolean restrictions which allow reasoning using a PBSAT solver.

Batory discovered the relation between feature models, grammars and propositional formulas [Bat05]. Based on his results, we built new grammars which express feature attributes and additional constraints. The feature models are represented by the GUIDSL language. However, the proposed extension of feature attributes and additional constraints is independent of the concrete feature model representation. Other representations can be plugged in to extend expressiveness and apply several analysis operations.

3.1 Attribute grammar

Feature attributes have to be defined in a special file. This file contains all attribute assignments in a JAVA-like syntax. Features are able to have several attributes which are identified by a unique name. The value of the attributes is limited to

integer values because of the PBSAT reasoning engine. The grammar of the attribute file is described using the extended Backus-Naur Form (EBNF) in figure 3.1.

$$
\begin{aligned}
assignments &= \{assignment\}; \\
assignment &= feature, \textbf{'.'}, attribute, \textbf{'='}, integer, \textbf{';'}; \\
feature &= identifier; \\
attribute &= identifier; \\
identifier &= (\textbf{'a-z'}|\textbf{'A-Z'}|\textbf{'_'}), \{ (\textbf{'a-z'}|\textbf{'A-Z'}|\textbf{'_'}|\textbf{'0-9'}) \}; \\
integer &= [(\textbf{'-'}|\textbf{'+'})], (\textbf{'1-9'}), \{ (\textbf{'0-9'}) \};
\end{aligned}
$$

Figure 3.1: Grammar of the attribute file

The listing in figure 3.2 shows an example attribute file. It extends the mobile phone feature model from 2.1 by feature attributes that describe the cost of the different components.

```
Phone.cost    = 25;
GPS.cost      = 30;
Standard.cost = 20;
HD.cost       = 55;
Camera.cost   = 35;
```

Figure 3.2: Sample attribute file

3.2 Constraint grammar

Additional constraints are also defined in a separate file. The file contains pseudo-boolean restrictions which refer features and/or attributes. Additionally, integer coefficients can be used. The constraint grammar is consistent to the attribute grammar because attributes are referred the same way. Figure 3.3 describes the constraint file grammar using EBNF.

Listing 3.4 shows how the attributes of listing 3.2 can be combined in an additional constraint that limits the total cost. In other words, the constraint forces the selection of cheap mobile phone variants.

$$
\begin{aligned}
constraints &= \{constraint\}; \\
constraint &= terms,\ relation,\ degree,\ \textbf{';'}; \\
terms &= firstTerm,\ termsTail; \\
termsTail &= \{term\}; \\
firstTerm &= [\,sign\,],\ uinteger,\ [\,negative\,],\ reference; \\
term &= sign,\ uinteger,\ [\,negative\,],\ reference; \\
reference &= feature,\ [\,[\,(\textbf{'.'}|\textbf{'\#'})\,],\ attribute\,]; \\
degree &= [\,sign\,],\ uinteger; \\
negative &= \textbf{'\sim'}; \\
relation &= (\textbf{'>='}|\textbf{'='}|\textbf{'<='}); \\
sign &= (\textbf{'-'}|\textbf{'+'}); \\
identifier &= (\textbf{'a-z'}|\textbf{'A-Z'}|\textbf{'_'}),\ \{\ (\textbf{'a-z'}|\textbf{'A-Z'}|\textbf{'_'}|\textbf{'0-9'})\ \}; \\
uinteger &= (\textbf{'1-9'}),\ \{\ (\textbf{'0-9'})\ \};
\end{aligned}
$$

Figure 3.3: Grammar of the constraint file

```
Phone + GPS.cost + Standard.cost + HD.cost + Camera.cost <= 99;
```

Figure 3.4: Sample constraint file

We included a special operator to the constraint file grammar. It allows to build sums of feature attributes like in listing 3.4. Listing 3.5 shows the usage of that shortcut. Notice that the constraint in listings 3.4 and 3.5 is indentical. The operator will be replaced by the sum of all subfeatures with the same attribute.

```
Phone#cost <= 99;
```

Figure 3.5: Alternative representation of sample constraint file

3.3 Generated parser

We used the ANTLR[1] parser generator to build the required attribute and constraint file parsers. Therefore, we created attributed grammars to generate the parser for

[1] http://www.antlr.org/

both file formats. We decided to skip the concrete grammar files here to focus on the general approach.

4 Mapping feature models to logic

Automated analysis of feature models needs a platform which enables reasoning. We decided to use a PBSAT solver as reasoning engine. It takes PBSAT instances and checks satisfiability which builds the base for most analysis operations.

We examined the form of PBSAT problems in chapter 2. It is basically a conjunction of linear equations and inequalities where each variable is either 0 or 1. An equivalent description would be a system of linear inequalities with limited solution sets, namely $\{0, 1\}$ for each variable x_i.

The mapping is responsible to somehow translate a feature model into a problem instance. The instance type depends on the required input of the reasoning engine. Since we use PBSAT solvers, we have to find a method that is able to convert a feature model into a PBSAT instance. Our proposed mapping method is very modular. We found a way to translate different components of the feature model into individual PBSAT instances. Therefore, we divided the feature model in the feature diagram component, the cross-tree constraints and additional pseudo-boolean equations and inequalities. Then we mapped each of them into sub-instances of PBSAT and joined them together to obtain the PBSAT instance that represents the whole feature model.

4.1 Feature diagram

The mapping of feature relations in the feature diagram has been shown by many authors [Bat05, CW07, BSRC10]. We applied a similar technique treating groups of features independent from each other. Furthermore, we focussed on having the most compact PBSAT representation for each group. Table 4.2 shows the mapping of all known feature diagram relations to propositional logic and PBSAT. Note that *mandatory* or *optional* features in *and-groups* allow separate translation. Notice also

that groups can be generalized using the mapping of cardinality groups. We extended the table by propositional logic mappings to emphasise the compactness of the pseudo-boolean representation. The size of both representations seems to be similar, but the conjunctive normal forms of the propositional logic mappings are more verbose. Especially the high-level functions $atLeast(n, X)$ and $atMost(n, X)$ cause much clauses depending on the parameters. The equivalent CNF representation of $atLeast$ and $atMost$ is shown in equations 2.1 and 2.2.

4.2 Cross-tree constraints

In some feature model specifications, cross-tree constraints are limited sets of predefined relations [BCTS06, pur09] between features which are not directly connected in the feature diagram. However, we think of cross-tree constraints as arbitrary propositional formulas like in GUIDSL [Bat05] or rather FeatureIDE [KTS+09]. This enables the application of our mapping on all specialized feature model specifications. To be more precise, we are able to apply the proposed mapping on each representation that is reducible to propositional logic.

The mapping itself is very simple and analogous to the SAT-based approach. We convert all propositional formulas into conjunctive normal form (CNF)[KK06] and create an equivalent cardinality restriction for each clause as shown in chapter 2. The conversion from an arbitrary propositional formulas can lead to an exponential explosion of clauses in the resulting CNF. However, cross-tree constraints commonly just affect a few features which guarantees quick conversions into CNF.

The example equations 4.1 shows the conversion from a cross-tree constraint which forces the selection of C if and only if A and B are selected. After the conversion into CNF, each clause is mapped to a cardinality restriction.

$$
\begin{aligned}
& (A \wedge B) \leftrightarrow C \\
\equiv \quad & ((A \wedge B) \rightarrow C) \wedge (C \rightarrow (A \wedge B)) \\
\equiv \quad & (\neg(A \wedge B) \vee C) \wedge (\neg C \vee (A \wedge B)) \\
\equiv \quad & \underbrace{(\neg A \vee \neg B \vee C)}_{1\bar{a}+1\bar{b}+1c \geq 1} \wedge \underbrace{(\neg C \vee A)}_{1\bar{c}+1a \geq 1} \wedge \underbrace{(\neg C \vee B)}_{1\bar{c}+1b \geq 1}
\end{aligned}
\tag{4.1}
$$

4.3 Additional pseudo-boolean equations and inequalities

In chapter 3, we introduced additional constraints based on pseudo-boolean restrictions. These restrictions provide the declaration of equations and inequalities that allow to refer features, attributes or even attribute sums. The mapping to the PBSAT representation is simple because of the identical form. We just have to replace feature names by boolean variables and attribute references by their assigned values. The sum operator requires a replacement by all feature attributes of the referred feature subtree according to the feature diagram. Table 4.3 shows the mapping of features, feature attributes and sums of feature attributes from the syntax described in chapter 3. The referred features and attributes are shown in figure 4.1 and 4.2. The row INTERPRETATION describes the logical consequence of the exemplary restrictions to ease comprehension.

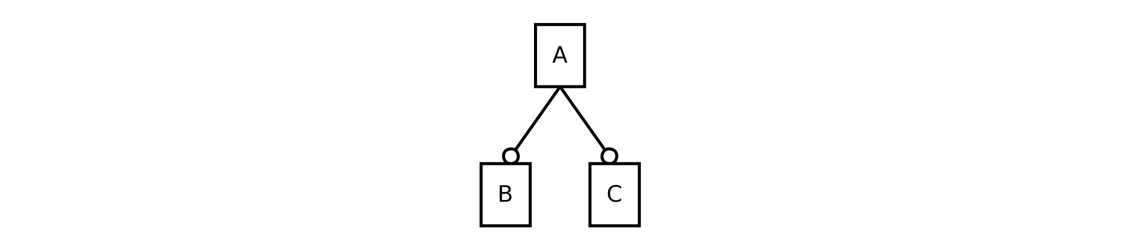


Figure 4.1: Subtree of a feature diagram

```
A.foo = 15;
B.foo = 6;
C.foo = 7;
```

Figure 4.2: Sample attribute file according to 4.1

Relation	Graphic	Propositional Logic	PBSAT
root	Root	r	$r \geq 1$
mandatory (single)	X, Y	$x \leftrightarrow y$	$x - y = 0$
mandatory (in and-group)	X, Y_1 ..... Y_i	$x \leftrightarrow y_1 \wedge \cdots \wedge y_i$	$i \cdot x - y_1 \cdots - y_i = 0$
optional (single)	X, Y	$y \rightarrow x$	$x - y \geq 0$
optional (in and-group)	X, Y_1 ..... Y_i	$y_1 \vee \cdots \vee y_i \rightarrow x$	$i \cdot x - y_1 \cdots - y_i \geq 0$
or (group [1,i])	X, Y_1 ..... Y_i	$x \leftrightarrow y_1 \vee \cdots \vee y_i$	$0 \leq i \cdot x - y_1 \cdots - y_i \leq i - 1$
alternative (group [1,1])	X, Y_1 ..... Y_i	$x \rightarrow atLeast(1, \{y_1, \cdots, y_i\}) \wedge atMost(1, \{y_1, \cdots, y_i\})$	$x - y_1 \cdots - y_i = 0$
custom cardinality (group [n,m])	X, [n,m], Y_1 ..... Y_i	$x \rightarrow atLeast(n, \{y_1, \cdots, y_i\}) \wedge atMost(m, \{y_1, \cdots, y_i\})$	$x \rightarrow (n \leq y_1 + \cdots + y_i \leq m)$ $\Leftrightarrow i - m \leq i \cdot x - y_1 \cdots y_i \leq i - n$

Table 4.2: Feature diagram mappings to PBSAT

TYPE	SYNTAX	PBSAT	INTERPRETATION
feature	`A >= 1`	$a \geq 1$	A is enabled
feature attribute	`A.foo <= 10`	$a \leq 10$	A is disabled, B and C too
feature attribute sum	`A#foo >= 20`	$10a + 6b + 7c \geq 20$	A is enabled, B and/or C too

Table 4.3: Additional constraint mapping according to 4.1 and 4.2

5 Analysis operations

Automated reasoning is a recent topic in software product line research. In the past, various tools were used to reason on feature models. Satisfiability solver (SAT), constraint satisfaction problem (CSP) and description logic (DL) were popular techniques [BSRC10]. But higher level tools like Prolog [BPSP04] or data structures like binary decision diagrams (BDD) [BSTC07] were also used. This chapter shows how PBSAT solvers can be used to reason on feature models. Therefore, several important analysis operations from different authors have been adapted to reason on extended feature models in pseudo-boolean representation. The different analysis operations will be introduced with raising complexity. The operations are separated by the stage of the engineering process they occur.

We added pseudo-code listings to describe the way the algorithms work. Therefore, some variables and functions have to be introduced. The parameter M represents a feature models which is already translated into a conjunction of pseudo-boolean restrictions. The parameter F denotes the set of all features present in the model M. The function $addAssumption()$ takes a model and an assumption (simple restriction) and returns a new model extended by the specified assumption. The function $removeAssumptions()$ removes all previously added assumptions. Finally, the function $satisfiable()$ symbolizes the call to the pseudo-boolean satisfiability solver. The call to $satisfiable()$ returns `true` if and only if there is a satisfying assignment for the passed PBSAT instance representing the feature model.

5.1 Domain engineering

We divided this chapter in the section *domain engineering* and *application engineering*. Both terms describe a specific process in software product-line engineering. We categorized the algorithms in these stages to emphasize the environment they usually occur.

Domain engineering describes the process of collecting individual customer needs and assembling them into a feature model. In this stage, the domain engineer creates and refactors the feature model. The various suggested algorithms support domain engineering by verifying feature models and reporting potential damages. Furthermore, changes on the model can be observed to preserve semantics. All these operations can run in background to inform the domain engineer if necessary. Automated reasoning in the domain engineering process eases development in software product line engineering and increases productivity.

5.1.1 Validation

Definition: A *valid* feature model requires that concrete products can be derived from the specification. If there is no derivable product, the model is called *void*. The validation is a basic reasoning operation since it guarantees that the model is consistent. Furthermore, a valid feature model is a pre-condition for deeper analysis which is explained later in this chapter.

Figure 5.1 illustrates the difference between valid and void feature models. Adding the cross-tree constraint $\neg A \wedge B$ makes the model invalid because both features are forced to be enabled cause of mandatory feature diagram relations.

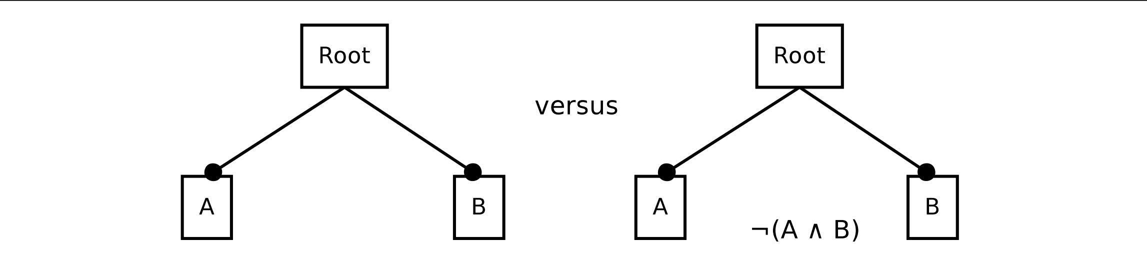


Figure 5.1: A *valid* and a *void feature model*

The validation algorithm [1] is simple and straightforward. We take the feature model M which is represented as conjunction of pseudo-boolean restrictions call the PBSAT solver on this representation. If the solver returns `false` which means "unsatisfiable", no solution exists and hence no product. Otherwise, the model is valid and describes real products.

Algorithm 1 Check validity of feature model

```
function isValid(M)
    return satisfiable(M)
end function
```

5.1.2 Feature properties

Many authors covered the properties that arise from the composition of features in their corresponding model [BSRC10]. They found out that some features are part of every product. At least the root feature fulfils this condition all the time. Sometimes, features can not be enabled during application engineering and consequently are never part of any derived product. In the following section, we introduce four different feature properties, give a proposal how to compute them, and examine the strong relation between them. In figure 5.2, we introduce a feature model with different feature properties. In the following section, this feature model will help to understand the described properties and to comprehend the proposed algorithms.

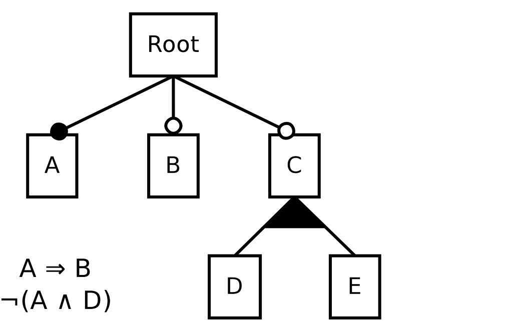


Figure 5.2: A feature model that contains features with different properties

Core features

Definition: A *core feature* is a component which is part in all products described by the product line. Core features can not be disabled cause of relations and/or constraints in the feature model. The root feature is obviously a core feature in every feature model.

Figure 5.3 emphasises the core feature in our example feature model. It is obvious that *Root* and the mandatory connected *A* are core features. Feature *B* is also a

core feature because of the cross-tree constraint $A \Rightarrow B$. The remaining features are not contained in the set of core features.

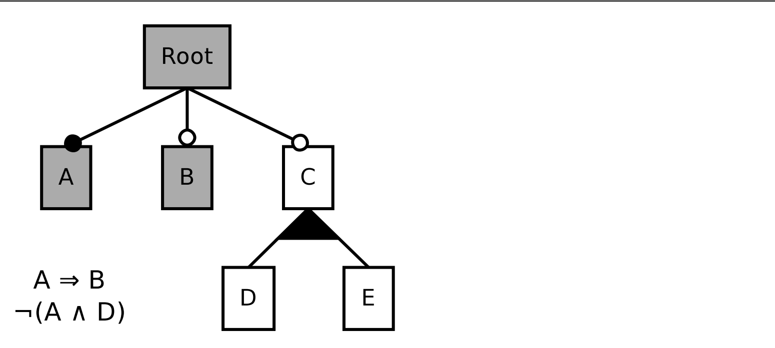


Figure 5.3: A feature model with emphasised *core features*

The core feature algorithm [2] takes a feature model M and a set F of all features present in the model. For each feature, we assume that it is disabled by adding an assumption to the model. If the PBSAT solver tells us that the feature is "unsatisfiable", we know the feature has to be part in any valid product. In other words, there is no product without the contemplated feature.

Algorithm 2 Compute set of core features

function COMPUTECOREFEATURES(M, F)
  $F_{core} \leftarrow \emptyset$
  for all $f \in F$ **do**
    $M \leftarrow addAssumption(M, f = 0)$ ▷ forces feature f to be disabled
    if $\neg satisfiable(M)$ **then**
      $F_{core} \leftarrow F_{core} \cup \{f\}$
    end if
    $M \leftarrow removeAssumptions(M)$ ▷ remove the assumption
  end for
  return F_{core}
end function

Dead features

Definition: A *dead feature* is a component which can not be part in any product although it is part of the feature model. Dead features occur because of additional constraints which force them to be disabled.

Benavides et al. [BSRC10] stated dead features as anomaly. In our opinion, dead features are not always problematic. They could be used to temporarily deactivate specific functionality. Another use case could be in staged configuration[CHE05], where previous (de-)selections by another party can not be undone. This leads to disabled or rather deselected features which can be interpreted as dead features.

Figure 5.4 shows a dead feature in the example feature model. Feature D is dead because of the cross-tree constraint $\neg(A \wedge D)$ and the fact that A is a core feature.

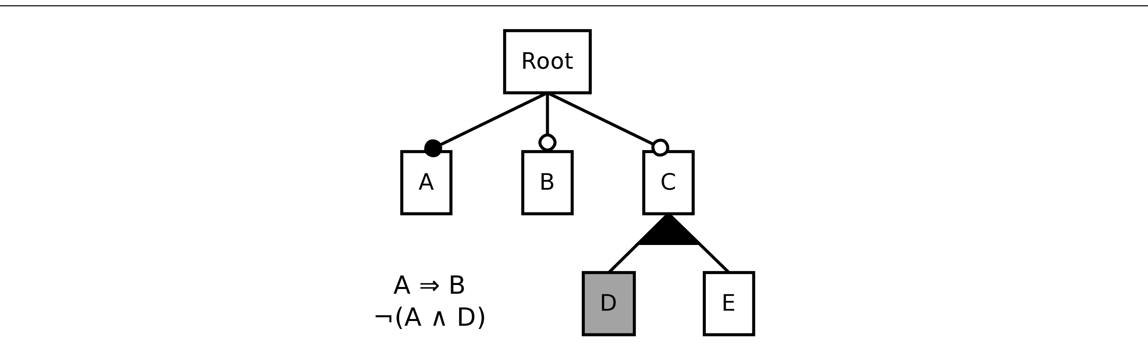


Figure 5.4: A feature model with emphasised *dead features*

If one selects a dead feature, the feature model becomes invalid. This characteristic explains the correctness of our iteratively assumption-based algorithm [3]. Notice, the computation of dead features is analogical to the core feature algorithm. The only difference are the temporarily added assumptions.

Algorithm 3 Compute set of dead features

function COMPUTEDEADFEATURES(M, F)
  $F_{dead} \leftarrow \emptyset$
  for all $f \in F$ **do**
    $M \leftarrow addAssumption(M, f = 1)$ ▷ forces feature f to be enabled
    if $\neg satisfiable(M)$ **then**
      $F_{dead} \leftarrow F_{dead} \cup \{f\}$
    end if
    $M \leftarrow removeAssumptions(M)$ ▷ removes the assumption
  end for
  return F_{dead}
end function

Variant features

In our opinion, there is no satisfying definition in the literature. *Benavides et al.* [BSRC10] defines variant features as features that do not appear in all products of a software product line. We propose a stronger definition of variant features. In our mind, features can either be *core*, *dead* or *variant*. This definition separates dead and variant features into two disjoint sets.

Definition: A *variant feature* describes a feature model component that can be enabled or disabled. Variant features extend the variability of the feature model, because their occurrence increases the number of potential products.

Figure 5.5 highlights all variant feature in the example feature diagram. The features C and E are variant because they are neither core nor dead features.

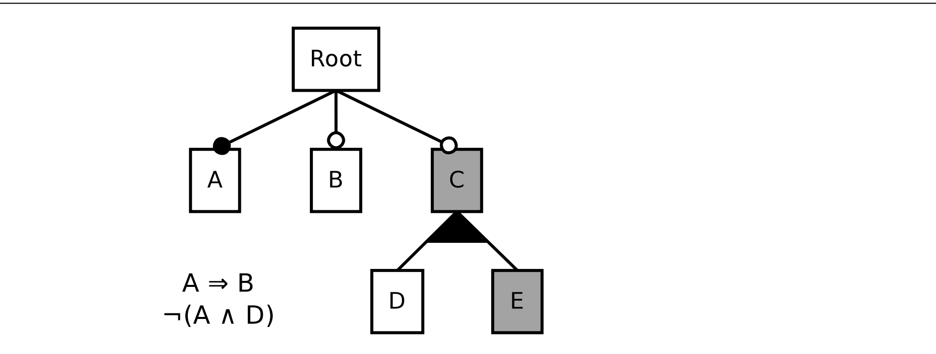


Figure 5.5: A feature model with emphasised *variant features*

The variant feature algorithm [4] takes the feature model M and a set of all features F. After computing core and dead features, the variant features can easily be determined. Variant features are those features that are neither core nor dead.

Algorithm 4 Compute set of variant features

function COMPUTEVARIANTFEATURES(M, F)
  $F_{core} \leftarrow computeCoreFeatures(M, F)$
  $F_{dead} \leftarrow computeDeadFeatures(M, F)$
  $F_{variant} \leftarrow F \setminus (F_{core} \cup F_{dead})$
  return $F_{variant}$
end function

Notice that the previously introduced properties are strongly related to each other. The computation can be combined in just one loop calling the satisfiability solver at all for $\mathcal{O}(2n)$ times where n denotes the number of features. Additionally, the following equations hold:

$$F = F_{core} \cup F_{dead} \cup F_{variant} \tag{5.1}$$

$$|F| = |F_{core}| + |F_{dead}| + |F_{variant}| \tag{5.2}$$

False-optional features

Definition: A *false-optional feature* is a component that seems to be optional although it is part of all valid products. False-optional features are obviously a subset of core features. Making false-optional features visible helps to restructure feature models while semantics are preserved.

Figure 5.6 emphasises the only false-optional feature in the example model. Feature B is core but at the same time not mandatory connected to the parent feature *Root* which makes it false-optional.

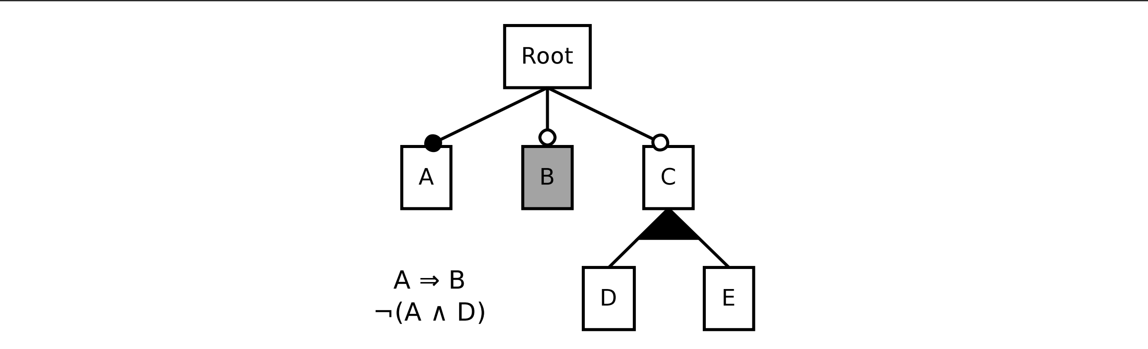


Figure 5.6: A feature model with emphasised *false optional features*

At first, the false-optional algorithm [5] determines all core features. After that, the core features will be checked whether they are optional or not. This is done by the $isOptional()$ function. The result are all core features in a optional relation which fits exactly the above definition.

Algorithm 5 Compute set of false-optional features

function COMPUTEFALSEOPTIONALFEATURES(M, F)
  $F_{core} \leftarrow computeCoreFeatures(M, F)$
  $F_{falseOpt} = \emptyset$
  for all $f \in F_{core}$ **do**
    if $isOptional(f)$ **then**
      $F_{falseOpt} \leftarrow F_{falseOpt} \cup \{f\}$
    end if
  end for
  return $F_{falseOpt}$
end function

5.1.3 Simplification

In almost all feature models, there are groups of features which can be handled as a single feature. In other words, if you select one feature of such a group, the selection will be propagated to all remaining features of the group. These groups are better known as atomic sets[ZZM04]. Each atomic set can be handled like a single feature in the mapping procedure. This enables simplification through replacement of features or rather variables. The consequence are fewer features and a more compact representation. Furthermore, atomic sets make the real variability of a feature model visible.

Figure 5.7 illustrates how a feature model can be simplified by replacement of strongly connected features. Notice that the cross-tree constraint $B \Leftrightarrow C$ is considered in the simplification.

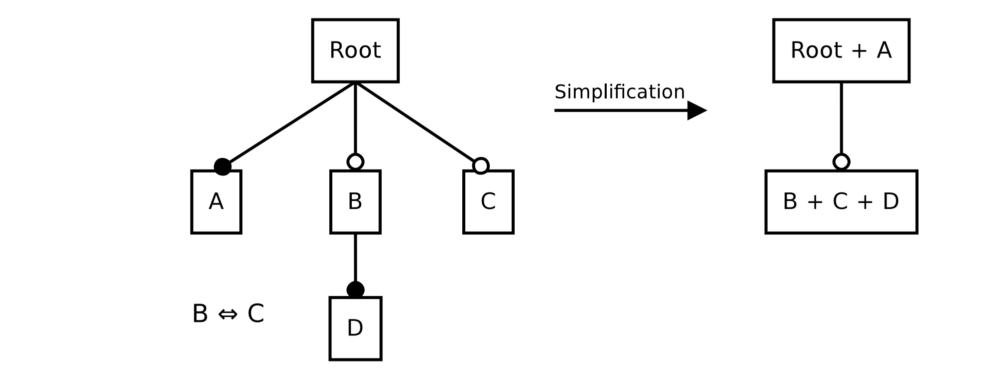


Figure 5.7: *Simplification* of a feature model with *atomic sets*

In the past, atomic sets were generated by traversal through the feature diagram[Seg08]. The algorithm grouped features which have a mandatory relation to each other. All those groups were merged which caused a simplification that decreased the number of variables and clauses[ZZM04]. Cross-tree constraints and other information attached to the feature model were skipped. To respect all the given information encoded in the feature model, we extended the traversal-based algorithm. We built a *divide and conquer* algorithm that is based on assumptions like the previously presented algorithms. This extension allows us to find **all** atomic sets in the feature model.

The simplification algorithm [6] takes a feature model M and a set of features F. At the initial call, F contains all features. In subsequent (recursive) calls, the set F contains just subsets of features which are potential atomic sets. The basic idea of the proposed algorithm is, to split F into two disjoint sets of features until we find an atomic set. As soon as we find an atomic set, we can merge features or rather simplify the feature model. An atomic set is found if $|F| = 1$ (trivial case) or each selection of a feature in F forces all remaining features to be enabled (by proof).

Proof: Let M be the PBSAT representation of the feature model. Let F be a set of features with $|F| > 1$. Let $f \in F$ and $G = F \setminus \{f\}$. The set F is atomic if and only if the selection of each feature $f \in F$ forces all remaining features $g \in G$ to be enabled. Assume f is enabled ($f = 1$) and g is disabled ($g = 0$). If the PBSAT solver determines that this expression is "unsatisfiable", we know that the following condition holds for feature model M.

$$\neg satisfiable(M \wedge f = 1 \wedge g = 0) \quad \Rightarrow \quad M \wedge f = 1 \rightarrow g = 1 \tag{5.3}$$

If we do this assummption-based check for each $f \in F$ and each $g \in G$ and show unsatisfiability of each combination, we know that F is an atomic set.

$$\begin{aligned} & \forall f \in F : \forall g \in G : \quad \neg satisfiable(M \wedge f = 1 \wedge g = 0) \\ \Rightarrow \quad & \forall f \in F : \forall g \in G : \quad M \wedge f = 1 \rightarrow g = 1 \\ \Rightarrow \quad & F \text{ is atomic set} \end{aligned} \tag{5.4}$$

For most calls to this procedure, F will not be an atomic set. In this case, we split the set of features as soon as possible. If the inner loop detects that the selection

propagates not to all $g \in G$, we split F into those features that were affected by the propagation including f and those which were not affected.

It is not required to run the traversal-based algorithm before applying our proposed one. The result will be the same in both cases. But we assume that the pre-processing by the traversal-based algorithm will accelerate the total runtime.

Algorithm 6 Compute the atomic sets

```
procedure COMPUTEATOMICSETS(M, F)
    if |F| = 1 then
        simplify(M, F)
        return
    end if
    for all f ∈ F do
        G ← F \ {f}
        H ← ∅
        for all g ∈ G do
            M ← addAssumptions(M, f = 1 ∧ g = 0)        ▷ enables f; disables g
            if ¬satisfiable(M) then
                H ← H ∪ {g}
            end if
            M ← removeAssumptions(M)                    ▷ remove assumptions
        end for
        if G ≠ H then
            computeAtomicSets(M, H ∪ {f})
            computeAtomicSets(M, F \ (H ∪ {f}))
            return
        end if
    end for
    simplify(M, F)
    return
end procedure
```

Performance adaptations

We already mentioned that the proposed algorithm is based on the divide and conquer design paradigm which provides parallel execution. Each (sub-)call of the $computeAtomicSets()$ procedure can be processed independently. To accomplish concurrency, the $simplify()$ function has to prevent race conditions. It can either

apply the simplifications after the computation of the atomic sets or it needs synchronization to simplify the feature model during parallel execution. Both variants will differ in implementation effort as well as performance. Given this extension, we are able to accelerate the runtime especially on large-scale feature models.

5.1.4 Edits

Definition: A *feature model edit* describes any modification on a feature model. Modifications occur during the development process of feature models. Adding, deleting or changing features and/or constraints are edits. Each edit affects the underlying representation of the feature model that causes semantic changes. Analysing these semantic changes allows to specify feature model edits.

Classifications of edits help to observe changes on the feature model. This allows to assist the domain engineer during feature modeling. Changes in the represented products will be reported and help to preserve semantics. Especially refactorings benefit from the categorization of feature model edits.

Figure 5.8 illustrates a modification on the feature model that is caused by adding a new feature C. The analysis will infer that new products were generated but none were eliminated. This observation is required to state the modification as *generalization*.

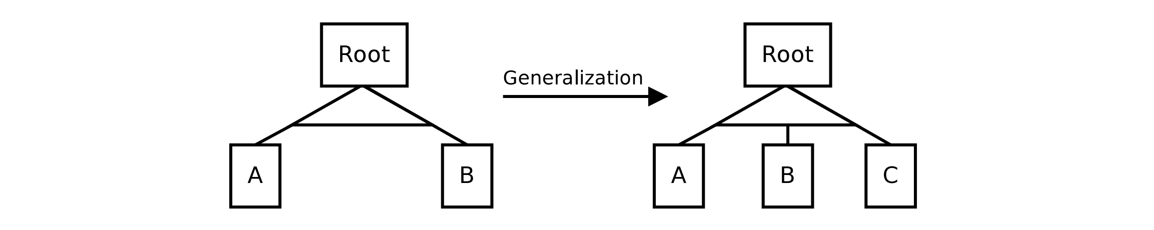


Figure 5.8: A *feature model edit* that is classified as *generalization*

Feature model edits are specified by analysis of the represented products. The analysis basically compares the feature model before and after the edit. It determines whether new variants have been generated and if variants have been eliminated. The edit types are either *refactoring*, *generalization*, *specialization* or *arbitrary edit*. We introduce $L(f)$ as the set of all products represented by feature model f. Figure 5.9

shows the four different edit types and their respective requirement which is expressed as subset relation of $L(f)$. Notice that g represents the feature model after the edit.

	$L(g) \subseteq L(f)$ (no products added)	$L(g) \not\subseteq L(f)$ (products added)
$L(f) \subseteq L(g)$ (no products deleted)	$L(f) = L(g)$ (Refactoring)	$L(f) \subset L(g)$ (Generalization)
$L(f) \not\subseteq L(g)$ (products deleted)	$L(g) \subset L(f)$ (Specialization)	(Arbitrary Edit)

Figure 5.9: Types of edits based on set inclusions or rather logical implications

Thüm et al. [TBK09] discovered a way to reason on feature model edits using a traditional SAT solver. This approach is based on the relation between $L(f)$ and the propositional logic representation. We already mentioned in chapter 2 that the number of products and the number of satisfying assignments of the propositional logic representation is equal. Therefore, let $P(f)$ be the propositional logic representation of f, which is primarily in CNF. Let v be a combination of features. If and only if $v \in L(f)$, then v is a valid product and consequently satisfies $P(f)$ by assigning all contained features of v to `true`. The connection between L and P allows to verify subset relation by logical implications as shown in 5.5.

$$L(f) \subseteq L(g) \quad \equiv \quad P(f) \Rightarrow P(g) \tag{5.5}$$

Thüm noticed that the conversion of $P(f) \rightarrow P(g)$ to CNF took much more time than the call to the SAT solver. The reason was the exponential explosion of clauses caused by $\neg P(f)$. An improved technique called *simplified reasoning* was intro-

duced. It provides a sophisticated way using multiple SAT solver calls and avoiding the time-consuming CNF conversion. It is based on the fact that $P(f)$ and $P(g)$ are very similar CNFs. Equation 5.6 declares c as the identical clauses whereas p_f and p_g are the distinct clauses.

$$\begin{aligned} P(f) &= p_f \wedge c \\ P(g) &= p_g \wedge c \end{aligned} \tag{5.6}$$

The *simplified reasoning* [TBK09] is based on the equivalence illustrated in 5.7. The resulting propositional formula consists of a disjunction of I_i for $i \in \{1, \cdots, n\}$. The I_i represents the expression $P(f) \wedge \neg R_i$ whereas each R_i for $i \in \{1, \cdots, n\}$ is a clause that arises by splitting the CNF p_g. The simplified reasoning checks satisfiability of each I_i independently. Therefore, we have up to n calls to the SAT solver. If one of the calls determines satisfiability, we know that $P(f) \nRightarrow P(g)$ holds. Otherwise, $P(f) \Rightarrow P(g)$ is valid.

$$\begin{aligned} & P(f) \nRightarrow P(g) \\ \equiv\ & P(f) \wedge \neg p_g \\ \equiv\ & \underbrace{(P(f) \wedge \neg R_1)}_{=I_1} \vee \cdots \vee \underbrace{(P(f) \wedge \neg R_n)}_{=I_n} \end{aligned} \tag{5.7}$$

Determining edit types using the PBSAT representation is similar to the presented approach. We just have to deal with linear equations and inequalities instead of CNF clauses. Therefore, we have to customize the presented approach slightly. The complete equivalence proof from 5.7 can be applied by thinking of equations and inequalities instead of clauses. The only adaption is the negation of the R_i's, which needs a little bit more work when treating equations and inequalities instead of clauses. Negating inequalities is trivial because of the discrete space. The equations in 5.8 visualize the procedure. Notice that we adjust the degree by $+1$ or rather -1 to preserve the pseudo-boolean representation.

$$\begin{aligned} \neg(\sum_{i=1}^{n} a_i x_i \geq d) \quad &\Leftrightarrow \quad \sum_{i=1}^{n} a_i x_i \leq d - 1 \\ \neg(\sum_{i=1}^{n} a_i x_i \leq d) \quad &\Leftrightarrow \quad \sum_{i=1}^{n} a_i x_i \geq d + 1 \end{aligned} \tag{5.8}$$

The negation of equations does not necessarily need more effort. The easiest way is the separation in two inequalities I'_i and I''_i which also requires two PBSAT solver calls instead of one. A smarter approach is the transformation from a disjunction of inequalities to a conjunction of inequalities. This allows the analysis in just one call of the PBSAT solver.

$$\begin{aligned}
& \neg(\sum_{i=1}^{n} a_i x_i = d) \\
\Leftrightarrow \quad & \sum_{i=1}^{n} a_i x_i \neq d \\
\Leftrightarrow \quad & \sum_{i=1}^{n} a_i x_i > d \quad \vee \quad \sum_{i=1}^{n} a_i x_i < d \\
\Leftrightarrow \quad & \sum_{i=1}^{n} a_i x_i \geq d + 1 \quad \vee \quad \sum_{i=1}^{n} a_i x_i \leq d - 1 \\
\Leftrightarrow \quad & \sum_{i=1}^{n} a_i x_i \geq d + 1 \quad \vee \quad -\sum_{i=1}^{n} a_i x_i \geq -d + 1 \\
\Leftrightarrow \quad & \sum_{i=1}^{n} a_i x_i \geq \underbrace{d + 1}_{=d'} \quad \vee \quad \sum_{i=1}^{n} a_i \overline{x_i} \geq \underbrace{-d + 1 + \sum_{i=1}^{n} a_i}_{=d''}
\end{aligned} \tag{5.9}$$

The arithmetic conversion in 5.9 is required to transform the logical operator. The conversion is based on the discrete space and equation 2.4. Equivalence 5.10 shows the transformation of the disjunction into a conjunction. The applied technique is similar to the reduction from the NP-complete problems "SAT" to "3SAT" [GJ79]. We have to introduce an auxiliary variable y to ensures satisfiability of both inequalities. If one of the inequalities is satisfied, the other becomes automatically satisfied by the right choice of y.

$$\begin{aligned}
& satisfiable\left(\sum_{i=1}^{n} a_i x_i \geq d' \quad \vee \quad \sum_{i=1}^{n} a_i \overline{x_i} \geq d''\right) \\
\Leftrightarrow \quad & satisfiable\left(d'y + \sum_{i=1}^{n} a_i x_i \geq d' \quad \wedge \quad d''\overline{y} + \sum_{i=1}^{n} a_i \overline{x_i} \geq d''\right)
\end{aligned} \tag{5.10}$$

The adaption to PBSAT has several advantages according to the reasoning performance. We already mentioned the compactness of the PBSAT representation. PBSAT instances are usually shorter than equivalent CNF instances. Therefore, fewer R_i's occur during simplified reasoning which decreases the number of calls to the

solver. We additionally benefit from the existence of equations in the R_i's because we do not need to split them in two separate inequalities. The presented negation uses an auxiliary variable to avoid unnecessary computation.

Thüm proposed two required extensions to handle addition and deletion of features as well as *abstract features*[TKES11]. Without those extensions, the results of the edit type reasoning would be wrong in some cases. The extension for added and removed features determines if new features have been added or existing features have been deleted. In this is the case, those features have to be disabled in the model they do not occur. This can be done by adding an assumption. We already handled assumptions in the previous algorithms. Disabling an arbitrary feature A can obviously be achieved by adding the inequality $a \leq 0$ to the PBSAT instance. The second extension for abstract features eliminates variables which represent abstract features. This is necessary because abstract features are *code-less* and therefore did not extend variability of the product line. We eliminate these variables by replacements known from linear equation systems. For each variable representing an abstract feature, we look for an equation to replace all occurrences of this variable in the remaining pseudo-boolean restrictions.

5.2 Application engineering

Application engineering describes the process of selecting and deselecting features to derive products. The configuration of products needs a feature model that defines the product line. This enables tool support to handle propagations of feature selections and provides explanations. Additionally, automated tool support prevents from building invalid products. The presented algorithm minimizes configuration effort and speeds up application engineering. Notice that we will use the term *feature selection* or rather *selection* to describe selections as well as feature **de**selections. The former describes addition of features where the latter means the exclusion of features.

5.2.1 Propagation of selections

Definition: Propagations are implicit feature selections that arise by manually selecting features during product derivation. The use of propagations can help to save

time because most of the selections can be done automatically according to the feature diagram and the cross-tree constraints. Furthermore, automatic selections prohibit to select contradictory or rather invalid products.

Propagations can be determined in different ways. One possible way is to encode selections in the feature model. This can be achieved by adding assumptions to forcedly enable or disable features. Afterwards, we run the core and dead feature algorithms to determine selections and deselections, respectively. The disadvantage of this approach are numerous calls to the SAT solver.

Our goal is to avoid as much (PB)SAT calls as possible. This can be done using *boolean constraint propagation* (BCP). Batory[Bat05] proposed a *logic truth maintenance system* (LTMS) that uses BCP to determine selections in feature models. BCP uses the characteristics of the conjunctive normal form in combination with assignments. Table 5.2 shows the different types of clauses that occur during BCP. The algorithm applies selections through assignments in the CNF and looks for *unit-open* clauses. A clause is unit-open if and only if it contains just a single literal. If we want to satisfy the whole CNF, we are forced to satisfy each clause and consequently all literals in unit-open clauses. If such a clause has been found, the boolean variable will be assigned to satisfy the clauses. The algorithm terminates if all variables are assigned or no further unit-open clauses exist.

TYPE	DESCRIPTION	EXAMPLE
satisfied	at least one literal is already satisfied	$(l_1 \wedge true \wedge l_3)$
violated	all literals are unsatisfied	$(false \wedge false \wedge false)$
unit-open	just one unassigned variable left to satisfy clause	$(false \wedge l_2 \wedge false)$
not unit-open	more than one variable is not yet assigned	$(l_1 \wedge false \wedge l_3)$

Table 5.2: The four different types of BCP clauses

BCP is not limited to propositional logic in conjunctive normal form. Pseudo-boolean systems of inequalities also provide BCP [CK03]. The algorithm has to be customized to deal with linear inequalities instead of clauses. Two important preliminaries are necessary to apply the adapted BCP. At first, we have to transform all equations and inequalities into greater-or-equal inequalities. This can be achieved by simple conversions shown in 5.11. Secondly, we have to convert the

left-hand side of the inequality to get only positive coefficients. The procedure to transform all coefficients positive is explained in chapter 2.

$$\begin{aligned}
\sum_{i=1}^{n} a_i x_i \leq d \quad &\Leftrightarrow \quad -\sum_{i=1}^{n} a_i x_i \geq -d \\
\sum_{i=1}^{n} a_i x_i = d \quad &\Leftrightarrow \quad \sum_{i=1}^{n} a_i x_i \leq d \quad \wedge \quad \sum_{i=1}^{n} a_i x_i \geq d
\end{aligned} \tag{5.11}$$

The basic functioning of the pseudo-boolean BCP is identical to the propositional logic version. Just the definitions of satisfied, violated, unit-open and not unit-open are different. Table 5.4 shows the definitions. An inequality is unit-open if and only if the variable with the biggest coefficient has to be satisfied to avoid violation. A violation occurs if the sum of all coefficients $\sum_{i=1}^{n} a_i$ is smaller than the degree d. In other words, even if all variables are satisfied, the inequality can not be valid. Each unit-open inequality produces new assignments which allow further propagations. The adapted BCP also stops if all variables were assigned or no unit-open inequalities exist anymore.

TYPE	DEFINITION
satisfied	$d \geq 0$
violated	$\sum_{i=1}^{n} < d$
unit-open	$\sum_{i=1}^{n} a_i \geq d > \sum_{i=1}^{n} a_i - a_k$
not unit-open	inequality is neither satisfied, violated or unit-open

Table 5.4: The four different types of BCP inequalities

Figure 5.10 shows a sample feature model where feature B was selected. This starts propagation which leads to the selection of feature D because of the constraint $B \Rightarrow D$. Moreover, feature E will be deselected because of the alternative group relation. Figure 5.11 illustrates the propagations of the pseudo-boolean BCP in a directed graph. The nodes represent assignments. The directed edges are annotated with the unit-open inequality that leads to new assignments. Notice that feature *Root* and C are core features and hence already assigned. Otherwise, the presented inequality $-c + d + e = 0$ would not be unit-open.

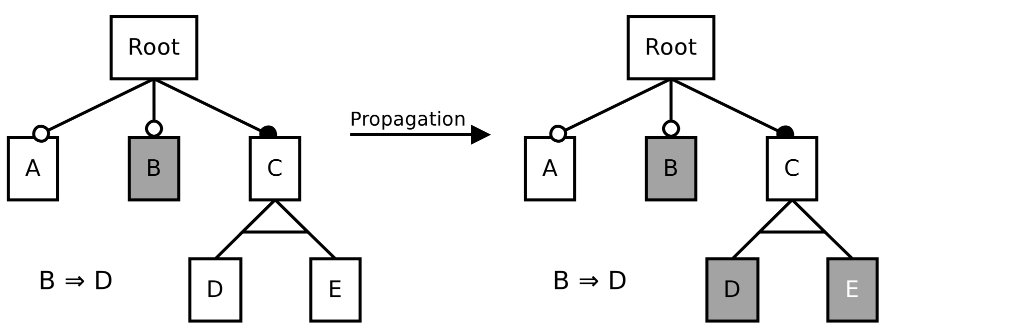


Figure 5.10: A *user selection* leads to a *propagation*

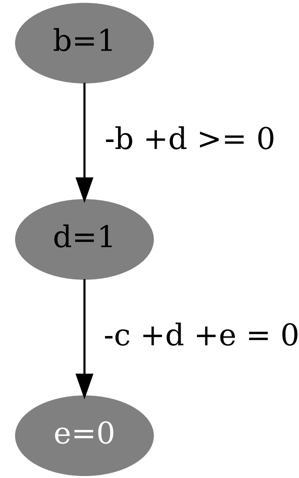


Figure 5.11: The propagation of 5.10 represented as graph

Both BCP algorithms are not complete which means that further propagations may exist. Therefore, all remaining features, which assignments have not been inferred, have to be checked by the (PB)SAT solver. We can use the core and dead feature algorithms as previously described.

6 Evaluation

The evaluation shall underline the applicability of the proposed analysis operations for the development of software product lines. We are interested in fast algorithms that allow quick responses. Minimizing the required runtime enables better user interaction in graphical tools like FeatureIDE and pure::variants. Therefore, we mainly focussed on the required computation time for selected algorithms. But we also examined the simplification potential of the contemplated feature models. At last, we tried to verify our assumption regarding the pre-processing to speed-up simplification. The evaluations revealed some unexpected results which may have an influence on further implementations.

We run the evaluation on an Intel Core 2 Duo CPU with a clock rate of 2 gigahertz. The installed operating system was a Linux system with kernel 2.6.35-30 (64 bit). The algorithms were executed on an OpenJDK[1] virtual machine for Java 6. We used the PBSAT solver of the SAT4J project[2] as reasoning engine. We measured the required time of the different analysis operations to check scalability. All contemplated algorithms were executed 100 times before we determined the arithmetic mean. We used the JAVA-method $System.nanotime()$ to determine the current time in nanoseconds. The stated times are all converted to milliseconds to ease comparison.

We used 11 real-world feature models of different software product lines from the FeatureIDE repository. Most of the feature models contain between 20 and 30 features. But we still have three larger models with more than 70 features whereas each of them has at least 20 additional constraints.

[1] `http://openjdk.java.net/`
[2] `http://sat4j.org/`

6.1 Runtime behaviour

Table 6.1 shows the analysed feature models with feature and constraint count as well as required time for different analysis operations. The column VALIDATION specifies the required time to determine whether the feature model is valid or void. It seems that the required processing time of the validation scales very well for feature models with less than a few hundred features. The PROPERTIES-column defines the required time to obtain the four different feature properties. We already mentioned in chapter 5 that the computation of dead, core, variant and false-optional features can be done in a compound algorithm. Therefore, we need $\mathcal{O}(2n)$ calls to the PBSAT solver where n denotes the number of features. We see that the required time increases very fast depending on the number of features. The last column PROPAGATION shows the required time to propagate a selection in the feature model. We evaluated the time by iteration over all features. We selected and deselected each features 100 times and determined the mean time until BCP stopped. We observe increasing time for larger models. But the algorithm scales very well even for the largest feature model.

MODEL	#FEAT.	#CONSTR.	VALIDATION (in ms)	PROPERTIES (in ms)	PROPAGATION (in ms)
FameDB2	21	1	0.543	6.089	0.825
FameDB	22	0	0.305	6.910	0.620
APL	23	2	0.194	3.530	0.838
SafeBali	24	0	0.204	3.818	0.688
Chat	25	1	0.230	4.561	0.663
APL-Model	28	8	0.224	4.635	0.767
TightVNC	28	3	0.207	4.623	0.636
GPL	38	15	0.605	8.359	2.217
BerkeleyDB	76	20	0.482	17.556	2.263
Violet	101	27	0.503	24.557	4.244
E-Shop	326	21	0.972	140.199	7.304

Table 6.1: Mean runtime of validation, property computation and propagation

6.2 Simplification potential

We extended the algorithm of atomic sets to guarantee maximum possible simplification of the feature model. Atomic sets allow to consider groups of features as

a single feature. Hence, atomic sets reduce the number of required variables and allow simplification of the underlying representation. The simplification eliminates redundant or rather satisfied entities of the representation. In our context, these entities are pseudo-boolean restrictions. Table 6.2 shows the number of features and restrictions for each feature model. Additionally, the number of atomic sets and restrictions after the computation are listed. We observed that all feature models had simplification potential. The potential varies between 2% in the Violet model and 58% in the SafeBali model. Even the largest feature model had a high simplification rate of 35%.

	Before computation		After computation			
Model	#Feat.	#Restr.	#AS's		#Restr.	
FameDB2	21	18	15	-29%	12	-33%
FameDB	22	16	15	-32%	9	-44%
APL	23	24	15	-35%	16	-33%
SafeBali	24	21	10	-58%	7	-67%
Chat	25	22	20	-20%	17	-23%
APL-Model	28	36	24	-14%	32	-11%
TightVNC	28	16	23	-18%	11	-31%
GPL	38	46	26	-32%	34	-26%
BerkeleyDB	76	109	54	-29%	80	-27%
Violet	101	143	99	-2%	141	-1%
E-Shop	326	294	212	-35%	179	-39%

Table 6.2: Results of the simplification (atomic set) computation

6.3 Pre-processing impact

In chapter 5, we assumed that the traditional tree-traversal algorithm for atomic sets is an efficient pre-processing for our extended algorithm. It turned out that this assumption seems to be wrong. The evaluation revealed that the pre-processing does not accelerate the required runtime. Furthermore, we noticed that the number of calls to the PBSAT solver does not correlate with the runtime. In some models, the computation of atomic sets was faster with pre-processing. In other models, the runtime without pre-processing had a better performance. Table 6.3 contains the results of our evaluation. We measured the time as well as the number of necesarry

call to the satisfiability solver. In most of the cases, the performance without pre-processing was better although more PBSAT calls were needed. The best example is the SafeBali model that needed more than 3 times PBSAT calls with disabled pre-processing. But the required time for the whole computation was still faster than the version with enabled pre-processing.

Model	+pre-processing		-pre-processing			
	runtime	SAT calls	runtime		SAT calls	
FameDB2	32.250	105	22.250	-31%	138	+24%
FameDB	22.218	105	20.514	-8%	151	+30%
APL	21.050	102	19.708	-6%	139	+36%
SafeBali	12.950	45	11.966	-7%	164	+264%
Chat	33.667	180	37.925	+13%	190	+6%
APL-Model	50.659	254	31.607	-38%	203	-20%
TightVNC	48.384	252	33.137	-32%	208	-21%
GPL	45.906	237	56.756	+24%	385	+62%
BerkeleyDB	200.078	1169	201.313	+1%	1648	+41%
Violet	773.947	3186	639.629	-17%	2592	-23%
E-Shop	3817.053	18672	3605.623	-6%	20238	+8%

Table 6.3: Mean runtime of the simplification (atomic set) algorithm

7 Conclusion

In this thesis, we proved that linear pseudo-boolean instances are able to represent feature models. We developed a mapping that allows independent translation of feature model components to PBSAT instances. Furthermore, we discovered that PBSAT instances allow handling and reasoning of integer attributes in features models. We proposed a special attribute file to assign values to feature attributes. Moreover, we also introduced another file to define additional constraints which can refer attributes.

Satisfiability solvers that work on the PBSAT representation allow similar reasoning as traditional SAT solvers. Therefore, we took different known feature model analysis operations and adapted them to work with a PBSAT solver. Simple algorithms needed almost no adaption. More sophisticated algorithms required deeper knowledge of the PBSAT representation and possible conversions. Some adapted algorithms enabled reasoning with fewer calls to the PBSAT solver. But fewer calls not necessarily indicated faster execution. We could not find analysis operations that are not feasible with the PBSAT representation.

In our evaluation, we saw that some algorithms scale very well. Especially the validation and the propagation algorithm. Other algorithms, which needed many PBSAT solver calls, slowed down if the number of features and additional constraints increased too much. The evaluation revealed that all feature models have simplification potential. This is very useful for refactoring and improving performance of other analysis operations. The evaluation also showed that the number of SAT calls does not correlate with the required time to look for satisfying assignments. We also verified that the pre-processing by the tree-traversal algorithm did not accelerate our proposed algorithm.

7.1 Contributions

The main contribution of the thesis is the usability of PBSAT problems to reason on feature models. In chapter 4, we examined how (extended) feature models can be mapped to PBSAT instances in the most compact representation.

During the adaption of the algorithms, we observed potential for further improvement and therefore extended existing analysis operations. We extended the simplification algorithm to find **all** existing atomic sets which ensures maximum simplification. The proposed algorithm supports the divide and conquer design paradigm that enables parallel execution. We also found out that reasoning of feature model edits using the PBSAT representation needs fewer calls to the solver. Further calls can be saved if our proposed conversion for equations will be used. In addition, we proposed the distinction between *tree-based atomic sets* and *real atomic sets* since the widely known tree-traversal algorithm does not detect all atomic sets.

We also proposed a new definition of the term *variant feature* to have a clear distinction between feature properties. In our point of view, features are either *core*, *dead* and *variant*.

We found out the PBSAT instances are well suited to represent feature models. The compactness and the expressiveness are the major benefits of this representation. Additionally, it allows us to define cardinality-based feature models easily and supports feature attributes.

7.2 Further work

During this thesis, we found several interesting issues we want to address in near future. This section gives a brief overview about the related topics we want to deal with.

Inspired by the logic truth maintenance system (LTMS) proposed by Batory[Bat05], we want to check feasibility of explanations using a PBSAT solver. Explanations describe automatic propagation in a human-readable way. This helps users to comprehend automatic selection during product derivation.

PBSAT solvers usually provide an objective function that allow to find optimal solutions. An optimal solution is a satisfying assignment which maximizes (or minimizes) the value of the objective function. We want to use this target function to derive optimal products based on different objectives.

The evaluation in this thesis used 11 different real-world feature models from software projects. In future, we want to enlarge our repository of feature models to get more representative evaluation results. Therefore, we will gather variability models from different business sectors.

We are also interested in a comprehensive performance comparison of SAT and PBSAT. Therefore, we have to implement missing analysis operation based on a SAT reasoning engine.

Bibliography

[Bat05] Don S. Batory. Feature models, grammars, and propositional formulas. In *Proc. Int. Software Product Line Conference (SPLC)*, pages 7–20. Springer, 2005.

[BCTS06] David Benavides, Antonio Ruiz Cortés, Pablo Trinidad, and Sergio Segura. A survey on the automated analyses of feature models. In *Proc. Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, pages 129–134, 2006.

[BHP08] Timo Berthold, Stefan Heinz, and Marc E. Pfetsch. Solving pseudo-boolean problems with scip. Technical Report 08-12, ZIB, Takustr.7, 14195 Berlin, 2008.

[BPSP04] Danilo Beuche, Holger Papajewski, and Wolfgang Schröder-Preikschat. Variability management with feature models. *Jour. Science of Computer Programming (SCP)*, 53(3):333–352, 2004.

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[BSTC07] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz Cortés. Fama: Tooling a framework for the automated analysis of feature models. In *Proc. Int. Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 129–134. The Irish Software Engineering Research Centre, 2007.

[BTRC05a] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated Reasoning on Feature Models. In *Proc. Int. Conf. Advanced Information Systems Engineering (CAiSE)*, pages 381–390. Springer, 2005.

[BTRC05b] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Using Constraint Programming to Reason on Feature Models. In *Proc. Int. Conf. Software Engineering & Knowledge Engineering (SEKE)*, pages 677–682. Knowledge Systems Institute, 2005.

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[CK03] Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. In *Proc. Design Automation Conference (DAC)*, pages 830–835. ACM Press, 2003.

[CW07] Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. In *Proc. Int. Software Product Line Conference (SPLC)*, pages 23–34. IEEE Computer Society, 2007.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[KK06] Martin Kreuzer and Stefan Kühling. *Logik für Informatiker (in German)*. Pearson, 2006.

[KTS+09] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. Featureide: A tool framework for feature-oriented software development. In *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE Computer Society, 2009.

[pur09] pure-systems GmbH. *pure::variants User's Guide: Version 3.0 for pure::variants 3.0*, 2009. Source: `http://www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf`.

[Seg08] Sergio Segura. Automated Analysis of Feature Models Using Atomic Sets. In *Proc. Int. Software Product Line Conference (SPLC)*, pages 201–207. Lero Int. Science Centre, 2008.

[TBK09] Thomas Thüm, Don S. Batory, and Christian Kästner. Reasoning about edits to feature models. In *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE Computer Society, 2009.

[TKES11] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. Abstract Features in Feature Modeling. In *Proc. Int. Software Product Line Conference (SPLC)*, pages 191 –200, 2011.

[WLS+07] Hai H. Wang, Yuan Fang Li, Jing Sun, Hongyu Zhang, and Jeff Pan. Verifying feature models using owl. *Jour. Web Semantics*, 5(2):117–129, 2007.

[ZZM04] Wei Zhang, Haiyan Zhao, and Hong Mei. A propositional logic-based method for verification of feature models. In *Formal Methods and Software Engineering*, pages 115–130. Springer, 2004.

Statutory declaration

Hereby I declare that I have written this bachelor thesis by my own. Furthermore, I confirm that no other sources have been used than those specified in the bachelor thesis itself. This thesis, in same or similar form, has not been available to any audit authority yet.

Passau, October 28, 2011

____

Sebastian Henneberg