



Bachelor's Thesis

Computing Attribute Ranges for Partial Configurations with JavaSMT

Authors:

Joshua Sprey, Chico Sundermann

April 15, 2018

Advisors:

Prof. Dr.-Ing Ina Schaefer
Dr.-Ing Thomas Thüm

Institute of Software Engineering and Automotive Informatics

Sprey, Joshua; Sundermann, Chico:
Computing Attribute Ranges for Partial Configurations with JavaSMT
Bachelor's Thesis, TU Braunschweig, 2018.

Abstract

Software product lines are able to describe multiple products sharing a common base of features and are commonly described as feature models. For complex software product lines, automatic analyses are required to ensure validity and to improve the interactive configuration process. Modern SAT solvers are vital components for the validation process of feature models. The increasing variability of software product lines implies the need to use more expressive solvers like SMT solvers. To assist the development of feature modeling tools, we compare SAT and SMT solvers for the automated analysis of feature models. During this thesis, we create an abstract data type to formally define analyses for feature model defects and their explanations. The result show that SAT solvers are more efficient at detecting the defects, while SMT solvers can find explanations for them multiple times faster. Feature models can be further expanded by attaching attributes to features. Such attributes may contain a numerical value. Additionally, one attribute can be defined for multiple features. In this thesis, we aim to support the interactive configuration process, by providing the range of the sum of values for an attribute. Such ranges depend on the remaining choices in a configuration of the product line. We provide an exact computation using SMT and an approximation using an heuristic. The evaluation results show that an SMT solver is not suitable for supporting interactive configuration. However, the approximated ranges of the provided heuristic were very close to the exact ones.

Acknowledgements

We would like to express our very great appreciation to our main advisor Dr.-Ing. Thomas Thüm for his continuous support during this thesis. He always took his time to provide constructive feedback for our drafts and ideas. We also would like to thank Sebastian Krieter for his support regarding the feature model analyses and their optimizations. Additionally, we would like to thank Michael Nieke.

Cooperation

Table 1 gives an overview about the distribution of this thesis. Every chapter is assigned to one of the authors or both. The chapters assigned to both cover the introduction, the background, the implementation of the feature attribute fundamentals, and the conclusion. For the related and future work chapters, we show the distribution for the paragraphs in Table 2.

Chapter	Both	Sprey	Sundermann
1	✓	-	-
2	✓	-	-
3	-	✓	-
4	-	-	✓
5	5.1, 5.4, 5.7	5.2, 5.3	5.5, 5.6
6	-	✓	-
7	-	-	✓
8	✓	-	-
9	✓	-	-
10	✓	-	-

Table 1: Overview about the distribution of the chapters

Chapter	Paragraph	Sprey	Sundermann
Related Work	Comparison of SAT and SMT Solvers for Software Product Lines	✓	-
Related Work	Feature Model Analysis	✓	-
Related Work	Optimization of Feature Attributes	-	✓
Future Work	Feature Attributes	✓	✓
Future Work	Feature Model Analysis	✓	-
Future Work	Comparison of SMT and SAT	✓	-
Future Work	Unsatisfiable Core	✓	-
Future Work	Computing Attribute Ranges	-	✓
Future Work	Support for the Interactive Configuration Process	-	✓

Table 2: Overview about the distribution of the paragraphs for the related and future work chapters

Contents

List of Figures	x
List of Tables	xi
1 Introduction	1
2 Background	5
2.1 Feature Modeling	5
2.1.1 Feature Models	5
2.1.2 Extended Feature Models	8
2.2 Satisfiability of Logical Expressions	9
2.2.1 Boolean Satisfiability	9
2.2.2 Satisfiability Modulo Theory	11
3 Unification of SAT and SMT Solvers for the Software Product Line Analysis	13
3.1 Motivating Example	13
3.2 Abstract Data Type for the Software Product Line Analysis	15
3.3 Feature Model Analyses	17
3.3.1 Void Feature Model Analysis	17
3.3.2 Core and Dead Feature Analysis	17
3.3.3 False-Optional Feature Analysis	19
3.3.4 Redundant Constraint Analysis	20
3.3.5 Tautological Constraint Analysis	22
3.4 Automated Feature Model Analysis	23
3.5 IncrementalSolver Extension for Explanations	23
3.6 Summary	24
4 Computing Attribute Ranges of Partial Configurations	25
4.1 Motivating Example	25
4.2 The Problem	26
4.3 Computing Attribute Ranges using SMT	27
4.4Computing Attribute Ranges using an Heuristic	31
4.5 Summary	38
5 Implementation into FeatureIDE	39
5.1 Building on Existing Tools	39
5.2 Data Structure for Formulas	40

5.3	Implementing the Abstract Data Type <code>IncrementalSolver</code>	43
5.4	Feature Attributes	49
5.5	Statistics for Configurations	58
5.6	Attribute Entry Extension	62
5.6.1	Compute Ranges using SMT	64
5.6.2	Estimation Algorithm	65
5.7	Summary	67
6	Evaluation of Sat4J and JavaSMT on SAT-based Tasks	69
6.1	Research Questions	69
6.2	Evaluation Setup	70
6.3	Efficiency of SAT and SMT on the Feature Model Analysis	71
6.4	Explanations for the Feature Model Analysis	78
6.5	Summary	80
7	Evaluation of the Attribute Range Computation	85
7.1	Research Questions	85
7.2	Creating the Models	86
7.3	Evaluation Setup	87
7.4	Evaluation of Efficiency	88
7.4.1	Runtime of Computing Attribute Ranges	88
7.4.2	Distribution of the Runtime Needed for Computing Attribute Ranges with an SMT Solver	91
7.5	Evaluation of Precision	94
7.6	Summary	96
8	Related Work	97
9	Conclusion	99
10	Future Work	101
A	Appendix	105
	Bibliography	109

List of Figures

2.1	Example feature diagram of a car	7
2.2	Example of a regular and an extended configuration for the car feature model	8
3.1	Feature model with visualized inconsistencies, adapted from [KAT16a]	13
3.2	Feature model with an explanation for the dead feature <i>Bluetooth</i> . .	15
3.3	The procedure of the void feature model analysis	17
3.4	The procedure of the unoptimized core and dead feature analysis . . .	19
3.5	The procedure of the core and dead feature analysis using filtering . .	19
3.6	The procedure of the core and dead feature analysis using filtering and different selection strategies	20
3.7	The procedure of the unoptimized false-optional feature analysis . . .	20
3.8	The procedure of the false-optional feature analysis using filtering . .	21
3.9	The procedure of the false-optional feature analysis using filtering and a different selection strategy	21
3.10	Procedure of the redundant constraint analysis with optimizations . .	22
3.11	Procedure of the tautological constraint analysis	23
3.12	Procedure of the automated feature model analysis	23
4.1	Motivating example of a feature model representing a sandwich . . .	25
5.1	The original state of PROP4J	42
5.2	Extension of PROP4J that we implemented to build restricted first-order expressions	45
5.3	Class diagram for the SAT and SMT problems	46
5.4	Class diagram for the solver interface	47
5.5	Class diagram for the analyses	48

5.6	Attribute implementation into FEATUREIDE	49
5.7	Architecture of the extended feature model implemented into FEATUREIDE	51
5.8	Architecture of the extended feature model format implemented into FEATUREIDE	52
5.9	Feature attribute view while filtering the feature <i>Toast</i>	53
5.10	<i>Sandwich</i> feature model in the feature diagram editor provided by FEATUREIDE on the left and in the feature attribute view on the right.	54
5.11	<i>Sandwich</i> feature model in the feature diagram editor provided by FEATUREIDE where the features <i>Cheese</i> and <i>Meat</i> are currently selected. On the right side is the attribute view with an activated filter for the features selected in the feature diagram editor.	55
5.12	Configuration for the <i>Sandwich</i> in the configuration editor provided by FEATUREIDE on the left side. On the right side is the attribute view that displays only configurable attributes of features that are currently selected in the configuration.	56
5.13	Class diagram for the architecture of the feature attribute view	56
5.14	Class diagram for the configuration outline	58
5.15	Class diagram for the outline entry interface	59
5.16	Example of the attribute entry statistics for the displayed configuration of a sandwich	60
5.17	Class diagram for the default outline entries	61
5.18	Architecture of the attribute outline entry	62
5.19	Example of attribute entry statistics for the displayed configuration of a sandwich	63
5.20	Example of the range computation with approximated on the left and exact values on the right side	63
5.21	Outline entry responsible for computing the maximal sum of the attributes values	64
5.22	Outline entry that is responsible for computing the minimal sum of the attributes values	65
5.23	Class diagram for the range computation using SMT	66
5.24	Class diagram for the range computation using the estimation algorithm	66
6.1	Results of the void feature model analysis. (a) and (c) show the total runtime of every solver. (b) and (d) show the percentages for the analysis steps of every solver	73

6.2	Results of the unoptimized core and dead feature analysis. (a) and (c) show the total runtime of every solver. (b) and (d) show the percentages for the analysis steps of every solver	74
6.3	Results of the optimized core and dead feature analysis. (a) and (c) show the total runtime of every solver. (b) and (d) show the percentages for the analysis steps of every solver	75
6.4	Results of the SAT4J optimized core and dead feature analysis. (a) and (c) show the total runtime of every solver. (b) and (d) show the percentages for the analysis steps of every solver	76
6.5	Results of the unoptimized false-optional feature analysis. (a) and (c) show the total runtime of every solver. (b) and (d) show the percentages for the analysis steps of every solver	77
6.6	Results of the optimized false-optional feature analysis. (a) and (c) show the total runtime of every solver. (b) and (d) show the percentages for the analysis steps of every solver	78
6.7	Results of the SAT4J optimized false-optional feature analysis. (a) and (c) show the total runtime of every solver. (b) and (d) show the percentages for the analysis steps of every solver	79
6.8	Results of the redundant constraint analysis. (a) and (c) show the total runtime of every solver. (b) and (d) show the percentages for the analysis steps of every solver	80
6.9	Results of the tautological constraint analysis. (a) and (c) show the total runtime of every solver. (b) and (d) show the percentages for the analysis steps of every solver	81
6.10	The results of finding explanations for dead features	82
6.11	The results of finding explanations for false-optional features	82
6.12	The results of finding explanations for redundant constraints	83
7.1	Runtime of the computation using Z3 and the heuristic for the bike feature model	89
7.2	Runtime of computing attribute ranges using Z3 and the heuristic for the car feature model	90
7.3	Runtime of computing attribute ranges using Z3 and the heuristic for the PC feature model	91
7.4	Runtime comparison of the computation of attribute ranges using Z3 for the three feature models	91
7.5	Required time for each step of the attribute range computation using Z3 for our bike feature model	92
7.6	Required time for each step of the attribute range computation using Z3 for our car feature model	93

7.7	Required time for each step of the attribute range computation using Z3 for our PC feature model	93
7.8	Percentage difference between exact and approximated values for our bike feature model	94
7.9	Percentage difference between exact and approximated values for our car feature model	95
7.10	Percentage difference between exact and approximated values for our PC feature model	95

List of Tables

1	Overview about the distribution of the chapters	iv
2	Overview about the distribution of the paragraphs for the related and future work chapters	iv
4.1	Attributes attached to features of the <i>Sandwich</i> model	26
5.1	The native solvers supported by JAVASMT and their functionality, adapted from [KFB16]	41
5.2	Comparison of the functionality of SAT4J and JAVASMT	41
5.3	PROP4J nodes for propositional formulas and their functionality . . .	43
5.4	PROP4J nodes for restricted first-order formulas and their functionality	44
5.5	Description for the different icons used in the feature attribute view .	54
A.1	List containing the name, number of features, and number of constraints for all 120 feature models that were evaluated for the comparison of SAT and SMT solvers	108

1. Introduction

The high demand for custom software, beginning in the early 90s, led the software industry to change the way software is developed. Companies were in need of efficient software with special requirements that were not provided by mass-produced software. Instead of focusing on generalized *one-size-fits-all* solutions and standard software, a new development approach was needed that offers diversity and fulfills the requirements of every customer while benefiting from mass production [ABKS13].

Software product lines realize the concept of *mass customization* to account the needs of a single customer while maintaining an efficient production and has therefore gained interest in the software industry [ABKS13]. A common way to display software product lines are feature models [BSRC10, BRN⁺13]. A feature model contains the information about different features of a software product line and the relationship between them [BSRC10]. Apel et al. define a feature as “a characteristic or end-user-visible behavior of a software system” [ABKS13]. As core elements for the software, the features are designed to be as independent and reusable as possible to create a configuration by combining them. Every feature model has a diagram which structures the features in a hierarchical order allowing us to display cross-tree-constraints which describe relations of features that are not directly connected [BSRC10]. Such constraints can significantly damage the consistency of the feature model, and therefore automated analyses are needed [ABKS13]. The process of the analysis is straightforward. The feature model is translated to propositional logic and the task of finding an assignment that fulfills the structural and the cross-tree constraints is typically delegated to a SAT solver [MWC09]. The analysis can detect multiple defects and therefore helps in validating the feature model [Bat05, EPAH09].

To further optimize the selection of configurations it is possible to add extra information to the features. The additional information is called a feature attribute [CHE05]. Feature models that contain features with attributes are called extended feature models [BSRC10]. As an example, the price for each feature can be assigned directly as an attribute.

Usually the customer does not have the overview of all the features that are provided. Therefore, the customer starts by selecting the most relevant features that are

important for him. Such selection is called partial configuration. As part of our work, we want to automatically calculate the ranges of feature attributes for a given partial configuration. The calculation of the ranges provides the customer with information. As an example, we can calculate the maximum and minimum price which can be reached by selecting the remaining features. With the calculation the customer directly sees the impact on the price range when selecting a new feature.

For the computation of feature attribute ranges the given structure for SAT solvers is too restrictive because the computation of non-Boolean attributes requires non-Boolean variables and arithmetic operations which are not supported by SAT solvers [RMH12]. In our work, we will focus on solving ranges for attributes with the help of an SMT solver, which solves formulas in first-order logic including the needs to construct queries for the attribute ranges [RMH12].

SMT solvers are superior to SAT solvers regarding functionality. Thus, it is also possible for SMT solvers to solve propositional formulas. The SMT solvers were improving significantly in the last two decades, making them a promising alternative to SAT solvers [BSS⁺09]. As part of our work, we want to know whether SMT solvers are also superior regarding efficiency by comparing SMT solvers with SAT solvers on analyzing feature models. The feature model analysis consists of multiple analyses, one for each kind of defect. When both kinds of solvers have different advantages for the various analyses, it is also interesting to determine the most suitable solver for each analysis. Further, if the combination of SAT solvers and SMT solvers, could improve the overall performance of the analysis.

Goal of this Thesis

The goal of this thesis is to further expand the possibilities of extended feature models and help developers to decide between SAT and SMT solvers while working with feature modeling tools. This will be achieved in the following steps:

First, we want to evaluate the efficiency and possibilities of SAT and SMT solvers, regarding their usage for the automated feature model analysis. In many cases developers have to choose the most suitable solver for their feature modeling tool. For example, checking the validity of a configuration is a relevant problem for feature modeling. To help with this decision, we aim to answer the following questions. Are SMT solvers superior to SAT solvers regarding their efficiency for the automated analysis of feature models? Does a combination of both kind of solvers improve the overall process?

Afterward, we want to allow the computation of attribute ranges for a partial configuration. These ranges result from the leftover choices. Let us take a look at an example to understand the benefit of it. The considered feature model describes the components of a car. A customer thinks about including wheels of a specific type but is not sure whether this will automatically exceed his budget. Computing the ranges shows whether there is a product containing his included wheel type within its budget. Unfortunately, every car with his desired wheel type is too expensive. Therefore, he is interested in which types allow him to freely decide on the other components. This requires multiple computations of ranges, as he needs one for each wheel type. This use-case demands a short runtime to ensure fluent work. In order

to fully satisfy the requirements following from the example, two different kinds of computations are needed. First, there has to be an efficient way to always get correct results. Additionally, as determining exact ranges might not scale, we need an accurate estimation for on-the-fly computations. The exact computation will be performed by an SMT solver. However, we aim to use an heuristic to get a close estimation. In both cases, the scalability of the found solutions will be a crucial property.

Structure of the Thesis

This thesis is structured as follows. In [Chapter 2](#), we give insight into the essential background of this thesis, mainly consisting of feature modeling and satisfiability of logical expressions. This is supposed to specify the state-of-the-art for both topics and enable the reader to comprehend the following chapters. In [Chapter 3](#), we discuss the conceptual development of creating an abstract data type for SAT and SMT solvers. Additionally, with the help of the abstract data type, we formally define an analysis for each feature model defect. In [Chapter 4](#), we describe the conceptual development of correctly computing and approximating attribute ranges of a partial configuration. In [Chapter 5](#), we describe the implementation of both concepts, which will be added to the feature modeling tool FEATUREIDE [MTS⁺17]. In [Chapter 6](#), we evaluate the implemented solvers by performing all analyses defined in the concept for both kinds of solvers. To ensure efficiency and correctness of the implemented attribute range computations, we then evaluate them, by using re-engineered publicly available industrial product configurators as feature models [Chapter 7](#). In the remaining chapters, we discuss related work, come to a conclusion, and propose possible future work.

2. Background

This chapter gives an introduction to the required background and terminology to comprehend the following chapters. In [Section 2.1](#), we explain the fundamentals of feature modeling. Afterward, we introduce the SAT and SMT problem and show the different tools to solve them in [Section 2.2](#).

2.1 Feature Modeling

A software product line describes a set of products sharing a base of reusable software parts, called features. Features are distinguishable characteristics of a product line [\[KOD13\]](#). The goal is to develop each of the features once and reuse them for different products. Each product is one specific selection of features of a product line [\[ABKS13\]](#). For example, a family of cars could be described by a product line, with features describing properties of the car (e.g., the type of the engine).

Software product lines provide a form of mass customization, as a wide variety of products typically results from a product line, without the need of developing each one from the scratch [\[ABKS13\]](#). Mass customization allows low development costs, while still fulfilling individual needs. In particular, customization is important in smaller market segments and resource-constrained environments [\[ABKS13\]](#).

2.1.1 Feature Models

A feature model represents features of a product line and their relations. It is used to visualize and analyze the variability of the product line. Such a model efficiently defines possible combinations of the features. For our definition of feature models, we adapted the definition provided by Knüppel et al. [\[KTM⁺17a\]](#).

Definition 2.1 (Feature Model). *A feature model M is given by a 6-tuple $M = (N, r, \omega, \lambda, \Pi, \Psi)$ where :*

- N is the set of features.

- r is the root feature of the model.
- $\omega : N \mapsto \{0, 1\}$ indicates that a feature $f \in N$ is either mandatory ($\omega(f) = 1$) or optional ($\omega(f) = 0$).
- $\lambda : N \mapsto \mathbb{N} \times \mathbb{N}$ is a function representing the relationship between a feature and its child features. $\lambda(f) = \langle n, m \rangle$ implies that at least n and at most m children have to be included.
- $\Pi \subset N \times N$ indicates the parents for each feature as if $(f, g) \in \Pi$, f is the parent of g .
- $\Psi \subseteq \text{Prop}_N$ is the set of additional propositional formulas. We define Prop_N as the set of propositional formulas that can be built by using set of literals N .

A feature model can also be described using a conjunctive normal form. During this thesis, we reference this as CNF_M .

Feature Diagram

A feature diagram is a graphical documentation of a feature model, which is structured as a tree model [ABKS13]. Each node of the tree references a feature of the model. The edges describe relations between them, with the help of the following types:

Optional: The inclusion of the parent does not force the inclusion of an optional child.

Mandatory: The inclusion of the parent demands the inclusion of all its mandatory children.

And: Any number of children can be included ($\lambda(\text{feature}) = \langle 0, n \rangle$). The children of an **And**-relation can be **Optional** or **Mandatory**.

Or: At least one of the children has to be included, if the parent is included ($\lambda(\text{feature}) = \langle 1, n \rangle$).

Alternative: Exactly one child has to be included, if the parent is included ($\lambda(\text{feature}) = \langle 1, 1 \rangle$).

The given example in Figure 2.1 shows a feature diagram that describes components of a car, using every introduced relation. However, not every relation between features can be expressed in a tree structure, as there might be dependencies between different subtrees. This can be solved by using cross-tree constraints [ABKS13].

A cross-tree constraint is a propositional formula that can be added to the dependencies given by the tree of a feature model. The overall formula is built by a conjunction of the constraints [ABKS13]. In our example, the additional constraint $\text{Diesel} \Rightarrow \neg \text{Petrol}$ is given.

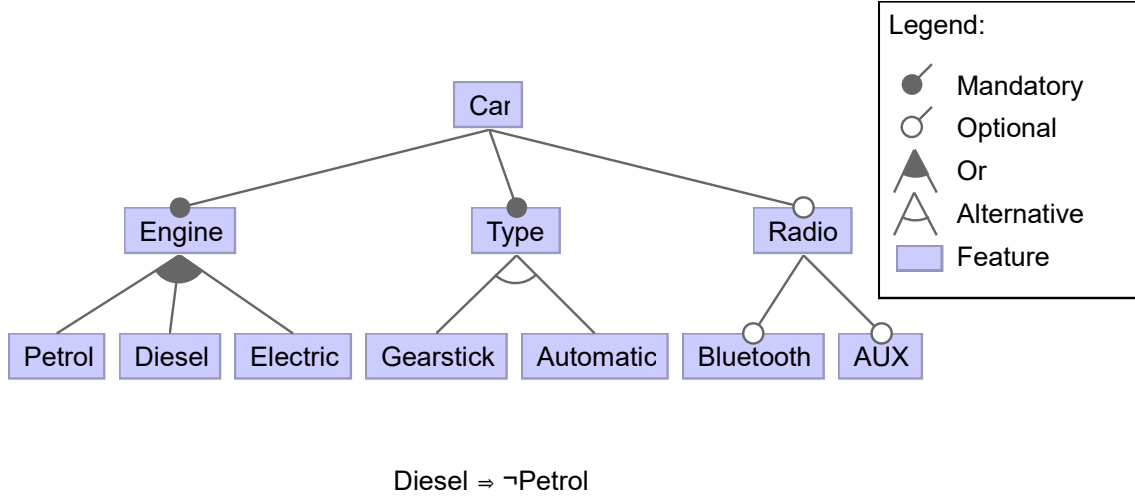


Figure 2.1: Example feature diagram of a car

Configuration

A configuration allows the developer to create specific products of a feature model by choosing the features that should be included. Additionally, a feature can be excluded. Therefore, each configuration of a feature model is defined by two sets. The first set contains all the selected the second one all unselected features.

Definition 2.2 (Configuration). *A configuration is a 3-tuple $C = (M, S, U)$ where:*

- M is the corresponding feature model.
- $S \subseteq N_M$ is the set of selected features.
- $U \subseteq N_M$ is the set of unselected features.
- $S \cap U = \emptyset$

Both pictures in Figure 2.2 show a valid configuration of our example feature diagram, representing a specific car with an electric engine, automatic handling and a radio with bluetooth. However, the right one shows additional information, like unselected features (denoted by the red minus) and the parent-children relationship of features in the feature diagram.

Definition 2.3 (Valid Configuration). *A configuration $C = (M, S, U)$ with $M = (N, r, \omega, \lambda, \Pi, \Psi)$ is called valid, when the propositional formula*

$$CNF_M \wedge \bigwedge_{s \in S} s \wedge \bigwedge_{u \in U} \neg u$$

is satisfiable. Equivalent, a configuration is valid when the following properties are given:

- Every feature $f \in N$ with $\omega(f) = 1$ and $(e \in S, f) \in \Pi$ is in S .

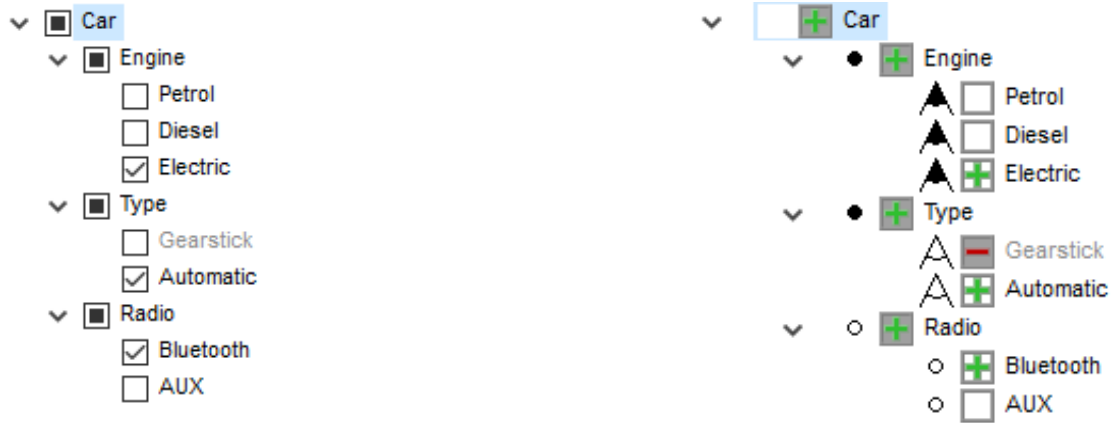


Figure 2.2: Example of a regular and an extended configuration for the car feature model

- For every feature $f \in N$ with $\lambda(f) = \langle n, m \rangle$, the number of included children is between n and m .
- Every formula $c \in \Psi$ is satisfied.

Sometimes it might be important to consider a subset of the product line. This can be achieved by using a *partial* configuration, which does not specify the selection status of every feature.

Definition 2.4 (Partial Configuration). A partial configuration $C = (M, S, U)$ is a configuration with $S \cup U \neq N_M$ [BSRC10].

Definition 2.5 (Full Configuration). A full configuration $C = (M, S, U)$ is a configuration with $S \cup U = N_M$ [BSRC10].

2.1.2 Extended Feature Models

Feature models can be expanded by attaching additional properties to features. These properties are called in feature attributes. Models containing features with attributes are referenced as extended (might also be called advanced or attributed) feature models [BSRC10].

Currently, there is no consensus on the structure of feature attributes. However, most agree the properties of an attribute should at least include a name, a domain and a value [BSRC10]. In the literature, the following domains of attributes are considered: \mathbb{R} , \mathbb{N} , boolean values, and enumerates [WDS09, CSHL13]. Additionally, the domains can either be finite or infinite [KOD13]. During this thesis, we consider the following domains for our attributes: \mathbb{R} , \mathbb{N} , boolean values and strings.

Let us take a look at an example of the usage of feature attributes. We consider the feature *Electric* of our example model in Figure 2.1. In addition to the dependencies in the tree, we want to capture the price and the power of our engine. To do so, we attach the following attributes:

- Name: Price
 - Value: 1200
 - Unit: €
- Name: Power
 - Value : 120
 - Unit : kW

Attaching such attributes to features allows access to additional information and further comparisons. Furthermore, the model analysis can be expanded to consider attributes (e.g. setting a maximum overall price).

Definition 2.6. *We assume Σ is the set of all characters. An extended feature model is a 3-tuple $E = (M, A, \pi)$ with:*

- M is a feature model.
- A is the set of attributes.
- $\pi_{att} : f \in N_M \mapsto \mathbb{R} \cup \{true, false\} \cup \Sigma^*$ is a function that returns the value of an attribute attached f . $\pi_{att}(f) = \emptyset$ indicates that att is not attached to f .

2.2 Satisfiability of Logical Expressions

In this section, we explain the satisfiability of logical expressions. In [Section 2.2.1](#), we introduce the satisfiability problem and the tools to solve them. Afterward, in [Section 2.2.2](#), we introduce the satisfiability modulo theories problem and the tools to solve them.

2.2.1 Boolean Satisfiability

Boolean satisfiability handles propositional expression, also called formulas, and asks whether a given formula is satisfiable. The formulas consist of boolean variables which can be set to either *true* or *false*. The variables are connected by different connectors. If it is possible to assign the values $\{true, false\}$ to the variables, such that the propositional formula evaluates to *true*, then the formula is satisfiable.

The SAT problem gets a formula in conjunctive normal form as input and decides whether the formula is satisfiable. The conjunctive normal form is a conjunction of multiple clauses. A clause is a disjunction of multiple literals, and a literal is a variable or a negated variable. Having that in mind the following definition originates:

Definition 2.7 (CNF). *We assume that $\alpha_{1,1} \dots \alpha_{n,m}$ for $n, m \in \mathbb{N}$ are variables. Let $A_1 \dots A_n$ be clauses with the literals $A_i = \bigvee_{k=1}^m \alpha_{ik}$. Then the CNF is defined as followed:*

$$CNF = \bigwedge_{i=1}^n A_i = \bigwedge_{i=1}^n \left(\bigvee_{k=1}^m \alpha_{ik} \right)$$

The challenge is now to find an assignment such that every clause will be evaluated to *true*, which causes the CNF to be *true*. If there exists no satisfying assignment, then the CNF is unsatisfiable.

Example 2.1. Let α, β, γ be variables. The following formulas are in CNF.

$$(\alpha \vee \gamma) \wedge (\neg\alpha \vee \gamma \vee \beta) \quad (2.1)$$

$$(\neg\alpha \vee \beta \vee \neg\gamma) \wedge (\neg\alpha \vee \gamma) \wedge \alpha \wedge \neg\beta \quad (2.2)$$

The CNF 2.1 is satisfiable because the assignment $\alpha = \text{true}, \beta = \text{true}, \gamma = \text{true}$ evaluates every clause to true and therefore evaluates the CNF to true. The CNF 2.2 is not satisfiable because the third and fourth clauses $\alpha \wedge \neg\beta$ evaluate α to true and β to false. Now, it is impossible to evaluate the first two clauses to true. Hence, CNF 2.2 is unsatisfiable.

Through a process called *clausification*, it is possible to transform every propositional formula into a CNF and therefore to a SAT problem [CESS08]. The SAT problem is NP-hard and therefore hard to solve on average [CESS08]. As the first NP-hard problem discovered, it gained a lot of attraction from researchers, and was about 20 years ago primarily a theoretical subject. However, that changed based on the fact that ways were found to model some real-world problems to propositional formulas. They can then be solved by SAT solving engines [CESS08]. One of the first big tasks for SAT problems were verification tools, and they are still the most important subject of interest [CESS08].

Tools or engines that can solve SAT problems are called SAT solvers. Such SAT solvers are vital components for verification tools [CESS08]. For example, they are used to verify hardware systems.

SAT solvers use different techniques to solve SAT problems. One is called boolean constraint propagation. Boolean constraint propagation can be used if there are already values assigned to variables. Then, every clause will be investigated to see if all literals except one, are *false*. Afterward, we can assign the value *true* to the last remaining literal [CESS08].

Example 2.2 (Boolean Constraint Propagation).

$$(\alpha \vee \neg\beta) \wedge (\gamma \vee \beta) \quad (2.3)$$

We have the partial assignment $\alpha = \text{false}$ and the other variables are unknown. Then by investigating the first clause, we can assign the value false to β . By using the new propagated value for β , we can assign the value true to γ .

The development of SAT solvers and their efficiency is defined by the following significant steps. At first, the SAT solvers used a simple but complete backtracking algorithm [CESS08]. The next big algorithm is called Davis-Putnam-Logemann-Loveland, in short DPLL, which is an evolved backtracking algorithm by combining it with boolean constraint propagation [CESS08]. Modern SAT solvers use an improved and refined DPLL algorithm that is referenced as conflict driven SAT solving. The conflict driven SAT solving is not entirely recursive anymore and can learn new clauses [CESS08].

Unsatisfiable Core

A subset of clauses from a formula in CNF, that makes the formula unsatisfiable, is called the unsatisfiable core [DHN06]. The unsatisfiable core is called minimal when removing one clause would make the formula satisfiable [DHN06]. For one formula multiple unsatisfiable cores can exist [DHN06].

Example 2.3 (Unsatisfiable Core). *The following formula is unsatisfiable:*

$$(\alpha \vee \beta \vee \neg\gamma) \wedge (\neg\delta \vee \epsilon) \wedge (\epsilon \vee \delta) \wedge \neg\epsilon \wedge (\neg\beta \vee \neg\alpha)$$

An unsatisfiable core is $\{(\neg\delta \vee \epsilon), (\epsilon \vee \delta), (\neg\epsilon)\}$. If we remove any of the three clauses the formula would be satisfiable. Thus, the unsatisfiable core is minimal.

2.2.2 Satisfiability Modulo Theory

The massive attraction to SAT solvers made it possible to solve a problem with hundreds of thousand variables [DMB11]. However, sometimes applications or problems need more expressive logic such as first-order logic [BSS⁺09]. Generally, first-order logic formulas can not be solved directly because operations are undefined [BSS⁺09].

Example 2.4. *The following formula is in first-order logic:*

$$\alpha \leq (\beta + 1) \tag{2.4}$$

Operations in first-order logic are generally not defined. In contrast, we aim to determine if there exist any functions (for example $\leq, +$), such that the formula is satisfiable. But instead, most applications are interested whether the formula is satisfiable by handling the operations \leq as the known order over integer and $+$ as the addition of integers [BSS⁺09]. Such formulas in first-order logic, where the operation is fixed, are called background theories [BSS⁺09].

The background theories can be seen as a fragment of the first-order logic where the operations are already defined. Problems that consist of first-order logic with respect to background theories and are interested whether the given formula is satisfiable, are called satisfiability modulo theories (SMT). There exist background theories for integer types, real types, data structures, arithmetic operations and more [BSS⁺09].

To solve SMT problems, the eager approach is to transform the first-order formula directly to propositional logic and solve the resulting formula with the help of SAT solvers [BSS⁺09]. Modern engines are a combination of SAT solvers for solving propositional logic and specific theorem solvers one for each background theory. Tools or engines that solve SMT problems are called SMT solvers. The increasing performance of SAT solvers and the interest in the specific theorem solvers lead to an increased performance for SMT solver over the last two decades [BSS⁺09]. The better performance and technological advances of applications made the SMT solvers a vital component for software and hardware verification, type inference, static program analysis, test-case generation, scheduling, planning and graph problems [DMB11].

3. Unification of SAT and SMT Solvers for the Software Product Line Analysis

3.1 Motivating Example

The introduction of cross-tree constraints for feature models expands the variability but can lead to inconsistencies [ABKS13]. In this section, we show the different inconsistencies, also called defects, and the approach used in [BSRC10] to automate the feature model analysis. Figure 3.1 shows a simplified software product line for a car which contains different defects. The feature model is a modified version adapted from [KAT16a].

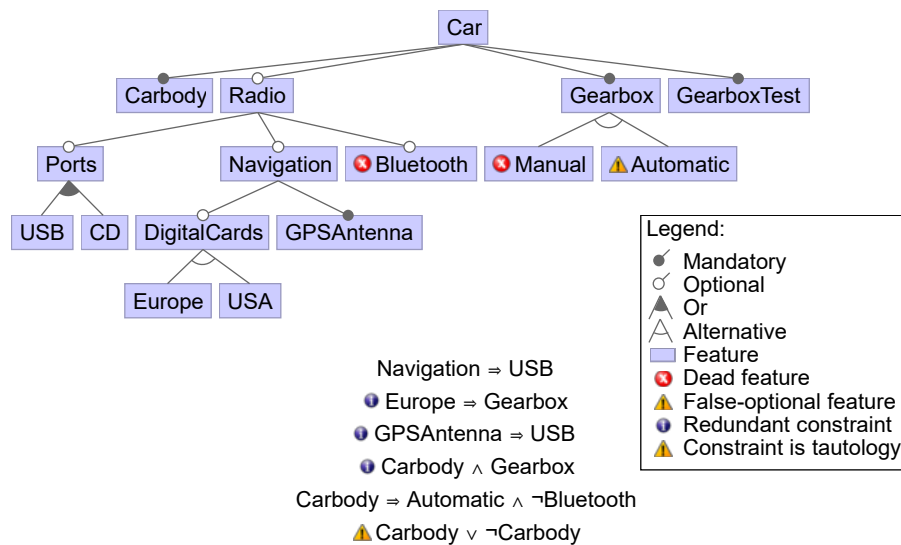


Figure 3.1: Feature model with visualized inconsistencies, adapted from [KAT16a]

The feature model in Figure 3.1 contains seven inconsistencies of four different types. The first type is a *dead feature* which is not part of any valid product of the feature

model [ABKS13, BSRC10]. The feature model in Figure 3.1 contains the two dead features *Bluetooth* and *Manual*. Both are dead because of the last constraint $Carbody \Rightarrow Automatic \wedge \neg Bluetooth$. *Carbody* is a core feature and implies that *Automatic* is always selected, preventing the alternative *Manual* to be selected. Thus, *Manual* is dead. *Carbody* also implies that *Bluetooth* cannot be selected making it dead as well. Dead features are severe defects because the feature is unusable.

The next type is a feature that is marked as optional but is always selected when their parent is selected [ABKS13, BSRC10]. Such a feature is called *false-optional features*. The only false-optional feature in Figure 3.1 is the feature *Automatic* because it is implied by the core feature *Carbody*, making *Automatic* occur in every product. False-optional features are not as severe as dead features, but at least smelly because they prevent us from selecting some feature combinations in the feature model [MTS⁺17].

Another type of an inconsistency is the *redundant constraint* which is a constraint that is already modeled in another way [ABKS13, BSRC10]. The feature model in Figure 3.1 contains three redundant constraints. The first constraint $Europe \Rightarrow Gearbox$ is redundant because the feature *Gearbox* is a core feature. The second constraint $GPSAntenna \Rightarrow USB$ is redundant because of the constraint $Navigation \Rightarrow USB$. If we want to select *GPSAntenna*, we need to select the parent feature *Navigation* as well, which already implies USB. Therefore, the implication from *GPSAntenna* to *USB* is redundant. The fourth constraint in Figure 3.1 $Carbody \wedge Gearbox$ is redundant because *Carbody* and *Gearbox* are both modeled as mandatory and are children of the root feature. Redundant constraints do not change the validity of the configurations [MTS⁺17]. They bloat the feature model and can be removed for the sake of brevity [MTS⁺17].

The next type is a *tautological constraint*. A Tautological constraints is always satisfied. The last constraint in Figure 3.1 $Carbody \vee \neg Carbody$ is a tautology. Tautological constraints do not affect the validity of the feature model. Removing the constraints is the best way to prevent the feature model from becoming more complex because of tautological constraints [MTS⁺17].

Now we know all inconsistencies of the feature model in Figure 3.1, but there is still an important type of inconsistency, the void feature model. A feature model is void when it contains no valid product, making the feature model entirely unusable [BSRC10].

The manual detection of inconsistencies is hard, time-consuming, and for large feature models infeasible [BSRC10]. An automated analysis is needed to verify the feature model and to warn the user about inconsistencies [ABKS13]. It is not only possible to show which defects exist, but also which circumstances lead to them. The information that shows us why a particular defect appears is called *explanation* [Gün17]. Explanations can detect the actuators of a particular defect [Gün17] and assist us in fixing the defect. As an example, the explanation for the dead feature *Bluetooth* is shown in Figure 3.2. Without any effort, we can see that *Bluetooth* is dead because of the last constraint and that *Carbody* is mandatory. Therefore, we can fix the defect by removing the last constraint or setting *Carbody* to optional.

There are several approaches to analyze feature models automatically. These approaches are based on propositional logic, description logic, or constraint program-

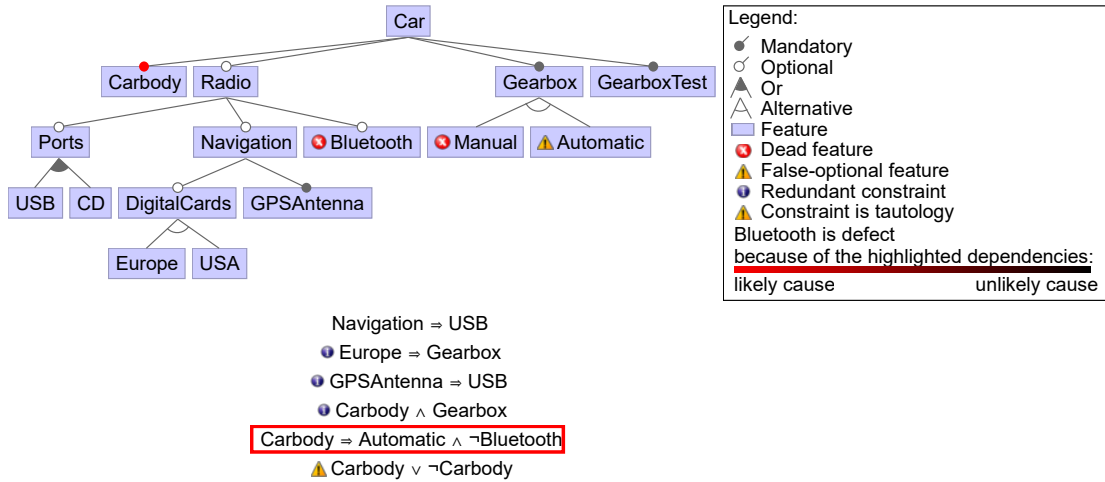


Figure 3.2: Feature model with an explanation for the dead feature *Bluetooth*

ming [BRCTS06]. In our case, we use propositional logic by transforming the feature model's structure and constraints to propositional logic. As a requirement for the automated analysis, we need the complete feature model in propositional logic. In 2002, Mannion connected feature models with propositional logic by proposing rules to transform a feature model into propositional logic [Man02]. Therefore, we fulfilled the requirement of having the feature model's structure and constraints in propositional logic. To finally automate the feature model analysis, two components are needed.

We need specific analyses, one for each kind of inconsistency. The second component is a SAT solver to perform the analyses depending on propositional expressions. However, SAT solvers are not the only solvers that can solve propositional expressions. The significant increase in performance of SMT solvers in the last two decades [DMB11] makes the SMT solver a promising candidate for the automated analysis. Additionally, the SMT solvers also provide more functionality giving us the essentials to compute attribute ranges. Therefore, we are interested in SMT solvers performing the automated analysis of feature models and computing explanations of defects. Furthermore, we aim to compare both kinds of solvers on these tasks.

SAT and SMT solvers have different implementations making it costly to write an analysis for every solver. By introducing an abstract data type, representing SAT or SMT solvers, it is possible to hide the different implementations. With the help of the abstract data type it is possible to formally define the different analyses.

3.2 Abstract Data Type for the Software Product Line Analysis

In this section, we propose a concept for an abstract data type for SAT and SMT solvers which is defined by a standard set of operations. The data type is used to write general analyses for SAT and SMT solvers and to replace existing or add new solvers easily. Additionally, the access to the native solvers is still possible and can be used to write an analysis with solver specific operations.

The data type makes it also possible to evaluate different SAT and SMT solvers that share the same fundamental operations. This results in an evaluation which not only depends on solving the problem but also on the workload of creating or altering a solver. Editing the solver's internal formula is also an essential aspect because many analyses depend on changes of the formula.

Every solver gets a SAT or SMT problem on creation which is not modifiable. That allows us to work with multiple solvers on the same problem without making the problem inconsistent. We decided to do that because we need to translate the feature model into the conjunctive normal form before giving it to the solver. This translation is costly and time-consuming. Therefore, multiple transformations should be prevented. The idea is to transform the feature model into the conjunctive normal form only once and give it to the multiple solvers as input.

Definition 3.1 (*IncrementalSolver*). CNF_N is the set of all propositional formulas in conjunctive normal form that can be build using the set of literals N . We assume that P represents the set of all clauses. The abstract data type *IncrementalSolver* is defined as follows:

Signature:

$init : CNF \rightarrow \text{IncrementalSolver}$

$push : \text{IncrementalSolver} \times P \rightarrow \text{IncrementalSolver}$

$pop : \text{IncrementalSolver} \rightarrow P$

$isSatisfiable : \text{IncrementalSolver} \rightarrow N$

Axioms:

$A1: \forall s : \text{IncrementalSolver}, \forall p : P : pop(push(s, p)) = p$

$A2: \forall s : \text{IncrementalSolver}, \forall c : CNF : pop(init(c)) = \emptyset$

Asserts:

$R1: \forall l_i, l_j : N, i \neq j : l_i \neq l_j$

$R2: \forall l : N : l \text{ is positive } \vee l \text{ is negative}$

Now, we explain the different signatures for the abstract data type. The *init* function allows us to create a solver by giving a propositional formula in conjunctive normal form as input. *push* and *pop* allow us to modify the solver by adding or removing clauses. *isSatisfiable* checks whether the internal formula is satisfiable and returns a set of literals when the formula is satisfiable, otherwise \emptyset . The set contains all literals with their assignments. The axiom *A1* defines the last-in-first-out behavior for the *push* and *pop* functions. Next, *A2* defines that it is not possible to modify the formula given on *init*. *R1* and *R2* assert that no duplicate literals exist in a solution and that a literal is either positive or negative.

With the abstract data type, we define general analyses for the inconsistencies of feature models. The general analyses can be used to evaluate both types of solvers. The use of both for the analyses is an exciting task because it can help to choose the most suitable solver for each general analysis.

3.3 Feature Model Analyses

In this section, we present the different analyses in detail. As part of our thesis, we aim to evaluate analyses for the void feature model, core and dead features, false-optional features, redundant constraints, and tautological constraints.

3.3.1 Void Feature Model Analysis

At the beginning of this chapter, we already faced various inconsistencies. Now, we formally define the inconsistencies and detailed analyses with the help of the abstract data type `IncrementalSolver`. We start with the most significant analysis, the void feature model analysis.

Definition 3.2 (Void Feature Model). *A feature model which contains no valid product is void [BSRC10]. Assume that FM is a feature model.*

$$\text{void}(FM) := \text{init}(CNF_{FM}).\text{isSatisfiable}() = \emptyset$$

The procedure of the detailed analysis for the void feature model is depicted in Figure 3.3. The void feature model analysis returns a satisfying assignment for every variable if the feature model is not void.

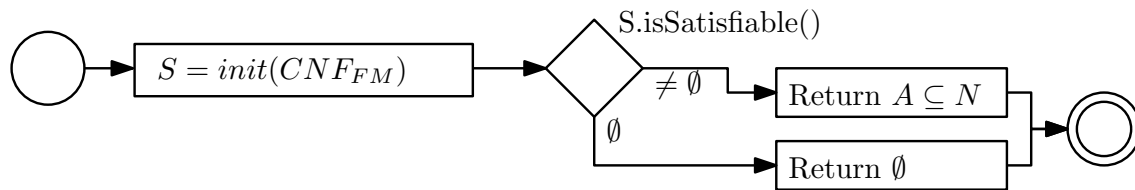


Figure 3.3: The procedure of the void feature model analysis

3.3.2 Core and Dead Feature Analysis

The most severe inconsistency besides the void feature model is the dead feature because it means that a part of the feature model cannot be used. The core features are the exact opposite of dead features because they appear in every solution. Thus, they are not defects but finding them gives us the knowledge which features should be prioritized in the development of the software product line [BSRC10]. The analyses for dead and core features are almost identical. Therefore, we write one analysis for both of them. We also present three different types of analyses. The first is a general analysis for all solvers without any optimizations. The analysis is based on the core and dead features definition [BSRC10, KAT16b]. Küchlin and Kaiser proposed optimizations for SAT solving algorithms that depend on filtering already known literals and modifying the literal selection strategy of a solver [KK01]. Janota, adapted both techniques in his work [Jan10] to refine his SAT solving environment. Janota, Küchlin, and Kaiser concluded that the proposed optimizations significantly improve the SAT solving process [KK01, Jan10]. Therefore, we adapted the optimizations for the use within the core and dead feature analysis. We created an optimized analysis that uses only the filtering technique. Afterward, we created an analysis that uses both optimizations. Before we explain the analyses in detail, we define the core and dead features as follows:

Definition 3.3 (Core Feature). A core feature is a feature which is part of every product of the feature model [KAT16b, BSRC10]. Assume that FM is a feature model and $f \in N_{FM}$ is a feature.

$$core(f) := init(CNF_{FM}).push(\neg f).isSatisfiable() = \emptyset$$

Definition 3.4 (Dead Feature). A dead feature is a feature which is not part of any product of the feature model [KAT16b, BSRC10]. Assume that FM is a feature model and $f \in N_{FM}$ is a feature.

$$dead(f) := init(CNF_{FM}).push(f).isSatisfiable() = \emptyset$$

Now, we present the three different analyses in detail. The first one is the unoptimized analysis depicted in Figure 3.4. The core features are detected by iterating every feature and assuming it to be *false*. If the formula is unsatisfiable, it is clear that the feature is a core feature. The procedure is almost the same for dead features. Instead of assuming the feature to be *false*, the feature are assumed to be *true* [KAT16a].

The second analysis is an improved version of the first one. We apply the filtering introduced by Küchlin, Kaiser, and Janota [KK01, Jan10] to reduce the number of iterations at least by half. At the beginning, a satisfying assignment is calculated and saved in a list. If the value of a feature is *true*, we know that the feature is a potential core feature and definitely not dead. In contrast, if the value of a feature is *false*, we know that the feature is a potential dead feature and definitely not a core feature. Thus, we only need to check features with the value *true* whether they are core features and features with the value *false* whether they are dead features. When a feature is identified as neither dead nor core, then it means the internal formula of the solver is satisfiable, and a solution can be retrieved. Now, we can compare the first solution with the new solution. If the values of a feature are different, we know that the feature is neither dead nor core, and set the value of the feature to *IGNORE*. Features marked as *IGNORE* will be skipped. The procedure for the analysis using the filtering is depicted in Figure 3.5.

For the next analysis, we apply the filtering and the modification of the literal selection strategy, proposed by Küchlin, Kaiser, and Janota [KK01, Jan10]. It is possible to retrieve two solutions at the beginning, by varying the selection strategy. SAT solvers use a backtracking algorithm to calculate a satisfying assignment. To do so, they need to assume literals to be either *true* or *false*. The SAT solver uses the selection strategy to decide between *true* and *false*. If the selection strategy is *POSITIVE*, the solver always assumes a literal to be *true*. To the contrary, when the selection strategy is *NEGATIVE*, the solver always assumes a literal to be *false*. Now, both solutions are compared, and features with different values are set to *IGNORE*. With the help of the two fundamentally different selection strategies, there is a high chance that features, which are neither core nor dead, have different values leading to a lot of features which can be ignored. Afterward, the analysis behaves like the analysis that uses only filtering. The procedure for the analysis using both optimizations is depicted in Figure 3.6.

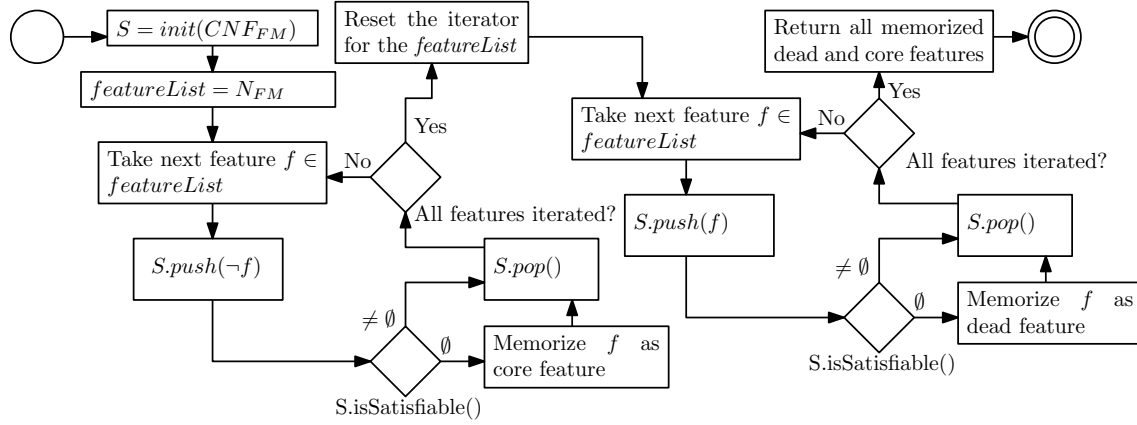


Figure 3.4: The procedure of the unoptimized core and dead feature analysis

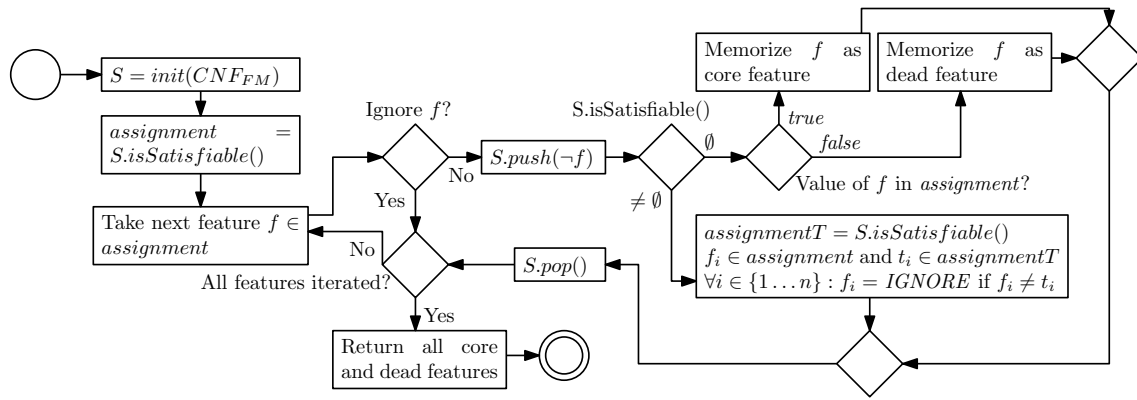


Figure 3.5: The procedure of the core and dead feature analysis using filtering

3.3.3 False-Optional Feature Analysis

This section is about false-optional features and their analyses. The first one is an unoptimized analysis, which calculates the false-optional features like they are defined by Kowal et al. [KAT16b]. Next, we extend the analysis by applying the filtering proposed by Küchlin, Kaiser, and Janota [KK01, Jan10]. Furthermore, we combine the filtering with the modification of the literal selection strategy proposed by Küchlin, Kaiser, and Janota [KK01, Jan10].

Definition 3.5 (False-Optional Feature). *A false-optional feature is modeled as optional in the feature model but in fact, is always selected when its parent is selected [KAT16b, BSRC10]. We assume that FM is a feature model and $f \in N_{FM}$ is a feature with $\omega(f) = 0$. The feature $p \in FN_{FM}$ is the parent of the feature f .*

$$falseOptional(f) := init(CNF_{FM}).push(p).push(\neg f).isSatisfiable() = \emptyset$$

Now, we explain three different analyses and highlight their improvements. The unoptimized analysis is depicted in Figure 3.7. At first, we retrieve every potential false-optional feature. Then, for every potential false-optional feature, we push their negation and their parent to the solver. If the resulting formula is unsatisfiable, the potential false-optional feature is, in fact, false-optional because no case exists such that the parent is selected, but the child feature is not [KAT16a].

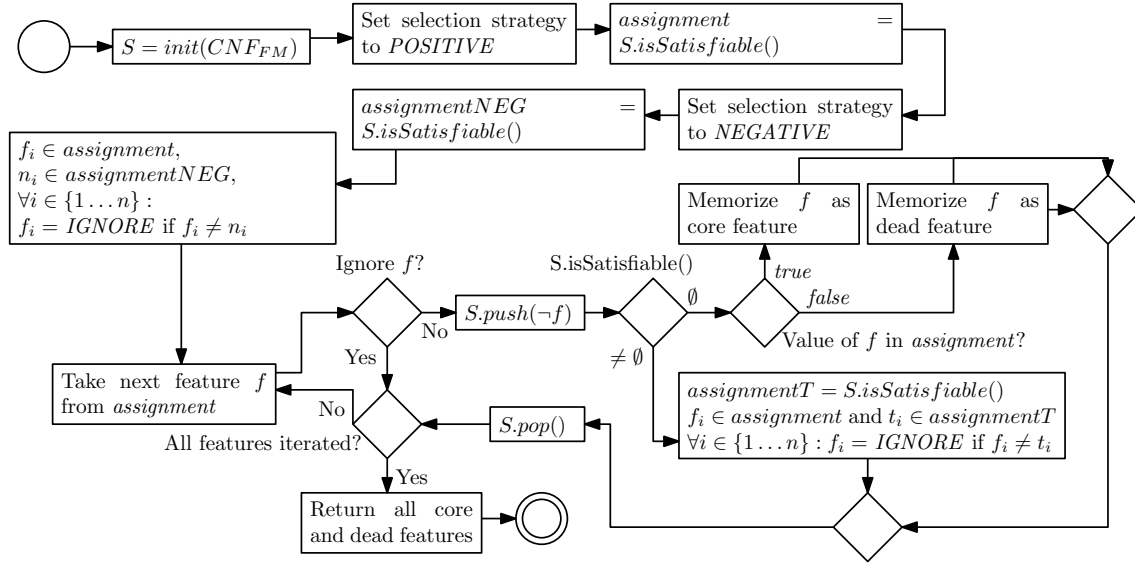


Figure 3.6: The procedure of the core and dead feature analysis using filtering and different selection strategies

We optimize the first analysis by applying the filtering proposed by K uchlin, Kaiser, and Janota [KK01, Jan10]. This can be achieved by saving multiple solutions and checking that the potential false-optional feature and his parent have the same value in every solution before pushing it to the solver. The optimized version is depicted in Figure 3.8. At the beginning, a solution is retrieved by $S.\text{isSatisfiable}()$ and added to the solutions. If the potentially false-optional feature, which is currently checked, is not false-optional, then another solution can be retrieved and added to the solutions. With the help of the saved solutions, we can skip every potential false-optional feature which has a different value than his parent.

By modifying the selection strategy at the beginning, we can already retrieve a solution and add it to the solver [KK01, Jan10]. The solution is retrieved with the selection strategy set to *RANDOM*. If the strategy is set to *RANDOM*, it is more likely that a feature and his parent have different values. The resulting analysis is depicted in Figure 3.9.

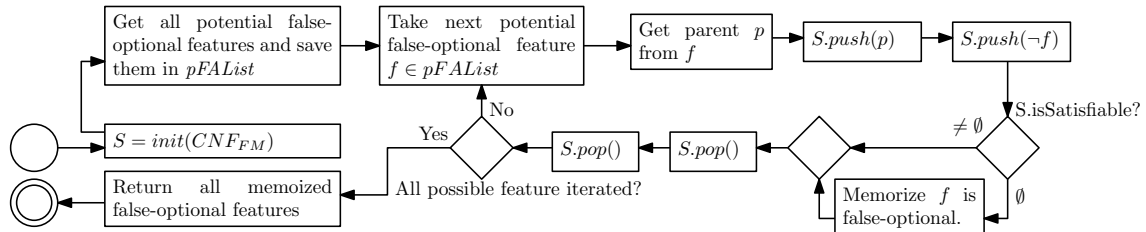


Figure 3.7: The procedure of the unoptimized false-optional feature analysis

3.3.4 Redundant Constraint Analysis

The next inconsistency that we faced in our feature model in Figure 3.1 was a redundant constraint. Such redundant constraints are pieces of information which are already modeled in another way through the structure of the feature model or

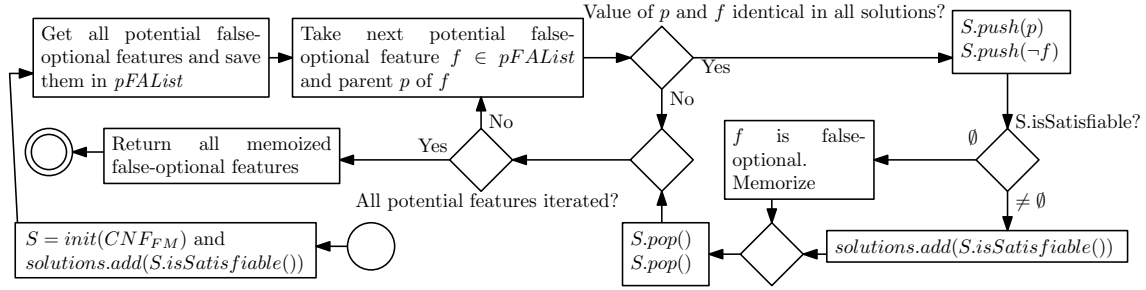


Figure 3.8: The procedure of the false-optional feature analysis using filtering

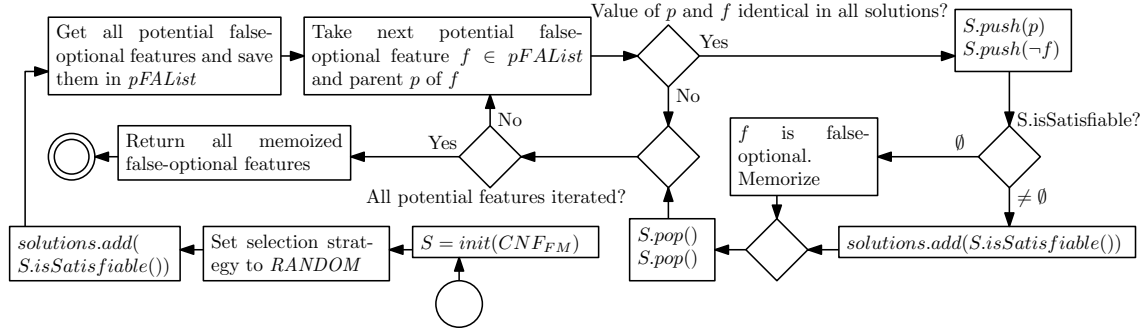


Figure 3.9: The procedure of the false-optional feature analysis using filtering and a different selection strategy

other constraints. Redundant constraints are not grave defects and only increase the effort to maintain the feature model [BSRC10].

Definition 3.6 (Redundant Constraint). *A cross-tree constraint is redundant when its semantic information was already modeled in another way [vdML04]. FM is a feature model and $c_j \in \Psi_{FM}$. FM_j is the feature model without the cross tree constraint c_j . The definition for redundant constraints given by Kowal et al. is not suitable for the data type **IncrementalSolver** [KAT16b]. Therefore, we need to convert the definition to a form, which can be expressed by the **IncrementalSolver**.*

$$\begin{aligned}
 \text{redundant}(c_j) &\equiv \text{TAUT}(FM_j \Leftrightarrow FM_j \wedge c_j) \text{ [KAT16b]} \\
 &\equiv \neg \text{SAT}(\neg((FM_j \Rightarrow FM_j \wedge c_j) \vee (FM_j \wedge c_j \Rightarrow FM_j))) \\
 &\equiv \neg \text{SAT}(\neg(FM_j \Rightarrow FM_j \wedge c_j) \wedge \neg(\neg(FM_j \wedge c_j) \vee FM_j)) \\
 &\equiv \neg \text{SAT}(\neg(\neg FM' \vee (FM' \wedge c_j))) \\
 &\equiv \neg \text{SAT}(\neg(\neg FM' \vee c_j)) \\
 &\equiv \neg \text{SAT}(FM' \wedge \neg c_j)
 \end{aligned}$$

The resulting definition for redundant constraints coincides with the results from Günther [Gün17]. Now, we can adapt the definition for the **IncrementalSolver**. We cannot push the constraint directly to the **IncrementalSolver** because a cross-tree constraint can contain multiple clauses. Therefore, we need to push all clauses from the constraint. Let $k_1 \dots k_n$ be the clauses of CNF_{-c_j} .

$$\text{redundant}(c_j) := \text{init}(CNF_{FM_j}).\text{push}(k_1).\dots.\text{push}(k_n).\text{isSatisfiable}() = \emptyset$$

The analysis for the redundant constraints is the most complex analysis. We begin with a solver which has only the feature model's structure in CNF as a problem.

Then all constraints are saved as a list. Next, we push all constraints to the solver. Only the constraint we will check is removed from the solver which is a problem because the `IncrementalSolver` provides no random access to the pushed clauses. Therefore, we need to pop all pushed constraints till we reach the constraint we want to check. Afterward, we need to push the constraints back to the solver. For every clause of our constraint, we push the negated formula to the solver. If the solver's internal formula is unsatisfiable, the constraint is marked as *REDUNDANT* and not pushed back to the solver to prevent redundancies in the opposite direction. If the constraint is not redundant, it is pushed back to the solver, and the next constraint can be checked. An optimized version can be achieved by sorting the list with the constraints at the start by their number of clauses in ascending order. The chance of a constraint to be redundant is higher for small constraints than for large constraints because large constraints contain multiple clauses which all need to be redundant to make the constraint redundant. For a better performance, we do not begin with the first constraint, instead, we start with the last constraint to minimize the effort to remove the overlying constraints in every iteration. The detailed procedure is depicted in Figure 3.10.

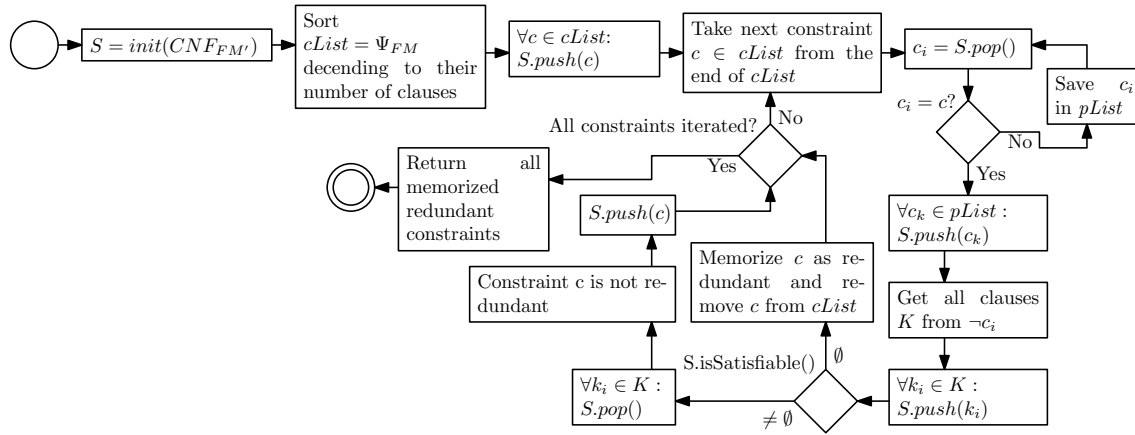


Figure 3.10: Procedure of the redundant constraint analysis with optimizations

3.3.5 Tautological Constraint Analysis

The last defect we saw in Figure 3.1 is the tautological constraint. If a constraint is a tautology, it can be decided without the context of the feature model and is always satisfied. Therefore, the constraint does not improve the variability of the feature model and should always be removed.

Definition 3.7 (Tautological Constraint). *A constraint is a tautology if it is always satisfied [BSRC10]. We assume that FM is a feature model and $c \in \Psi_{FM}$ is a constraint.*

$$\text{taut}(c) := \text{init}(\text{CNF}_{-c}).\text{isSatisfiable}() = \emptyset$$

The analysis for tautological constraints is depicted in Figure 3.11. The detection of tautological constraints is quite simple. At first, we retrieve a list containing all constraints from the feature model. Now, we check every constraint separately. We start by transforming the negated constraint $\neg c$ into the conjunctive normal

form $CNF_{\neg c}$. Next, we create an **IncrementalSolver** with $CNF_{\neg c}$ as input. If the **IncrementalSolver** returns unsatisfiable, then the constraint c is a tautology because there is no satisfying assignment such that the negated constraint is satisfiable.

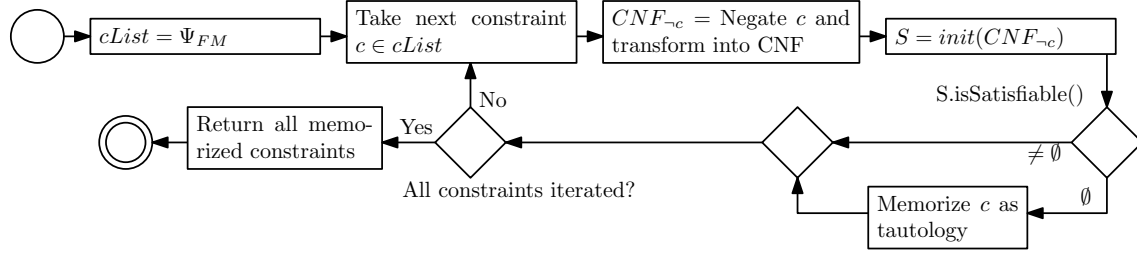


Figure 3.11: Procedure of the tautological constraint analysis

3.4 Automated Feature Model Analysis

Now, with the help of the abstract data type **IncrementalSolver** and the analyses for the various inconsistencies, we can automate the feature model analysis. At first, we transform the current feature model's structure into the conjunctive normal form with its constraints CNF_{FM} and again without constraints $CNF_{FM'}$. The first analysis checks whether the feature model is not void. After that, the four analyses for core and dead features, false-optional features, redundant constraints, and tautological constraints are performed. The results can be used to warn the user about the inconsistencies like shown in Figure 3.1. The procedure for the automated analysis is depicted in Figure 3.12.

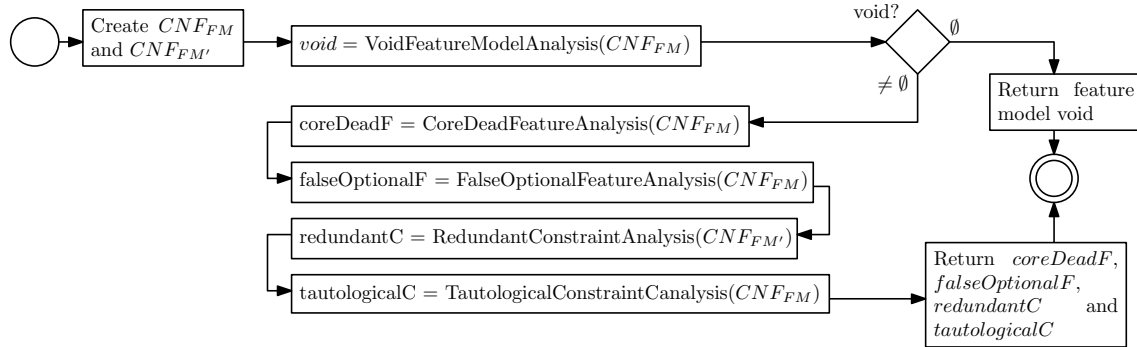


Figure 3.12: Procedure of the automated feature model analysis

3.5 IncrementalSolver Extension for Explanations

Currently, the data type *IncrementalSolver* cannot explain defects. Therefore, we need an extension to do that. With our extension, it will then be possible to calculate the explanations. Explanations are nothing else than the unsatisfiable core of an unsatisfiable formula. The unsatisfiable core is the set of clauses that causes the current formula in conjunctive normal normal form to be unsatisfiable. Therefore, we extend our data type **IncrementalSolver** with a function to calculate the unsatisfiable core.

Extension 3.1. We extend the data type `IncrementalSolver`.

Signature:

$unsatCore : \text{IncrementalSolver} \rightarrow CNF_{FM}$

Assert:

R3: $\forall m : \text{IncrementalSolver} : m.isSatisfiable() = \emptyset : m.unsatCore() \neq \emptyset$

R4: $\forall m : \text{IncrementalSolver} : m.isSatisfiable() \neq \emptyset : m.unsatCore() = \emptyset$

The function *unsatCore* retrieves the unsatisfiable core for the solver's current formula. The asserts *R3* and *R4* define that the unsatisfiable core can only be retrieved when the current formula of a solver is unsatisfiable. With the help of the extension, it is now possible to find explanations for core features, dead features, false-optional features, and redundant constraints. To do so, we only need to extend the defined analyses by calling *unsatCore* every time we detect a defect. The received unsatisfiable core is then the explanation for the particular defect.

3.6 Summary

We began by introducing the different inconsistencies that can appear when dealing with cross-tree constraints on feature models. We concluded that the detection of such defects by hand is infeasible. Therefore, we introduced the automated analysis on feature models which can detect such inconsistencies. Such an automated analysis is commonly performed by a SAT solver. We motivated the detection of defects using an SMT solver because of the significant improvements of SMT solvers. Additionally, we motivated the comparison of both types of solvers on the automated analysis of feature models. We concluded that the comparison requires a standard data type that can be used to evaluate both types of solvers. Therefore, we created the abstract data type `IncrementalSolver` which is defined over a common set of operations. Afterward, we formally defined every defect which was introduced at the beginning of the chapter with the help of the `IncrementalSolver`. We used the definitions to present an analysis for each kind of defect in detail. We adapted the optimizations proposed by Küchlin, Kaiser, and Janota [KK01, Jan10] to improve some analyses. We used the multiple analyses to describe the complete procedure of an automated analysis of a feature model. Next, we were also interested whether SMT solvers can compete with SAT solvers on finding explanations for the defined defects. Therefore, we created an extension for the `IncrementalSolver` with the functionality needed to find explanations.

4. Computing Attribute Ranges of Partial Configurations

In this section, we describe our concept for the computation of the ranges of a numerical attribute for a partial configurations. First, we show the relevance of those computations. Afterward, we discuss our proposal to the task, which is separated into an approximate and an exact computation.

4.1 Motivating Example

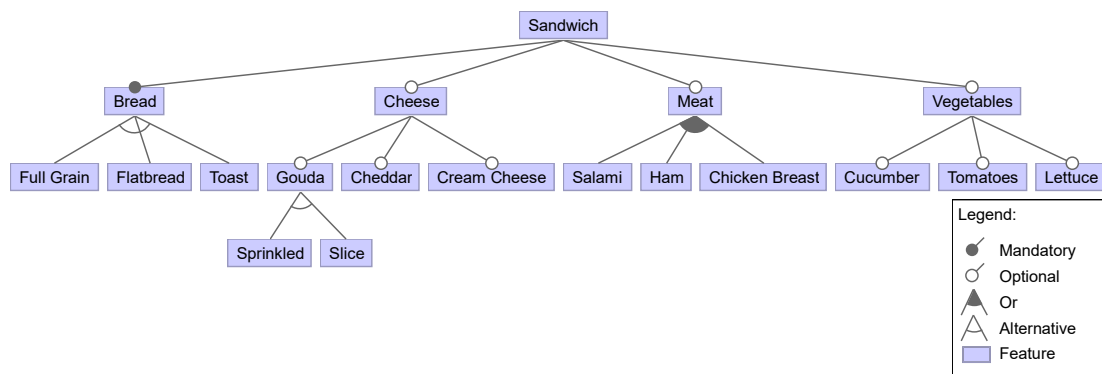


Figure 4.1: Motivating example of a feature model representing a sandwich

The given feature model in Figure 4.1 represents a family of sandwiches. A customer is able to retrieve the possible combinations of ingredients for his order from this model. However, other properties of the ingredients may be relevant for our customer as well. Additional properties might consist of the data available in Table 4.1. Using the attribute table, the customer is also able to calculate the price and calories for each sandwich and can account whether an ingredient is organic. Even though the model is relatively small (19 features), calculating the overall price or calories for multiple sandwich variants manually can take a considerable amount of time, as there are still 2,808 possible configurations.

Feature Name	Price (\$)	Calories (kcal)	Organic food
Full Grain	1,99	203	✓
Flatbread	1,49	50	-
Toast	1,79	313	-
Gouda Sprinkled	0,69	70	✓
Gouda Slice	0,49	72	✓
Cheddar	0,69	81	-
Cream Cheese	0,59	52	-
Salami	1,29	116	✓
Ham	0,99	92	-
Chicken Breast	1,39	56	✓
Cucumber	0,29	2	✓
Tomatoes	0,39	3	✓
Lettuce	0,19	2	✓

Table 4.1: Attributes attached to features of the *Sandwich* model

A customer would profit from automatic computations on the numerical attributes. First, we could compute the price of a specific sandwich, which results from the sum of prices of the selected ingredients. However, we are interested in another more complex computation an undecided customer profits from. Our goal is to acquire the minimal and maximal possible price and amount of calories given a few chosen ingredients. We call this the computation of attribute ranges given a partial configuration. An example in our model might be calculating the calorie ranges of the sandwich family with full grain bread and ham. Now, we explain the complexity of this problem.

Computing a possible maximal or minimal sum of a numerical attribute is complex because of the exponential growth of the number of configurations per feature. For a feature model M , calculating the sum of the attribute values needs $O(|N_M|)$ time, as in the worst case $\pi_{att}(f)$ for every $f \in N_m$ has to be added to the sum. The amount of configurations is $2^{|N|}$ (one binary decision per feature) in the worst case. Therefore, brute-forcing a solution needs $O(|N_m| * 2^{|N_m|})$ time. Let us now define the computation of attribute ranges given a partial configuration more formally.

4.2 The Problem

In this section, we formally define the algorithmic problem of our computations.

The input of our problem is defined as a numerical attribute att and a configuration $C = (M, S, U)$, where M is an extended feature model, S are the selected, and U the unselected features. Additionally, we define Val as the set of all valid configurations of M . For the remainder of this chapter, we assume every value $\pi_{att}(f)$ is in \mathbb{R} .

Our goal is now to compute the minimum, which results from the subset T of features forming a valid configuration. Additionally, T has to include all selected and exclude all unselected features and imply the smallest sum $\sum_{t \in T} \pi_{att}(t)$.

$$\text{Min}(\text{att}) = \sum_{t \in T} \pi_{\text{att}}(t) \text{ with}$$

$$T = \{T_1 \subseteq N_M \mid C_i = (M, T_i, U_i) \in \text{Val}, T_i \cap U = \emptyset, S \setminus T_i = \emptyset, \\ \forall T_i : (\sum_{t \in T_1} \pi_{\text{att}}(t) \leq \sum_{t' \in T_i} \pi_{\text{att}}(t'))\}$$

$T_i \cap U = \emptyset$ ensures that no T_i contains unselected features. $S \setminus T_i = \emptyset$ ensures that every selected feature is included in each T_i . Additionally, we aim to compute the maximum, which is similarly defined as the minimum, but $\sum_{t \in T} \pi_{\text{att}}(t)$ has to be the largest sum.

$$\text{Max}(\text{att}) = \sum_{t \in T} \pi_{\text{att}}(s) \text{ with}$$

$$T = \{T_1 \subseteq N_M \mid C_i = (M, T_i, U_i) \in \text{Val}, T_i \cap U = \emptyset, S \setminus T_i = \emptyset, \\ \forall T_i : (\sum_{t \in T_1} \pi_{\text{att}}(t) \geq \sum_{t' \in T_i} \pi_{\text{att}}(t'))\}$$

4.3 Computing Attribute Ranges using SMT

In this section, we explain and discuss the computation of our problem using an SMT solver, which is able to optimize variables. The main task is to map our input into a formula in first-order logic, which can be used to optimize for a given variable by an SMT solver.

For the sake of clarity, we split our overall formula γ into four different parts and explain them separately. The parts consist of a formula representing the dependencies of the feature model, (partial) configuration, and the optimization for our attribute.

First, we need to capture the logical constraints of the feature model M itself. These are given by the CNF_M .

To express the sets S and U of our configuration $C = (M, S, U)$, we create literals s_i for every $s_i \in S$ and $\neg u_i$ for every $u_i \in U$. By conjuncting the created literals to our formula γ , we ensure that the selected features are included and unselected ones are excluded.

At last, we build the part of the formula representing the attributes. The idea here is to create a variable att_{feat} for every instance of the attribute in the feature model and optimize their sum. We define the formulas we add in the following way:

$$(\text{feat} \Rightarrow \text{att}_{\text{feat}} = \pi_{\text{att}}(\text{feat})) \text{ and} \\ (\neg \text{feat} \Rightarrow \text{att}_{\text{feat}} = 0)$$

This works as follows, the inclusion of $feat$ sets att_{feat} to $\pi_{att}(feat)$, increasing the attribute sum by its value in the process. Otherwise its value has no impact on the range.

We add these formulas for every feature with the attribute attached to it. Additionally, we define $N_{M_{att}}$ as the set only containing features which have our attribute attached. Now, we add our variable we are optimizing for, which represents the sum of our attributes values. Formally, we define the sum as:

$$sum = \sum_{f \in N_{M_{att}}} att_f$$

Again, we conjunct the formula for the attribute and their sum to γ . Now, we define the overall formula.

$$\begin{aligned} & CNF_M \wedge s_1 \wedge s_2 \wedge \cdots \wedge s_n \wedge \neg u_1 \wedge \neg u_2 \wedge \cdots \wedge \neg u_m \wedge \\ & (f_1 \Rightarrow att_{f_1} = \pi_{att}(f_1)) \wedge (\neg f_1 \Rightarrow att_{f_1} = 0) \wedge \cdots \wedge \\ & (f_k \Rightarrow att_{f_k} = \pi_{att}(f_k)) \wedge (\neg f_k \Rightarrow att_{f_k} = 0) \wedge \\ & (sum = att_{f_1} + att_{f_2} + \cdots + att_{f_k}) \end{aligned} \quad (4.1)$$

The resulting formula [Equation 4.1](#) is in first-order logic, because it only contains propositional expressions and equality formulas. As our next step, we show that every satisfying assignment of [Equation 4.1](#) is a valid configuration and the implied sum is correct.

Correctness

In this paragraph, we show that the result given by the SMT solver with our mapped first-order logic formula [Equation 4.1](#) is correct. To achieve this, two requirements need to be fulfilled. First, the full configuration which includes the features whose attribute values build our minimal/maximal sum and excludes the others, has to be valid. Second, the optimized variable sum has to be equal to the sum of the attribute values of the selected features. Additionally, sum has to be minimal/maximal, but this is up to the solver and not part of this thesis.

Theorem 4.1. *Every satisfying assignment for our formula [Equation 4.1](#) represents a valid configuration.*

Proof. Let $C = (M, S, U)$ be the configuration representing our satisfying assignment. We assume that C is not valid. It follows that, the formula $CNF_M \wedge s_1 \wedge \cdots \wedge s_n \wedge \neg u_1 \wedge \cdots \wedge \neg u_m$ with $s_i \in S$ and $u_i \in U$ is not satisfied. Therefore, [Equation 4.1](#) is not satisfied and our assumption leads to a contradiction. \square

Now, we show that, given a valid full configuration of an extended feature model, sum is equal to the sum of the attribute values of the selected features.

Theorem 4.2. *For every valid full configuration $C = (M, S, U)$ with $M = ((N, r, \omega, \lambda, \Pi, \Psi), A, \pi)$ being an extended feature model the following holds:*

$$sum = \sum_{f \in S} \pi_{att}(f).$$

Proof. By definition, the following is given:

$$sum = att_{f_1} + att_{f_2} + \cdots + att_{f_k} \text{ with } k = |N|$$

$$att_{f_i} = \begin{cases} \pi_{att}(f_i) & \text{if } f_i \in S \\ 0 & \text{if } f_i \in U \end{cases}$$

This implies:

$$\begin{aligned} sum &= att_{f_1} + att_{f_2} + \cdots + att_{f_k} = \sum_{i=1}^k att_{f_i} = \\ &= \sum_{f \in S} att_f + \sum_{f \in U} att_f = \\ &= \sum_{f \in S} \pi_{att}(f) + \sum_{f \in U} 0 = \\ &= \sum_{f \in S} \pi_{att}(f) \end{aligned}$$

□

Example

To further improve the understanding of our computation using SMT we map our example Figure 4.1 into an instance of Equation 4.1. In our scenario the customer is deciding on his sandwich. He has an appetite for ham and also has not eaten full grain bread in a long amount of time. Therefore, his sandwich definitely includes both. However, he is allergic to tomatoes. Now, he wants to know the minimal and maximal possible calories of a sandwich that has full grain bread, ham, and no tomatoes. As a result, the partial configuration $C_{Sandwich} = (M_{Sandwich}, S_{Sandwich}, U_{Sandwich})$ with $S_{Sandwich} = \{Full\ Grain, Ham\}$ and $U_{Sandwich} = \{Tomatoes\}$ and the attribute calories (*cal*) form our input.

First, we want to create the conjunctive normal form of our sandwich feature model.

$$\begin{aligned}
& \text{Sandwich} \\
& \wedge (\text{Vegetables} \vee \neg \text{Cucumber}) \wedge (\text{Vegetables} \vee \neg \text{Tomatoes}) \\
& \wedge (\text{Vegetables} \vee \neg \text{Lettuce}) \\
& \wedge (\text{Cheese} \vee \neg \text{Gouda}) \wedge (\text{Cheese} \vee \neg \text{Cheddar}) \wedge (\text{Cheese} \vee \neg \text{Cream C.}) \\
& \wedge (\text{Bread} \vee \neg \text{Full Grain}) \wedge (\text{Bread} \vee \neg \text{Flatbread}) \wedge (\text{Bread} \vee \neg \text{Toast}) \\
& \wedge (\text{Full Grain} \vee \neg \text{Flatbread}) \wedge (\text{Full Grain} \vee \neg \text{Toast}) \wedge (\text{Flatbread} \vee \neg \text{Toast}) \\
& \wedge (\text{Meat} \vee \neg \text{Salami}) \wedge (\text{Meat} \vee \neg \text{Ham}) \wedge (\text{Meat} \vee \neg \text{Chicken B.}) \\
& \wedge (\text{Salami} \vee \text{Ham} \vee \text{Chicken B.} \vee \neg \text{Meat}) \\
& \wedge (\text{Sandwich} \vee \neg \text{Bread}) \wedge (\text{Sandwich} \vee \neg \text{Cheese}) \\
& \wedge (\text{Sandwich} \vee \neg \text{Meat}) \wedge (\text{Sandwich} \vee \neg \text{Vegetables}) \\
& \wedge (\text{Bread} \vee \neg \text{Sandwich}) \wedge (\text{Gouda} \vee \neg \text{Sprinkled}) \wedge (\text{Gouda} \vee \neg \text{Slice}) \\
& \wedge (\text{Sprinkled} \vee \text{Slice} \vee \neg \text{Gouda}) \wedge (\neg \text{Sprinkled} \vee \neg \text{Slice})
\end{aligned}$$

Next, we conjunct our selected and unselected features as literals to the formula:

$$\text{Full Grain} \wedge \text{Ham} \wedge \neg \text{Tomatoes}$$

For our last step, we conjunct our attribute terms starting with implications representing the calories of each feature:

$$\begin{aligned}
& (\text{Full Grain} \Rightarrow \text{cal}_{\text{Full Grain}} = 203) \wedge (\neg \text{Full Grain} \Rightarrow \text{cal}_{\text{Full Grain}} = 0) \wedge \\
& (\text{Flatbread} \Rightarrow \text{cal}_{\text{Flatbread}} = 50) \wedge (\neg \text{Flatbread} \Rightarrow \text{cal}_{\text{Flatbread}} = 0) \wedge \\
& (\text{Toast} \Rightarrow \text{cal}_{\text{Toast}} = 313) \wedge (\neg \text{Toast} \Rightarrow \text{cal}_{\text{Toast}} = 0) \wedge \\
& (\text{Sprinkled} \Rightarrow \text{cal}_{\text{Sprinkled}} = 70) \wedge (\neg \text{Sprinkled} \Rightarrow \text{cal}_{\text{Sprinkled}} = 0) \wedge \\
& (\text{Slice} \Rightarrow \text{cal}_{\text{Slice}} = 72) \wedge (\neg \text{Slice} \Rightarrow \text{cal}_{\text{Slice}} = 0) \wedge \\
& (\text{Cheddar} \Rightarrow \text{cal}_{\text{Cheddar}} = 81) \wedge (\neg \text{Cheddar} \Rightarrow \text{cal}_{\text{Cheddar}} = 0) \wedge \\
& (\text{Cream C.} \Rightarrow \text{cal}_{\text{Cream C.}} = 52) \wedge (\neg \text{Cream C.} \Rightarrow \text{cal}_{\text{Cream C.}} = 0) \wedge \\
& (\text{Salami} \Rightarrow \text{cal}_{\text{Salami}} = 116) \wedge (\neg \text{Salami} \Rightarrow \text{cal}_{\text{Salami}} = 0) \wedge \\
& (\text{Ham} \Rightarrow \text{cal}_{\text{Ham}} = 92) \wedge (\neg \text{Ham} \Rightarrow \text{cal}_{\text{Ham}} = 0) \wedge \\
& (\text{Chicken B.} \Rightarrow \text{cal}_{\text{Chicken B.}} = 56) \wedge (\neg \text{Chicken B.} \Rightarrow \text{cal}_{\text{Chicken B.}} = 0) \wedge \\
& (\text{Cucumber} \Rightarrow \text{cal}_{\text{Cucumber}} = 2) \wedge (\neg \text{Cucumber} \Rightarrow \text{cal}_{\text{Cucumber}} = 0) \wedge \\
& (\text{Tomatoes} \Rightarrow \text{cal}_{\text{Tomatoes}} = 3) \wedge (\neg \text{Tomatoes} \Rightarrow \text{cal}_{\text{Tomatoes}} = 0) \wedge \\
& (\text{Lettuce} \Rightarrow \text{cal}_{\text{Lettuce}} = 2) \wedge (\neg \text{Lettuce} \Rightarrow \text{cal}_{\text{Lettuce}} = 0) \wedge
\end{aligned}$$

and the overall sum of calories:

$$\text{sum}_{\text{cal}} = \text{cal}_{\text{Full Grain}} + \text{cal}_{\text{Flatbread}} + \text{cal}_{\text{Toast}} + \dots + \text{cal}_{\text{Lettuce}}$$

The formula resulting from the conjunction of our parts can be used to compute the minimum and maximum by optimizing for sum_{cal} with an SMT solver. For the record, in this case our calories range is: $\min(\text{cal}) = 295$ and $\max(\text{cal}) = 676$.

Discussion of the Solution

As we were able to map our algorithmic problem into a first-order logic formula, it is indeed possible to compute ranges of numerical attributes for partial configurations using an SMT solver. However, even though the solver always returns exact results, optimizing variables is a complex task to solve. Our formula shows linear growth with the occurrence of our attribute, as for every occurrence two implications are added. Additionally, the number of clauses of our CNF_{FM} is bounded by $O(n^2)$ with $n = |N_{FM}|$. Each clause of CNF_{FM} contains up to n literals [TBK09].

Let us consider a new example, our customer enjoyed his sandwich and now moves on to an even more difficult decision. He has to decide on a new car and instead of 19 features the car's feature model consists of 1,000 features. As a reminder this model allows for up to 2^{1000} configurations. Our customer is very undecided, he is still considering four different engines, six fittings, eight seat types, and three air conditioning systems. For his research, he does not need exact prices but he wants to know an approximate influence on the price for every component he is undecided on. For every calculation using SMT, multiple SAT requests are performed. Therefore, the computation using an SMT solver might become too costly for large feature models.

To ensure the possibility of computing several ranges in a short amount of time, we provide an heuristic to our algorithmic problem, which only approximates the ranges but should return a solution in linear time.

4.4 Computing Attribute Ranges using an Heuristic

In this section, we discuss our computation of attribute ranges with an heuristic. In this case, the focus is on performance instead of exact results. For the computation, we propose a recursive algorithm, that returns correct results for feature models without cross-tree constraints and estimations for models containing them.

The idea is to recursively calculate the overall attribute value of a sub-tree and add favorable or mandatory ones. By favorable we mean positive values for the maximum or negative values for the minimum. For example, a sub-tree whose overall value is negative would be favorable when computing the minimum. A sub-tree being mandatory can result from the tree structure (e.g. $\omega(\text{root}_{\text{sub-tree}}) = 1$) or the selected features.

During this the following case distinction, we call the configuration, inducing our maximum or minimum respectively, the *target configuration*. Now, we want to obtain rules on which sub-tree we add to our target configuration. These rules depend on the relationship between the parent and its children. We now formulate our rules for the And-, Or-, and Alternative-relation, which build the fundamentals for our algorithms [Minimal Estimation Algorithm](#) and [Maximal Estimation Algorithm](#):

- Alternative-relation

In this case, we choose to add exactly one of the children to the target configuration. If there is a child $f \in S$, we always add this one. Otherwise we include $f \notin U$ with the highest or lowest overall sub-tree value for the maximum and minimum, respectively.

- And-relation

First, all features f with $f \in S$ or $\omega(f) = 1$ are added. Afterward, we add all children whose overall sub-tree value is negative and which are not in U for the computation of our minimum. For the maximum, we include all children whose overall sub-tree value is positive and which are not in U .

- Or-relation

First, we add all features f with $f \in S$. Afterward, we add all children whose overall sub-tree value is negative and which are not in U for the computation of our minimum. For the maximum, we include all children whose overall sub-tree value is positive and which are not in U . However, if no child was included by these rules the child with the highest or lowest overall sub-tree value that is not in U has to be added to the product for the maximum and minimum, respectively.

Now, we analyze our the correctness of our heuristic.

Algorithm 1 Minimum Estimation Algorithm

```

1: procedure GETSUBTREEMINIMUM(FEATURE)
2:    $min = \pi_{att}(feature)$ 
3:   if not hasChildren then return  $min$ 
4:   if isAnd then
5:     for all children do
6:       if  $child \in U$  then
7:         continue
8:       if  $\omega(child) = 1$  or  $getSubtreeMinimum(child) < 0$  or  $child \in S$ 
9:         then
10:            $min+ = getSubtreeMinimum(child)$ 
11:   if isOr then
12:      $numberOfUnselected = 0$ 
13:      $unaddedValues = \emptyset$ 
14:     for all children do
15:       if  $child \in U$  then
16:         increment  $numberOfUnselected$ 
17:         continue
18:       if  $child \in S$  or  $getSubtreeMinimum(child) < 0$  then
19:          $min+ = getSubtreeMinimum(child)$ 
20:       else
21:          $unaddedValues = unaddedValues \cup getSubtreeMinimum(child)$ 
22:       if  $|unaddedValues| + numberOfUnselected = |children|$  then
23:          $min+ = \min\{value \mid value \in unaddedValues\}$ 
24:   if isAlternative then
25:      $values = \emptyset$ 
26:     for all children do
27:       if  $child \in U$  then
28:         continue
29:       if  $child \in S$  then return  $min+ = getSubtreeMinimum(child)$ 
30:       else
31:          $values = values \cup getSubtreeMinimum(child)$ 
32:        $min+ = \min\{value \mid value \in values\}$ 
33:   return  $min$ 
34: procedure INITIALISATION
35:    $min = getSubtreeMinimum(r_M)$ 

```

Correctness

In this paragraph, we analyze our computation using the presented estimation algorithms. During the analysis, we prove the correctness of the algorithm regarding our goal to calculate the minimum/maximum of a variable without cross-tree constraints.

Algorithm 2 Maximum Estimation Algorithm

```

1: procedure GETSUBTREEMAXIMUM(FEATURE)
2:    $max = \pi_{att}(feature)$ 
3:   if not hasChildren then return  $max$ 
4:   if isAnd then
5:     for all children do
6:       if  $child \in U$  then
7:         continue
8:       if  $\omega(child) = 1$  or  $getSubtreeMaximum(child) > 0$  or  $child \in S$ 
9:         then
10:            $max+ = getSubtreeMaximum(child)$ 
11:   if isOr then
12:      $numberOfUnselected = 0$ 
13:      $unaddedValues = \emptyset$ 
14:     for all children do
15:       if  $child \in U$  then
16:         increment  $numberOfUnselected$ 
17:         continue
18:       if  $child \in S$  or  $\pi_{att}(child) > 0$  then
19:          $max+ = getSubtreeMaximum(child)$ 
20:       else
21:          $unaddedValues = unaddedValues \cup getSubtreeMaximum(child)$ 
22:       if  $|unaddedValues| + numberOfUnselected = |children|$  then
23:          $max+ = \max\{value \mid value \in unaddedValues\}$ 
24:   if isAlternative then
25:      $values = \emptyset$ 
26:     for all children do
27:       if  $child \in U$  then
28:         continue
29:       if  $child \in S$  then return  $min+ = getSubtreeMaximum(child)$ 
30:       else
31:          $values = values \cup getSubtreeMaximum(child)$ 
32:        $max+ = \max\{value \mid value \in values\}$ 
33:   return  $max$ 
34: procedure INITIALISATION
35:    $max = getSubtreeMaximum(r_M)$ 

```

Definition 4.1. A full configuration $C_t = (M, S_t, U_t)$ extends the configuration $C = (M, S, U)$ when the following properties are given:

- $S \subseteq S_t$
- $U \subseteq U_t$

Prior to proving the correctness of the result of both algorithms, we formally define our feature model $M_{\Psi=\emptyset}$ that is not considering cross-tree constraints.

Definition 4.2. Given a feature model $M = (N, r, \omega, \lambda, \Pi, \Psi)$, we get $M_{\Psi=\emptyset} = (N, r, \omega, \lambda, \Pi, \emptyset)$ as our alternative feature model that only considers the tree structure.

Theorem 4.3. Given a valid configuration $C = ((M, A, \pi), S, U)$ with $M = (N, r, \omega, \lambda, \Pi, \Psi)$ as input, the full configuration $C_{result} = ((M_{\Psi=\emptyset}, A, \pi), S_{result}, U_{result})$, representing the result of the maximum estimation algorithm extends the configuration $C' = ((M_{\Psi=\emptyset}, A, \pi), S, U)$ with $M_{\Psi=\emptyset} = (N, r, \omega, \lambda, \Pi, \Psi_\emptyset = \emptyset)$ and is valid.

Proof. The Theorem 4.3 is equivalent to the fulfilment of the following properties:

1. Every feature f with $\omega(f) = 1$ and $(e \in S_{result}, f) \in \Pi$ is in S_{result}
2. For every feature $f \in N$ with $\lambda(f) = \langle a, b \rangle$ is the number of its included children between a and b
3. Every formula $F \in \Psi_\emptyset$ is satisfied
4. $S \subseteq S_{result}$
5. $U \subseteq U_{result}$

The properties 1.–3. are given by the definition of a valid configuration and 4./5. by the definition of C_{result} extends C' . Therefore, proving every one of these statements proves the theorem. For this proof, we use N_f to denote the set of children of feature f in $M_{\Psi=\emptyset}$. Additionally, (1) denotes the first line of the Maximum Estimation Algorithm.

1. For every feature $f \in S_{result}$ `getSubtreeMaximum(f)` is executed. Therefore, every mandatory child $c \in N_f$ is included by definition of the algorithm, as long as c with $\omega(c) = 1$ is not in U (8). However, this would lead to a contradiction with the assumption of C' being valid.
2. There are three cases for λ in feature models representing the And-, Alternative and Or-relation. Without loss of generality, we assume $f \in N$ is an arbitrary feature. The idea of the proof for this statement is to show that for each input feature f of `getSubtreeMaximum(f)` the number of children that are included lies within the boundaries of $\lambda(f)$.

$\lambda(f) = \langle 0, n \rangle$ (And-relation) with $n = |N_f|$ is fulfilled for every possible inclusion of children.

The Alternative-relation is represented by $\lambda(f) = \langle 1, 1 \rangle$. We show that `getSubtreeMaximum(f)` always includes exactly one child feature $c \in N_f$ for every f with $\lambda(f) = \langle 1, 1 \rangle$. There are two possible cases. If a child $c \in N_f$ is in S the algorithm immediately stops iterating over the children and includes c (28). Otherwise, the child with the maximal sub-tree value is added, as long as $N_f \not\subseteq U$ (31). However, $N_f \subseteq U$ contradicts the assumption of C' being valid. Following from this, in both cases exactly one child is included.

The Or-relation is represented by $\lambda(f) = \langle 1, n \rangle$ with $n = |N_f|$. We show that `getSubtreeMaximum(f)` always includes at least one child feature $c \in N_f$ for

every f with $\lambda(f) = \langle 1, n \rangle$, with $n = |N|$. We assume no child was added during the iteration over the children (13–20). This implies $|unaddedValues| + numberOfUnselected = |children|$. As C' is valid, there has to be at least one child not in U . Therefore, $unaddedValues$ can not be empty and at least one child feature is included (21,22).

3. As $\Psi_\emptyset = \emptyset$ this holds trivially.
4. For the And-(8) and Or-relation(17) every feature $f \in S$ is included. Regarding the alternative relation, it is necessary to show that only one of the children can be in S . This is implied by the validity of C' . Therefore, every feature $f \in S$ is included in C_t (28).
5. The algorithm skips every feature $f \in U$ for every relation type (6, 14, 26).

□

Theorem 4.4. *Given a valid configuration $C = ((M, A, \pi), S, U)$ with $M = (N, r, \omega, \lambda, \Pi, \Psi)$ as input, the full configuration $C_{result} = ((M_{\Psi=\emptyset}, A, \pi), S_{result}, U_{result})$, representing the result of the minimum estimation algorithm extends the configuration $C' = ((M_{\Psi=\emptyset}, A, \pi), S, U)$ with $M_{\Psi=\emptyset} = (N, r, \omega, \lambda, \Pi, \Psi_\emptyset = \emptyset)$ and is valid.*

The proof for the [Minimum Estimation Algorithm](#) is analogous. Now, we show that given a configuration $C = (M, S, U)$ and an attribute att as input the maximum estimation algorithm will return the maximal possible sum of $\pi_{att}(f)$ for $C' = (M_{\Psi=\emptyset}, S, U)$ and att .

Theorem 4.5. *Given a valid configuration $C = ((M, A, \pi), S, U)$ with $M = (N, r, \omega, \lambda, \Pi, \Psi)$ and a numerical attribute $att \in A$ as input, the full configuration $C_{result} = ((M_{\Psi=\emptyset}, A, \pi), S_{result}, U_{result})$, representing the result of the maximum estimation algorithm, has the following property:*

$$\sum_{f \in S_{result}} \pi_{att}(f) \geq \sum_{f' \in S_i} \pi_{att}(f')$$

for every valid $C_i = ((M_{\Psi=\emptyset}, A, \pi), S_i, U_i)$ that extends $C' = ((M_{\Psi=\emptyset}, A, \pi), S, U)$.

Proof. We prove this statement by a contradiction. We assume there is a full configuration $C_a = (M_{\Psi=\emptyset}, S_a, U_a)$ that extends C' with $\sum_{f' \in S_a} \pi_{att}(f') > \sum_{f \in S_{result}} \pi_{att}(f)$. Without loss of generality, $f \in N$ with $f \notin S_a \cap S_{result}$ and $f \in S_a \cup S_{result}$ is a feature whose inclusion (if $f \notin S_{result}$) or exclusion (if $f \in S_{result}$) increases our sum. Such a feature always exists, as we demanded $\sum_{f' \in S_a} \pi_{att}(f')$ to be strictly higher than our sum. Furthermore, we know $f \notin U$ and $f \notin S$, as $f \notin U$ contradicts $f \in S_a \cup S_{result}$ and $f \notin S$ contradicts $f \notin S_a \cap S_{result}$. We now show that every case of a case distinction over the relationship types (And, Or, Alternative) of the feature f 's parent leads to a contradiction. It is worth noting that the actual value of a sub-tree only depends on the sub-tree itself, because $\Psi_{M_{\Psi=\emptyset}} = \emptyset$. For our cases, we define the set $N_{siblings_f} \subset N$ as the children of our feature f 's parent and $N_{selectable} = N \setminus U$.

1. And-relation

Without loss of generality, we assume $\omega(f) = 0$. We have two cases $f \in S_a$ and $f \notin S_a$:

$$f \in S_a \Rightarrow \text{getSubtreeValue}(f) > 0 \Rightarrow f \in S_{\text{result}}$$

$$f \notin S_a \Rightarrow \text{getSubtreeValue}(f) < 0 \Rightarrow f \notin S_{\text{result}}$$

The first case contradicts $f \notin S_a \cap S_{\text{result}}$ and the second one $f \in S_a \cup S_{\text{result}}$.

2. Or-relation

We have two cases $f \in S_a$ and $f \notin S_a$:

$$f \in S_a \Rightarrow \text{getSubtreeValue}(f) > 0$$

$$\begin{aligned} \forall \neg \exists g \in N_{\text{neighbors}_f} \cap N_{\text{selectable}} : \text{getSubtreeValue}(g) > \text{getSubtreeValue}(f) \\ \Rightarrow f \in S_{\text{result}} \end{aligned}$$

$$f \notin S_a \Rightarrow \text{getSubtreeValue}(f) < 0$$

$$\begin{aligned} \forall \exists g \in N_{\text{neighbors}_f} \cap N_{\text{selectable}} : \text{getSubtreeValue}(g) > \text{getSubtreeValue}(f) \\ \Rightarrow f \notin S_{\text{result}} \end{aligned}$$

The first case contradicts $f \notin S_a \cap S_{\text{result}}$ and the second one $f \in S_a \cup S_{\text{result}}$.

3. Alternative-relation

Without loss of generality, we assume $g \in N_{\text{neighbors}_f}$ is the second feature with $g \notin S_a \cap S_{\text{result}}$ and $g \in S_a \cup S_{\text{result}}$. We have two cases $f \in S_a$ and $f \notin S_a$:

$$f \in S_a \Rightarrow \text{getSubtreeValue}(f) > \text{getSubtreeValue}(g) \Rightarrow f \in S_{\text{result}}$$

$$f \notin S_a \Rightarrow \text{getSubtreeValue}(g) > \text{getSubtreeValue}(f) \Rightarrow g \notin S_{\text{result}}$$

The first case contradicts $f \notin S_a \cap S_{\text{result}}$ and the second one $f \in S_a \cup S_{\text{result}}$.

As every case leads to a contradiction, C_a cannot exist. Therefore,

$\sum_{f \in S_{\text{result}}} \pi_{\text{att}}(f)$ is indeed our desired maximum. □

Theorem 4.6. *Given a valid configuration $C = ((M, A, \pi), S, U)$ with $M = (N, r, \omega, \lambda, \Pi, \Psi)$ and a numerical attribute $\text{att} \in A$ as input, the full configuration $C_{\text{result}} = ((M_{\Psi=\emptyset}, A, \pi), S_{\text{result}}, U_{\text{result}})$, representing the result of the minimum estimation algorithm, has the following property:*

$$\sum_{f \in S_{\text{result}}} \pi_{\text{att}}(f) \geq \sum_{f' \in S_i} \pi_{\text{att}}(f')$$

for every valid $C_i = ((M_{\Psi=\emptyset}, A, \pi), S_i, U_i)$ that extends $C' = ((M_{\Psi=\emptyset}, A, \pi), S, U)$.

The proof for the minimum is analogous to the proof for the maximum. As we have shown the correctness, we now discuss the relevance of our result.

The estimation results bound the actual minimum and maximum, because the sets $N_i \subseteq N$ that leave C satisfied are a subset of the sets $N'_i \subseteq N$ that leave C' satisfied. So the following always holds true: $\text{Min}_{\text{estimated}} \leq \text{Min}_{\text{exact}}$ and $\text{Max}_{\text{estimated}} \geq \text{Max}_{\text{exact}}$. Therefore, the approximation is conservative. Furthermore, the estimations have a linear runtime. We will further analyze the runtime in Chapter 7.

4.5 Summary

In this chapter, we presented a formal definition of the algorithmic problem of computing attribute ranges for partial configurations. Additionally, we described two methods for the computation of attribute ranges and showed their correctness.

We discussed the calculation of the minimum and maximum with the help of an SMT solver. The main task is mapping the problem into a first-order logic formula, which represents the *CNF* of a feature model, the selected and unselected features, the attribute values, and the sum of all attribute values. This sum is the numerical variable the SMT solver is supposed to optimize. However, the optimization of a variable using SMT is a complex task and we expected a long runtime for feature models with a large number of features. Therefore, we proposed an heuristic, that approximates the range.

The heuristic approximates attribute ranges by not considering cross-tree constraints of the feature model. We proposed two algorithms, one for computing the minimum and one for computing the maximum. These algorithms conservatively approximate the ranges in linear time.

5. Implementation into FeatureIDE

In this chapter, we describe how we implemented the proposed concepts from [Chapter 3](#) and [Chapter 4](#). We implement both concepts into the framework `FEATUREIDE` which is introduced in [Section 5.1](#). In [Section 5.2](#), we start with the implementation of a general data structure which represents the input for a solver. Next, we describe how we implemented the abstract solver type `IncrementalSolver` into `FEATUREIDE` and which solvers we add in [Section 5.3](#). [Section 5.5](#) describes the default statistics for configurations and the view to display them. Then, [Section 5.6](#) describes the additional statistics regarding feature attributes, also including the implementation of the attribute range computation.

In [Chapter 3](#), we discuss the comparison of SAT and SMT solvers on the task of the automated analysis of feature models. We proposed the concept for the `IncrementalSolver`, an abstract data type to compare both kinds of solvers on that task. Now, we are interested whether SMT solvers are superior to SAT solvers regarding efficiency. Therefore, we defined general analyses with the help of the abstract data type to benchmark SAT and SMT solvers. Additionally, we provided two methods capable of computing attribute ranges for partial configurations in [Chapter 4](#). We decided to implement these into an existing feature modeling tool. Using this implementation, we aim to compare our heuristic, consisting of the estimation algorithms for the minimum and maximum, to a computation of attribute ranges using an SMT solver. Furthermore, we aim to compare SAT and SMT solvers on the task of automated analysis of feature models.

5.1 Building on Existing Tools

In this section, we are going to introduce the framework `FEATUREIDE` and present the different solvers we implemented into it. `FEATUREIDE` is an integrated development environment based on `ECLIPSE` that assists the user in developing software product lines. It consists of multiple plug-ins and provides a variety of different tools like a graphical feature model editor, configuration editor, and the support of different

composers. FEATUREIDE is an open source project written in JAVA.¹ Thus, JAVA is the programming language that we used to implement our thesis.

We chose FEATUREIDE because it is one of the most used open-source solutions and it provides a certain level of abstraction, which gives us the ability to create our own feature model and configuration formats which can still be used with the editors provided by FEATUREIDE [BRN⁺13]. By implementing our feature attributes with the help of the interface provided by FEATUREIDE, we greatly benefit from the rich functionality of FEATUREIDE. Our implementation of feature attributes and their user interface is modeled as an additional plug-in. If a user wants the functionality of the attributes he only needs to install that plug-in. The source code for this thesis is available on Github.²

The feature model editor provided by FEATUREIDE can automatically analyze the feature model. FEATUREIDE uses the SAT solver SAT4J to automatically detect the defects explained in Section 3.3. By replacing the static implementation of the SAT4J solver with the abstract data type `IncrementalSolver` proposed in Section 3.2, FEATUREIDE gains the ability to easily replace solvers used for any analysis. For our thesis, we reimplemented SAT4J and implemented the SMT solver API JAVASMT for comparison.

SAT4J is an open-source library which can be accessed easily, is written entirely in JAVA, provides SAT solving technologies, and is reusable [Ber]. A native implementation for SAT4J was already present in FEATUREIDE. Therefore, we decided to adapt the implementation to the `IncrementalSolver` data type.

For the SMT solvers we decided to use the solver API JAVASMT because it provides multiple solvers at once and is entirely written in JAVA. JAVASMT is an open-source library which communicates with the native solvers directly [KFB16]. The different solvers and their supported theorems and functions are shown in Table 5.1. The biggest difference between both solvers is that SAT4J is a native SAT solver and has many optimization options. In contrast, JAVASMT handles the communication to five different native SMT solvers and does only provide restricted optimization options. The differences and similarities of JAVASMT and SAT4J are summarized in Table 5.2.

5.2 Data Structure for Formulas

Addressing multiple solvers with the help of an interface requires a standard data structure that represents formulas. FEATUREIDE uses PROP4J to represent a feature model in conjunctive normal form. PROP4J is a library that makes it easier to create propositional expressions. To calculate feature attribute ranges we decided to extend PROP4J by making it possible to create restricted first-order logic expressions.

PROP4J allows the creation of nodes which represent variables, called `Literal`, or boolean operations. The nodes can also transform themselves into conjunctive normal form which saves us the effort to write an algorithm for the transformation. The

¹<https://github.com/FeatureIDE/FeatureIDE>

²https://github.com/Subaro/BachelorThesis_Sprey_Sundermann

Solver:		Z3	OPTIMATHSAT	MATHSAT	SMTINTERPOL	PRINCESS
Theorems	Integer	✓	✓	✓	✓	✓
	Rational	✓	✓	✓	✓	-
	Array	✓	✓	✓	✓	✓
	Bitvector	✓	✓	✓	-	-
	Float	-	✓	✓	-	-
Functions	Unsat Core	✓	✓	✓	✓	-
	Partial Models	✓	-	-	-	✓
	Assumptions	✓	✓	✓	✓	✓
	Quantifiers	✓	-	-	-	✓
	Interpolation	✓	✓	✓	✓	✓
	Optimization	✓	✓	-	-	-
	Incremental Solving	✓	✓	✓	✓	✓
	SMT-LIB2	✓	✓	✓	✓	✓

Table 5.1: The native solvers supported by JAVASMT and their functionality, adapted from [KFB16]

class diagram of the relevant PROP4J nodes is shown in Figure 5.1. For the sake of clarity, other nodes were omitted. The different nodes that PROP4J provides and their functionality is shown in Table 5.3.

These nodes allows us to model any propositional expression needed to represent feature models and their constraints. However, the computation of attribute ranges requires more nodes to model restricted first-order expressions.

The class diagram for the PROP4J extension is shown in Figure 5.2. We extended the nodes to model atomic formulas. An atomic formula consists of two terms and evaluates to a boolean value based on a given operation. A term is a variable, a constant, or a function. As an example the formula $integerVariable \geq 334$ is an

Aspect	SAT4J	JAVASMT
Solves SAT	✓	✓
Solves SMT	-	✓
JAVA	✓	✓
Optimization options	✓	✓(restricted)
Native	✓	-
Multiple Solver	-	✓
Memory header	-	✓
Can explain defects	✓	✓

Table 5.2: Comparison of the functionality of SAT4J and JAVASMT

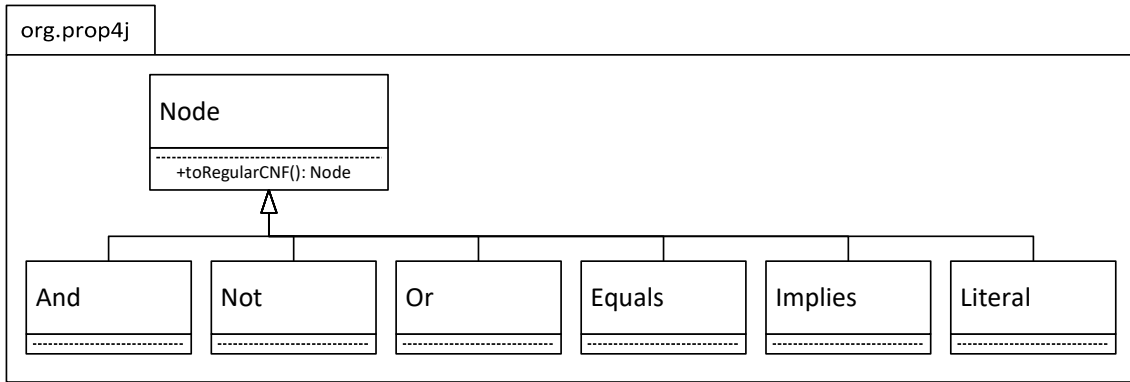


Figure 5.1: The original state of PROP4J

atomic formula that consist of a variable of the type `Integer` and 334 which is a constant of the type `Integer`. Both terms are connected by the relation \geq . A detailed explanation for the new nodes is given in Table 5.4.

Term represents the terms of atomic formulas and they evaluate to a numerical value instead of a boolean value. There are three types of terms: **Variable**, **Constant**, and **Function**. **Constant** is used to create constants for the formulas. The second type, **Variable**, is used to create variables for the formulas which are identified by a **String**. They represent numerical variables instead of boolean variables. The last type called **Function** represents functions over numerical values like addition, subtraction and more. To ensure type safety, we created the class **Datatype** to identify the current type for a specific variable or constant. As seen in Figure 5.2, we implemented the types **Long** and **Double**. With the new nodes it is now possible to model restricted first-order logic formulas which can be evaluated using an SMT solver. However, the nodes from PROP4J cannot be used as an input for our abstract data type **IncrementalSolver** because for the usage of explanations, we need a specific mapping from the clauses and variables into integers and the other way around which is not provided. Thus, we need a new data structure which contains the required mappings.

As proposed in Section 3.2, every solver receives an unmodifiable problem. That unmodifiable problem is designed to be as reusable as possible to prevent multiple creations of the same problem. The problem is designed to be a SAT or an SMT problem. The architecture is shown in Figure 5.3. The **ISolverProblem** is a JAVA interface providing methods that every problem either SMT or SAT need to support. All problems contain a **Node** which is the root node for the conjunctive normal form. For the explanations of defects, a specific mapping from clause and variable into integer is needed. Therefore, every problem provides the mapping.

ISmtProblem and **ISatProblem** are sub interfaces to the **ISolverProblem** used to identify the current problem either as a SAT or SMT problem and helps to prevent the creation of SAT solvers with SMT problems. The specific implementation for the SAT problem is **SatProblem** and for the SMT problem is **SmtProblem**. Every problem can be created by giving a **Node** as input. For **SatProblem**, the given node needs to be in conjunctive normal form. For **SmtProblem**, the **Node** needs to consist of conjunctions of clauses and **AtomicFormula**. After an **ISolverProblem** is created,

Node	Description
Node	Is an abstract class and provides functionality to transform the formula into conjunctive normal form and presets that every node can have multiple Nodes as children. The following implemented nodes exist
Literal	Represents a boolean variable which can be either <i>positive</i> or <i>negative</i> and is identified by an Object .
Not	Represents the logical negation \neg and evaluates to <i>true</i> when the child evaluates to <i>false</i> .
And	Represents the logical conjunction \wedge and evaluates to <i>true</i> when all the children evaluate to <i>true</i> .
Or	Represents the logical disjunction \vee and evaluates to <i>true</i> when at least one child evaluates to <i>true</i> .
Equals	Represent the logical equality $=$ and evaluates to <i>true</i> when all children evaluate to the same value.
Implies	Represents the logical implication \Rightarrow and evaluates to <i>true</i> if the left child evaluates <i>false</i> or the right child evaluates <i>true</i> .

Table 5.3: PROP4J nodes for propositional formulas and their functionality

it cannot be changed, making it possible to take the same problem for all analyses without making it inconsistent. However, most of them need to change the current formula. Therefore, we implement the `IncrementalSolver` which can **push** and **pop** additional clauses to the internal formula without making the problem inconsistent.

5.3 Implementing the Abstract Data Type `IncrementalSolver`

We implemented the abstract data type `IncrementalSolver` as a JAVA interface. The interface allows us to add or remove solvers, modify the internal formula, generate unsatisfiable cores, and optimize variables. The class diagram in Figure 5.4 shows the architecture of the solver interface with the extensions explained in Chapter 3. The interface `ISolver` provides us all necessary operations for the abstract data type `IncrementalSolver` defined in Section 3.2. Every solver also needs a mapping from clauses into integer and back to provide indexes even for clauses which were pushed to the solver and are not part of the `ISolverProblem`. It is also possible to set configurations for the specific solver by calling the provided `setConfiguration(...)` method. SMT and SAT solvers are realized into the new sub interfaces `ISatSolver` and `ISmtSolver`. These are used to prevent analyses using the wrong type of solver. As an example, the analyses for attribute ranges cannot be solved by `ISatSolver` permitting only `ISmtSolver` for the calculation of attribute ranges.

The extension of the solver interface to find explanations for feature model inconsistencies is modeled as an additional interface `IMusExtractor`. The interface provides the functionality to calculate the unsatisfiable core of a solver's internal formula.

Node	Description
AtomicFormula	Is the abstract class for the new nodes presetting that every atomic formula has two terms.
LessThan	Evaluates to <i>true</i> when the value of the left term is less than the value of the right term.
LessEqual	Evaluates to <i>true</i> when the value of the left term is less equal the value of the right term.
GreaterThannd	Evaluates to <i>true</i> when the value of the left term is greater than the value of the right term.
GreaterEqual	Evaluates to <i>true</i> when the value of the left term is greater equal the value of the right term.
Equal	Evaluates to <i>true</i> when the value of the left term is equal the value of the right term.

Table 5.4: PROP4J nodes for restricted first-order formulas and their functionality

Günther proposed a concept of finding explanations for feature model inconsistencies which he also implemented in FEATUREIDE [Gün17]. We made his implementation compatible with our solver interface for finding and evaluating explanations. **IOptimizaionSolver** realizes the second extension to solve optimization problems. Solvers, which implement this interface can evaluate the minimum and maximum value of a given numeric variable, making it possible to calculate the ranges for our attributes.

All the provided interfaces grant a certain level of abstraction, making it possible to separate the usability of the solver interface with different implementations for the solvers. Therefore, it is possible to add every solver to the interface. As part of our thesis, we implemented the solver SAT4J and the solver API JAVASMT. For JAVASMT, two implementations are needed. The first is handled as a pure SAT solver used only for solving SAT problems. The other one is the implementation for solving SMT.

It is quite simple to add new solvers. Solvers that are used to solve SAT problems need to implement the **ISatSolver** interface or to extend the **AbstractSatSolver**. In contrast, solvers only used to solve SMT problems need to implement the **ISmtSolver** interface or extend the **AbstractSmtSolver** class. If a solver is supposed to solve SAT and SMT, like JAVASMT, we need to create both implementations.

Implementation of Sat4j

The package `org.prop4j.impl.sat4j` contains the concrete implementations for SAT4J. The `Sat4jSatSolver` is a pure SAT solver and inherits `AbstractSatSolver`. By extending the `Sat4jSatSolver` with the `IMusExtractor` in the class `Sat4jSatMusExtractor`, SAT4J enables the retrieval of unsatisfiable cores.

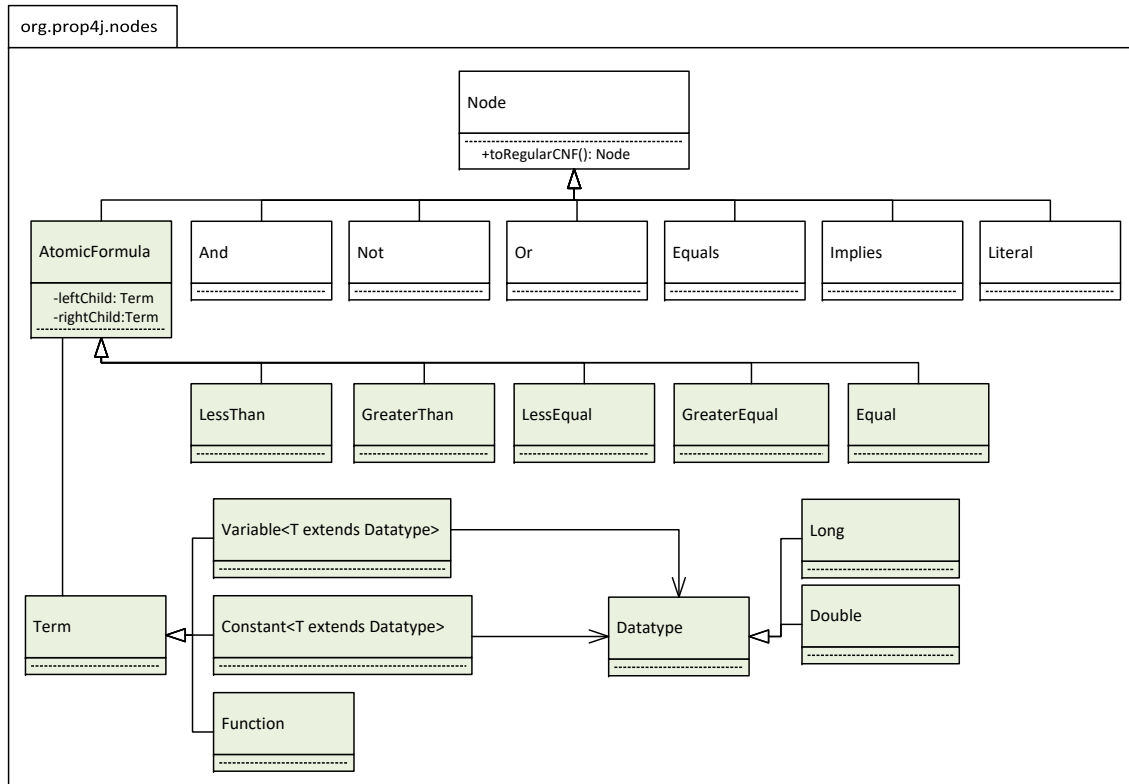


Figure 5.2: Extension of PROP4J that we implemented to build restricted first-order expressions

Implementation of JavaSMT

JAVASMT provides multiple solvers to use. You can easily choose between the different solvers. Changes to already instantiated `ISolver` instances are possible with the help of the `setConfiguration(...)` method. We realized the SAT solver implementation of JAVASMT in `JavaSmtSatSolver`. The implementation prevents the solver from getting an `ISmtProblem` as an input and therefore from facing SMT problems. By extending the `JavaSmtSatSolver` with the `IMusExtractor` in the class `JavaSmtSatMusSolver`, we provided the functionality to calculate unsatisfiable cores. Optimizing non-boolean variables is not possible with SAT solving. So we had to create the `JavaSmtSolver`, which extends the `AbstractSmtSolver`, making it possible to solve SMT problems. By also implementing the `IOptimizationSolver` interface we gave `JavaSmtSolver` the ability to optimize variables.

We decided to split the SAT and SMT solver implementation for JAVASMT. Therefore, we can evaluate the `JavaSmtSatSolver` on the automated analysis of feature models, while the `JavaSmtSolver` is used for the evaluation of computing attribute ranges for parital configurations. Both `Sat4jSatSolver` and `JavaSmtSatSolver` are later used for the feature model inconsistency analyses because they only depend on SAT. Their extensions `Sat4jSatMusSolver` and `JavaSmtSatMusSolver` are used for find explanations with the architecture given by Günther [Gün17]. The solver `JavaSmtSolver` can optimize variables and is therefore used to calculate the feature attribute ranges.

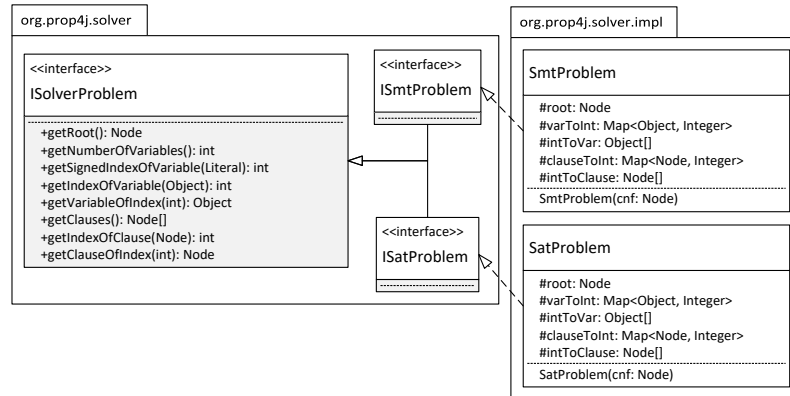


Figure 5.3: Class diagram for the SAT and SMT problems

Implementation of Analyses

Now, we present our implementation for the analyses and show how the analyses and solvers are created. For the creation of analyses and solvers, a factory pattern [GHJV95] is used which helps us to switch between multiple factories easily. Every analysis is created in a factory class. On creation, a new solver is instantiated providing the functionality to decide on a different solver for every analysis. We implemented the two factories `JavaSmtSolverAnalysisFactory` and `Sat4jSolverAnalysisFactory`. With the help of the factory pattern, the addition of a new analysis is straightforward, and the substitutability of the different factories makes it easy to add new solvers. Next, we explain the architecture used for the analyses in Figure 5.5.

Every analysis needs to implement the `ISolverAnalysis<T>` interface. Because of the interface, it is possible to hide the different implementations for the analyses. The generic type `T` determines the result or return value of the analysis. The next class `GeneralSolverAnalysis<T>` is a subclass of `LongRunningMethod<T>`, provided by FEATUREIDE, that allows us to run every analysis in a separate thread [KPK⁺17]. `GeneralSolverAnalysis` also implements `ISolverAnalysis` which makes it possible to write general analyses for `ISolver`. `AbstractSmtSolverAnalysis<T>` is the specification of `GeneralSolverAnalysis` to create analyses only for `ISmtSolver`. `AbstractSatSolverAnalysis<T>` is the specification of `GeneralSolverAnalysis` to create analyses only for `ISatSolver`.

General Analyses

Comparing JAVASMT and SAT4J requires analyses which can be performed by SAT and SMT solvers. Therefore almost every analysis proposed in the concept is realized as a subclass of `GeneralSolverAnalysis`. All general analyses are in the package `org.prop4j.analyses.impl.general`. The following analyses were implemented:

`SatisfiabilityAnalysis` checks whether the solver's internal formula is satisfiable. The analysis returns an array of indices representing the satisfying assignment. If the analysis returns null, then the model is unsatisfiable. The `SatisfiabilityAnalysis`, therefore, represent our feature model void analysis shown in Figure 3.3.

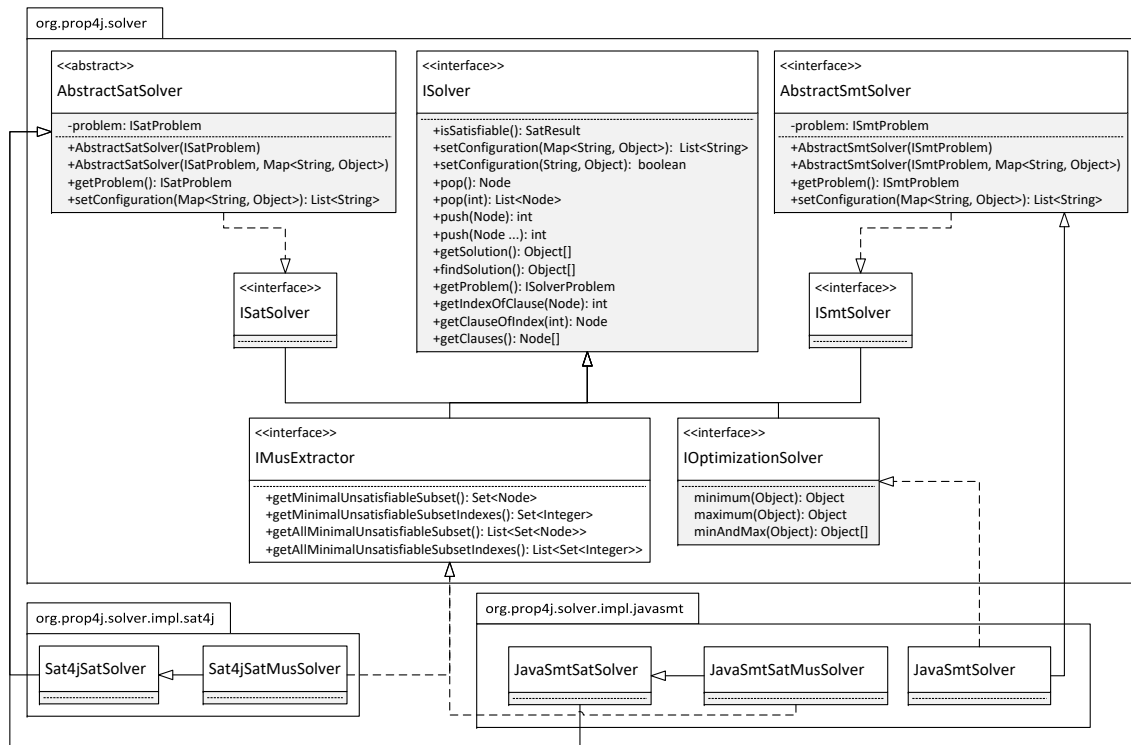


Figure 5.4: Class diagram for the solver interface

`CoreDeadAnalysis` computes every core and dead feature and returns them as list of indexes. Every feature is mapped into a unique index inside the `ISolverProblem`. If the list contains the positive index of a feature, then it is a core feature. In contrast, if the list contains the negative index of a feature, then it is a dead feature. The class `UnoptimizedCoreDeadAnalysis` is the representation of the unoptimized analysis shown in Figure 3.4. `CoreDeadAnalysis` does optimizations for the general analysis as shown in Figure 3.5. Even further optimizations for the SAT4J analysis in Figure 3.6 are made in the `Sat4jCoreDeadAnalysis`.

`UnoptimizedImplicationAnalysis` calculates the false-optional features without optimizations shown in Figure 3.7 and returns them as a list. Applying some optimizations leads to the optimized general analysis shown in Figure 3.8 which is implemented in the class `ImplicationAnalysis`. The false-optional features also have a SAT4J optimized analysis shown in Figure 3.9 which is implemented in the class `Sat4jImplicationAnalysis`.

`RedundantConstraintAnalysis` implements the analysis shown in Figure 3.10 and returns a list of constraints which are redundant.

`TautologicalConstraintAnalysis` implements the analysis shown in Figure 3.11 and returns a list of constraints which are tautologies.

The package `org.prop4j.analysises.impl.smt` contains the analyses used to calculate ranges of an attribute using an SMT solver. The analysis `FeatureAttributeMaximumAnalysis` return the maximum value of a variable. The analysis `FeatureAttributeMinimumAnalysis` return the minimum value of a variable.

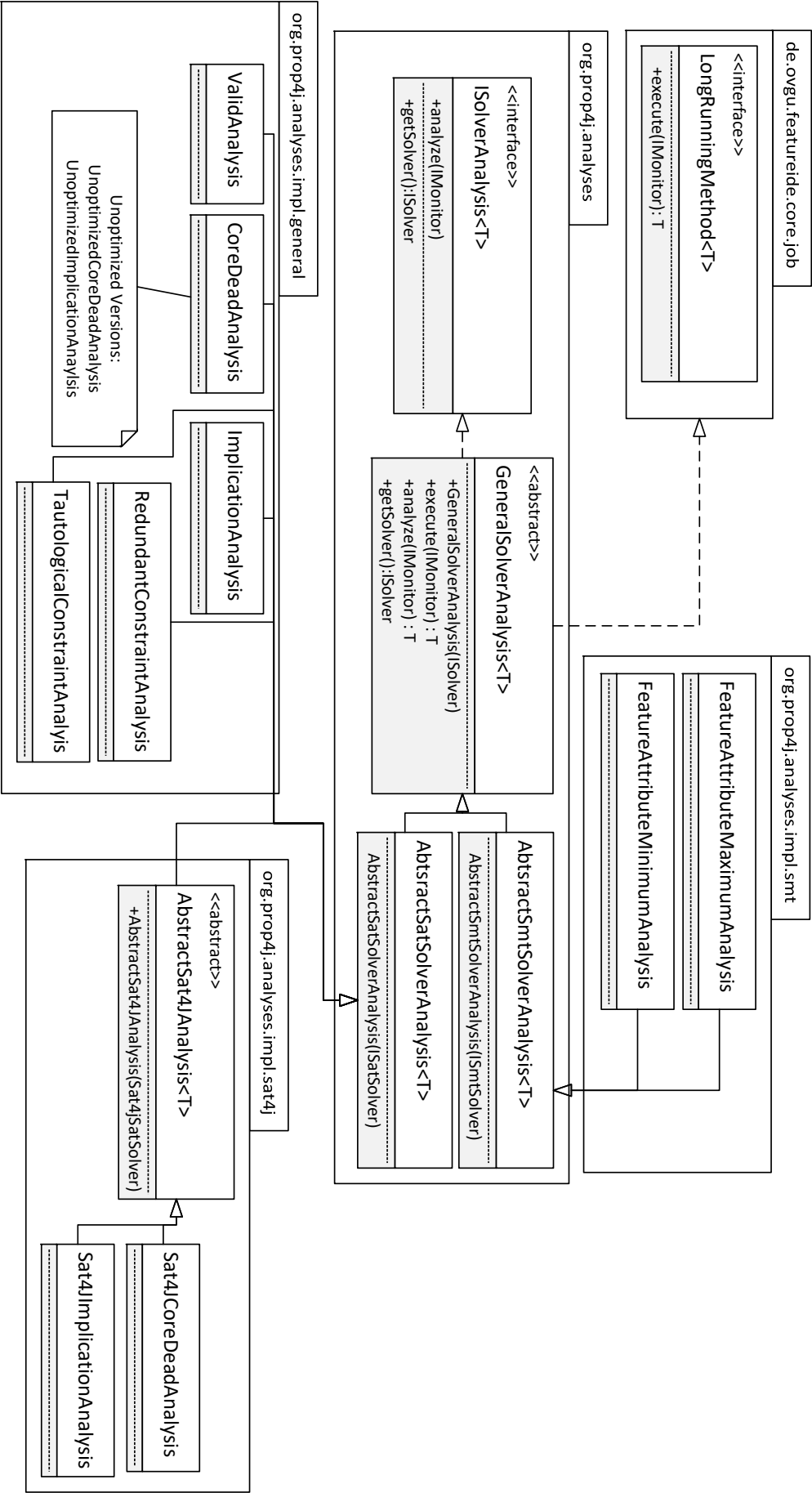


Figure 5.5: Class diagram for the analyses

5.4 Feature Attributes

This section describes the implementation of feature attributes in `FEATUREIDE`. First, we take a look at the implementation of our attribute structure. Afterward, we discuss the integration into the structure of `FEATUREIDE`. Then, we present our user interface.

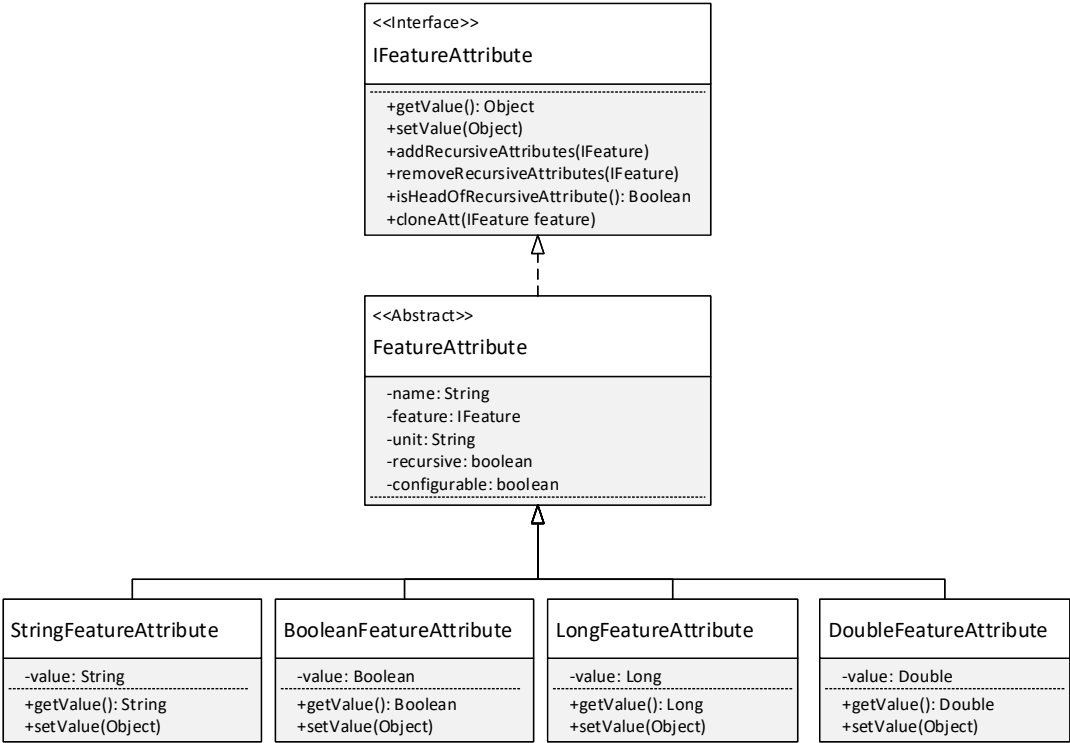


Figure 5.6: Attribute implementation into `FEATUREIDE`

Figure 5.6 shows how we implemented the attributes. Now, we want to define each of the shown classes more precisely.

`IFeatureAttribute` is the interface used for our attributes. It handles the communication with `FEATUREIDE` without requiring to specify a type. Furthermore, it defines mandatory functions for each attribute type to implement. These conventions regarding functions also allow the simple addition of new types.

`FeatureAttribute` is an abstract class inherited by each instance of an attribute. Every attribute is identified by its variable `name` and the `feature` holding it. Additionally, an attribute contains a `unit` and two indicators that show whether the attribute is `recursive` and/or `configurable`. If a `FeatureAttribute` is `recursive`, all of its feature’s descendants hold an instance of it. `configurable` indicates that the value can also be changed for each configuration.

`StringFeatureAttribute` is an instance of a `FeatureAttribute` whose value is a `String`. A use-case for this type might be attaching an URL to each feature of the model.

BooleanFeatureAttribute is an instance of a **FeatureAttribute** whose value is a **Boolean**. For example, this type can be used to indicate whether a component of a sandwich is organic.

LongFeatureAttribute is an instance of a **FeatureAttribute** whose value is a **Long**. Using this type a developer could add the amount of RAM needed for a specific software fragment to run.

DoubleFeatureAttribute is an instance of a **FeatureAttribute** whose value is a **Double**. An example for the usage of this type is the attachment of a price to each feature.

Now, we specify and explain the properties of a **FeatureAttribute** in more detail. Afterward, the reader will be able to understand the functionality of our attribute design.

The string **name** is the primary identifier of an attribute. While the attribute name has to be unique within the attribute list of a single feature, different features can hold attributes with the same name.

Every specific attribute belongs to exactly one feature. This feature is referenced by the **IFeature feature**. A pair of the variables **name** and **feature** uniquely identifies one attribute instance.

An attribute's **value** is an extension of **Object** that differs for each attribute type. **value** does not have to be specified, meaning it is either an instance of the **Object**-extension according to the attribute type or **null**.

The **unit** given as a **String** can be used to further describe an attribute. For example, you could specify that a price is given in €. This variable does not have to be specified. In this case it is an empty **String**.

If an attribute is set to **recursive**, a copy with the **value** set to **null**, is added to every descendant of **feature**. The recursively implemented function **addRecursiveAttributes(IFeature)** realizes this by adding the copy to every child of its input. We call the attribute that was originally set to recursive the head of this recursive attribute. If we change a property of our head attribute other than **value**, the other recursive attributes will be updated in the same way. Note that these properties can only be edited at the head of our attribute. Changing **recursive** of our head to **false** or deleting the attribute leads to the removal of all descendant attributes with the same **name**. This functionality is realized by **removeRecursiveAttributes(IFeature)**, which works similar to **addRecursiveAttributes(IFeature)**, but instead of adding a copy it, removes the recursive attribute.

The **value** of an attribute that is **configurable**, can not only be adjusted in the feature model but also for each configuration. In this case, the user can decide on a default value in the feature model editor. However, the value can still be adjusted in the configuration editor. Following from this, there can be multiple values for one attribute in different configurations. We now specify the integration of our feature attributes into FEATUREIDE.

Integration into FeatureIDE

FEATUREIDE provides multiple interfaces to create a custom feature model and configuration format. The architecture for the extended feature model is shown in Figure 5.7. FEATUREIDE provides the interfaces `IFeature` and `IFeatureModel`. Both interfaces are used to encapsulate the specific implementations from the usage within the framework.

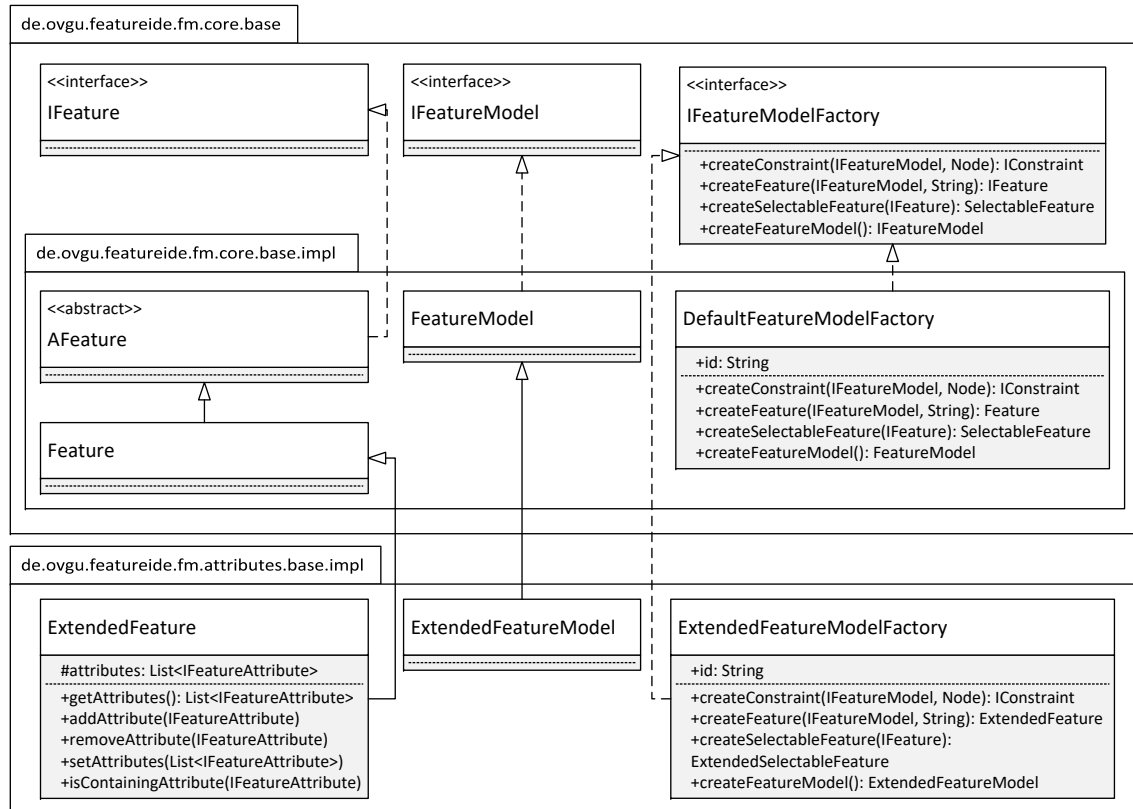


Figure 5.7: Architecture of the extended feature model implemented into FEATUREIDE

`AFeature` is the abstract class for all features and implements the interface `IFeature`. It is used by FEATUREIDE features to save properties. `Feature` is the concrete implementation of a regular feature. A `Feature` cannot save the attributes we described earlier. Hence, we created the subclass `ExtendedFeature` that manages the attributes.

`FeatureModel` represents a feature model and implements the interface `IFeatureModel`. It manages the features, constraints, and the structure of the model. We implemented `ExtendedFeatureModel` as subclass of the `FeatureModel`. In FEATUREIDE, the feature model, features, constraints, and other elements for the feature model editor are created through a factory. The interface `IFeatureModelFactory` specifies a method for each element that handles their creation. We implemented the `ExtendedFeatureModelFactory` that implements `IFeatureModelFactory`. It creates an `ExtendedFeature` instead of a `Feature` and an `ExtendedFeatureModel` instead of a `FeatureModel`. FEATUREIDE uses the class `XmlFeatureModelFormat` to read feature models from XML files and to write feature models into XML files.

If FEATUREIDE reads a feature model from a specific format, it uses the `IFeatureModelFactory` assigned to the format to create the specific elements. By creating an own format for extended feature models it is possible to assign the `ExtendedFeatureModelFactory` to the format. The architecture for the extended feature model format is shown in Figure 5.8.

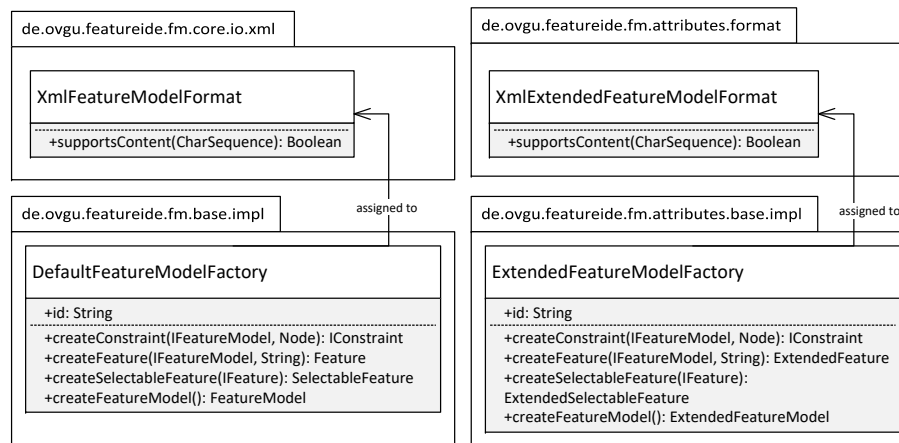


Figure 5.8: Architecture of the extended feature model format implemented into FEATUREIDE

We adapted `XmlFeatureModelFormat` in the class `XmlExtendedFeatureModelFormat` and extended it to read and write extended feature models which include the reading and persistent saving of the attributes and all their properties. Examples for the XML format used by FEATUREIDE and for our extended XML format are shown in Listing A.1 and Listing A.2. We assigned the `ExtendedFeatureModelFactory` to the `XmlExtendedFeatureModelFormat`. Next, we registered the `ExtendedFeatureModelFactory` through the extension point `de.ovgu.featureide.fm.core.FMFactory` provided by FEATUREIDE. Additionally, we registered the `XmlExtendedFeatureModelFormat` through the extension point `de.ovgu.featureide.fm.core.FMFormat` provided by FEATUREIDE. If we open a file through the feature model editor, the content is given to every format that is currently registered. Each format decides whether the content is acceptable by validating it through the method `supportsContent(CharSequence)`. If a format accepts the content, the feature model editor generates the elements of the feature model through the assigned `IFeatureModelFactory`.

Figure 5.9 shows the realization of configurable feature attributes in FEATUREIDE. We implemented the functionality required for these within the classes `ExtendedSelectableFeature` and `XmlExtendedConfFormat`.

`SelectableFeature` represents features for configurations in FEATUREIDE. Each instance of a `SelectableFeature` is identified by an `IFeature`. Additionally, it contains three instances of `Selection`, which indicate the status of the `SelectableFeature` in the configuration.

`ExtendedSelectableFeature` contains an additional `Map<String,String>` of the attributes whose value in the corresponding configuration is different to the default

value set in the feature model. This map can be obtained with `getConfigurableAttributes()`. Whenever the value of an attribute is changed in the configuration editor, `addConfigurableAttribute(String, String)` adds an entry, containing the name and value of the attribute, to the feature.

`XMLConfFormat` enables the persistent storage of a configuration. The relevant data can be obtained from a XML file with `readDocument(Document, List<Problem>)` and saved in a XML file with `writeDocument(Document)`. However, not every `IConfigurationFormat` is supported by this format. Therefore, `supportsContent(CharSequence)` only returns `true` if the `CharSequence` contains the root element `<configuration>`.

`XMLExtendedConfFormat` is responsible for handling XML files representing configurations with attributes. This functionality is realized by `readDocument(Document, List<Problem>)` and `writeDocument(Document)`. These functions extend the corresponding functions of `XMLConfFormat` with support of XML content representing attributes.

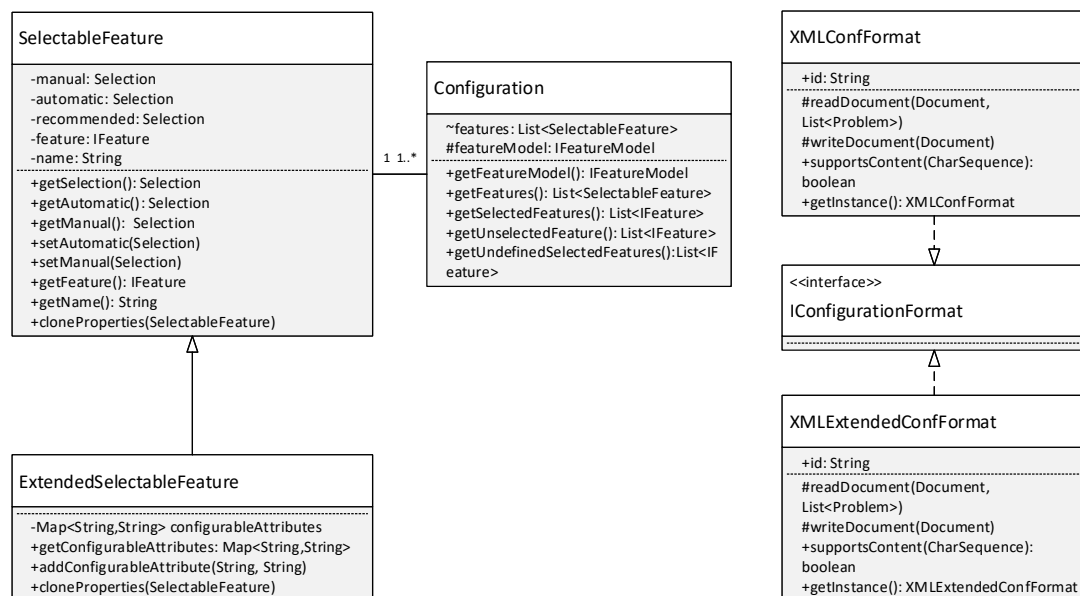


Figure 5.9: Feature attribute view while filtering the feature *Toast*

User Interface

Now, we are going to explain how we implemented the graphical user interface for the attributes. The interface helps the user to manage attributes which include the addition, deletion, and edit of attributes for extended feature models. The final user interface is shown in Figure 5.10. It shows the modified attributes for the feature of the feature model in Figure 4.1. The interface also provides a filter to display only features of interest and their attributes. An example of the filter is shown in Figure 5.11, where the feature *Toast* is currently filtered. The green entries are the selected features and their attributes. The ancestors of the selected features are







Icon	Description
	Identifies the entry as a feature.
	Identifies the entry as an attribute.
	Identifies the entry as an recursive attribute.
	Collapse all features except the root feature.
	Expands all features.
	Activates/deactivates the filter to show only features which are currently selected in the <code>FeatureDiagramEditor</code>

Table 5.5: Description for the different icons used in the feature attribute view

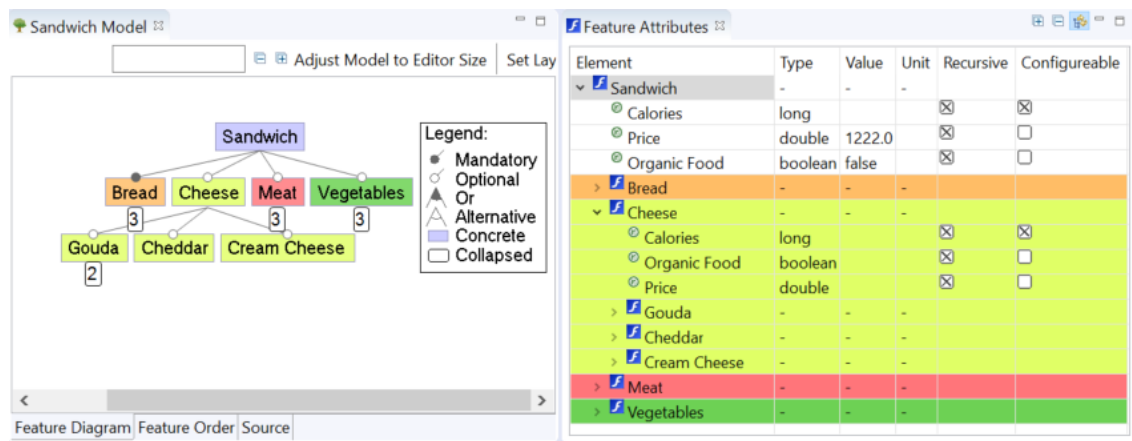


Figure 5.10: *Sandwich* feature model in the feature diagram editor provided by FEATUREIDE on the left and in the feature attribute view on the right.

colored gray. We explain the different icons that appear in the user interface in Table 5.5.

We implemented a view to display the attributes of the currently selected feature model. A view is a graphical component of the ECLIPSE framework. The creation and extension of views are simple while still benefiting from the rich functionality provided by ECLIPSE. The architecture for the attribute view is depicted in Figure 5.13.

`ViewPart` is an abstract class provided by ECLIPSE that needs to be extended by every view. It manages the complete life-cycle for us and provides us with many helpful extensions. We realize the view as the class `FeatureAttributeView` which is a subclass of `ViewPart`. Now, we will briefly list the requirements for the view:

- *R0*: The view works only with extended feature models or extended configurations.
- *R1*: The feature and their attributes are shown as a tree.
- *R2*: The view contains six columns for the name, the type, the value, the unit, recursive, and configurable.
- *R3*: It should be possible to add and delete attributes with the help of a context menu.

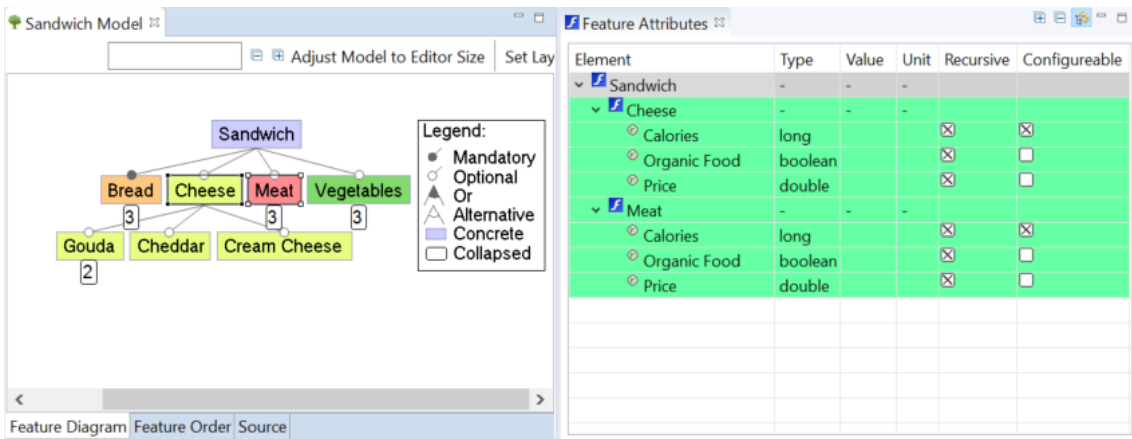


Figure 5.11: *Sandwich* feature model in the feature diagram editor provided by FEATUREIDE where the features *Cheese* and *Meat* are currently selected. On the right side is the attribute view with an activated filter for the features selected in the feature diagram editor.

- *R4*: Columns for name, value, unit, recursive, and configurable can be edited.
- *R5*: Editing or adding of invalid attributes should be prevented.
- *R6*: Features can be collapsed to clear the view.
- *R7*: Features can be filtered to display only features of interest and their attributes.
- *R8*: If configuration is open, show only configurable attributes and disable edit support except for the value column.
- *R9*: The features and their attributes are colored the same as in the feature model editor or in the configuration editor.

To further describe our view we need to introduce the `FeatureModelEditor` and `ConfigurationEditor` which are provided by FEATUREIDE. The `FeatureModelEditor` contains the `FeatureDiagramEditor` which allows the user to edit the feature model using a graphical interface. Our view can detect when an instance of the `FeatureDiagramEditor` is opened or in focus and retrieves the currently linked feature model. If the feature model is an instance of an `ExtendedFeatureModel`, then our view will display the attributes. Furthermore, if the feature model is not an extended feature model, then the user will be notified that only extended feature models support attributes. This behavior fulfills the requirement *R0*. If a `ConfigurationEditor` from FEATUREIDE is opened, a different behavior applies which will be explained later in this section.

The features and their attributes are displayed as a tree. The class `TreeViewer` is provided by ECLIPSE and allows us to display a tree with multiple columns inside our view. The elements that should be displayed are defined inside the `FeatureAttributeContentProvider`. Inside the content provider, we can determine which elements are shown and how many children an element has. The input for the content

Element	Type	Value	Unit	Recursive	Configurable
Sandwich	-	-	-		
Calories	long			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Bread	-	-	-		
Calories	long	75		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Full Grain	-	-	-		
Calories	long	203		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Cheese	-	-	-		
Meat	-	-	-		
Calories	long	20		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Salami	-	-	-		
Calories	long	116		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Figure 5.12: Configuration for the *Sandwich* in the configuration editor provided by FEATUREIDE on the left side. On the right side is the attribute view that displays only configurable attributes of features that are currently selected in the configuration.

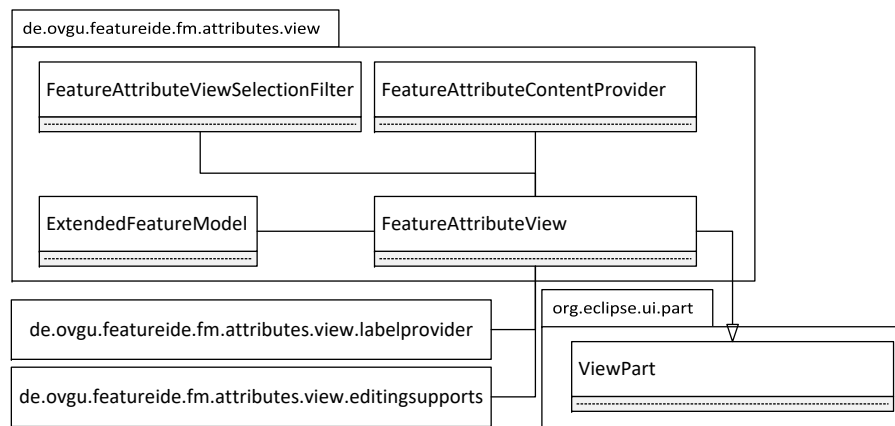


Figure 5.13: Class diagram for the architecture of the feature attribute view

provider is the currently selected feature model. We display all features and their attributes and preserve the tree structure used by the **FeatureDiagramEditor**. We implemented the **FeatureAttributeLabelProvider** that manages the information which are displayed in the view. In the label provider, we defined for every feature and his attributes that they take the same background color as the feature in the feature model editor. Additionally, we can expand and collapse features of the tree structure using the **TreeViewer** which provides that functionality natively. After collapsing a feature, its attributes and child features are omitted. Through the **TreeViewer** it is also possible to define multiple columns for our view. We add the six columns mentioned in the requirements to the view. All the described functionality fulfill the requirements *R1*, *R2*, *R6*, and *R9*.

Our view can display the feature model structure and the attributes for every feature. Now, we need operations to create and delete attributes. The actions can be added through the **IMenuManager** which is supported by the **TreeViewer** and allows us to create a context menu easily. By right-clicking a feature, a context menu appears containing actions to add an attribute of the type **String**, **Boolean**, **Long**, or **Double**.

Performing the operation creates an attribute with a unique name and the desired type and assigns it to the selected feature. The attributes can be removed in the same way. We just need to right-click an attribute, and a context menu containing a remove operation appears. The deletion of recursive attributes is only possible when the recursive head feature is removed, otherwise the delete action is not available. With the help of the context menu, the requirement *R3* is fulfilled.

The created attributes do not contain any values yet. Therefore, we needed to provide a way to edit the features. It is possible to edit the attributes through the editing support of each column. Editing support is a functionality provided by ECLIPSE to edit the entry of the currently selected cell. By creating a custom editing support for each column, we can verify the provided values. With that verification, it is possible to prevent inconsistent attributes, like two attributes with the same name assigned to the same feature, or assignments of invalid values to an attribute. Additionally, we verify that attributes of the same name, assigned to different features, have the same type. The editing support for recursive needed extended verification because it also adds attributes to the child features. Therefore, it is only permitted to create a recursive attribute if the descendants of the recursive attribute do not contain attributes of the same name. With the editing support and the verification of the attributes, the requirements *R4* and *R5* are fulfilled. The editing support for all columns are inside the package `de.ovgu.featureide.fm.attributes.view.editingsupports`.

Next, we implemented a toggleable filter to show only the features and attributes of the features which are currently selected in the `FeatureDiagramEditor`. This allows us to overview the attributes even for large feature models, and makes it possible to show only the features of interest. The filter `FeatureAttributeViewSelectionFilter` extends the class `ViewerFilter` which is provided by ECLIPSE and enables us to efficiently filter the tree by providing only the features of interest. The features of interest can be obtained easily through a selection listener that is registered to the `FeatureDiagramEditor`. Thus, the requirement *R7* is fulfilled.

Furthermore, we implemented a special interaction between the view and the `ConfigurationEditor` from `FEATUREIDE`. Now, only attributes that fulfill the following two requirements will be shown: First, the feature of an attribute needs to be selected in the `ConfigurationEditor`. Second, the attribute needs to be marked as configurable. It is only permitted to change the value of the configurable attributes. The editing support for the other columns is disabled. The values for the configurable attributes are not saved inside the feature model but they are saved inside the specific configuration instead. Hence, requirement *R8* is fulfilled.

5.5 Statistics for Configurations

This section describes the implementation of statistics regarding the configuration in FEATUREIDE. These statistics consist of a few simple properties like the amount of selected features and the attribute range computations. First, we describe the view displaying our statistics. Afterward, we specify the different entries in more detail. However, the main focus is on the range computations.

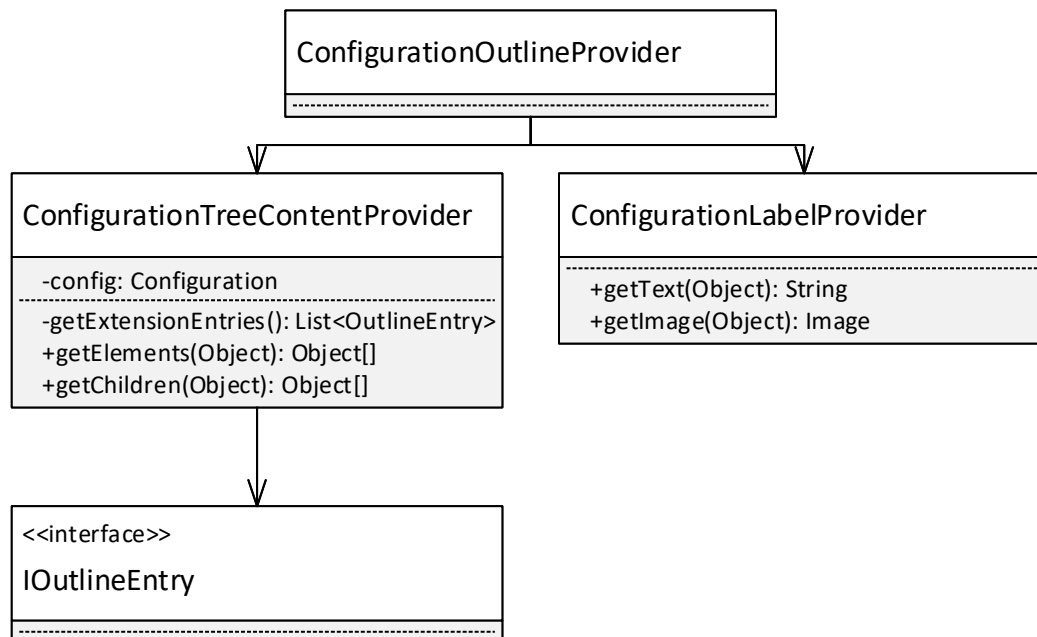


Figure 5.14: Class diagram for the configuration outline

Figure 5.14 gives an overview of our views architecture. The view is implemented as an outline, which shows specific information of an editor. Outline and editor are viewable at the same time, enabling a better workflow. The class **ConfigurationOutlineProvider** extends **OutlineProvider** from FEATUREIDE and mainly consists of one **ConfigurationTreeContentProvider** and one **ConfigurationLabelProvider**. The content provider extends FEATUREIDE's **OutlineTreeContentProvider** and is responsible for handling the content of our outline view. Each entry of the outline view is an instance of an **IOutlineEntry**. The label provider is responsible for displaying the statistics. It provides the displayed labels and images of the outline. Now, we further describe the details of our providers.

The **ConfigurationTreeContentProvider** defines which entries are to be added. Starting with the top-level elements which are defined by `getElements()`. This function returns an array `Object[]` containing all top level entries. These consist of the **ConfigurationOutlineStandardBundle**, which contains five properties of the configuration. Additional instances of **IOutlineEntry** can be added via an ECLIPSE EXTENSION POINT, which is provided by ECLIPSE and can be used to attach extensions. `getExtensionEntries()` is responsible for collecting all added extensions. Additionally, children can be added to each entry by `getChildren(Object)` which

usually takes an `IOutlineEntry` as input (another possible entry is a `String`, which is then used as the label of this entry). The definition of their children is up to each entry instance.

The `ConfigurationLabelProvider` determines the string and image displayed for each entry of the outline. `getText(Object)` returns the label for a given `IOutlineEntry` and `getImage(Object)` an image. Both the label and the image are defined by the `IOutlineEntry` itself. It is worth noting, that the image is optional as `null` is a legal return value for the method. In this case, only the label of the given entry is displayed. As our next step, we specify our entry interface.

Configuration Outline Entries

In this paragraph, we define `IOutlineEntry` more precisely. The interface is used to formulate conventions for entries and allows simple extensions of the outline. The possibility of extending is necessary, as `FEATUREIDE` is not always delivered with attributes. Therefore, the statistics regarding attributes are added via an extension point. Additionally, other statistic extensions can be added easily. First, we describe the architecture of the interface. Then, we present different instances of the interface.

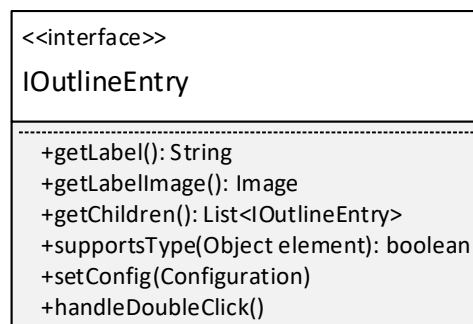


Figure 5.15: Class diagram for the outline entry interface

Figure 5.15 shows the class diagram of our interface. `getLabel()` and `getLabelImage()` define the displayed information and are used by the `ConfigurationLabelProvider`. The function `getChildren()` can be used to attach additional entries to the `IOutlineEntry` and return them. An instance of `IOutlineEntry` might not be suitable for every input element. In this case, the addition of the entry can be prevented with the help of `supportsType(Object)`, which indicates whether a given type is supported. Additionally, the configuration, referenced by each entry, can be defined with `setConfig(Configuration)`. `handleDoubleClick` is called whenever an entry is double clicked. The resulting behavior can be defined for each entry.

Figure 5.16 shows an example of the configuration outline displaying our default entries for two different configurations of our sandwich feature model. For every configuration, the view displays the number of selected, manually selected, unselected, manually unselected, and undecided features. Now, we describe the instances of `IOutlineEntry` which are added to the outline by default and provide the shown statistics.

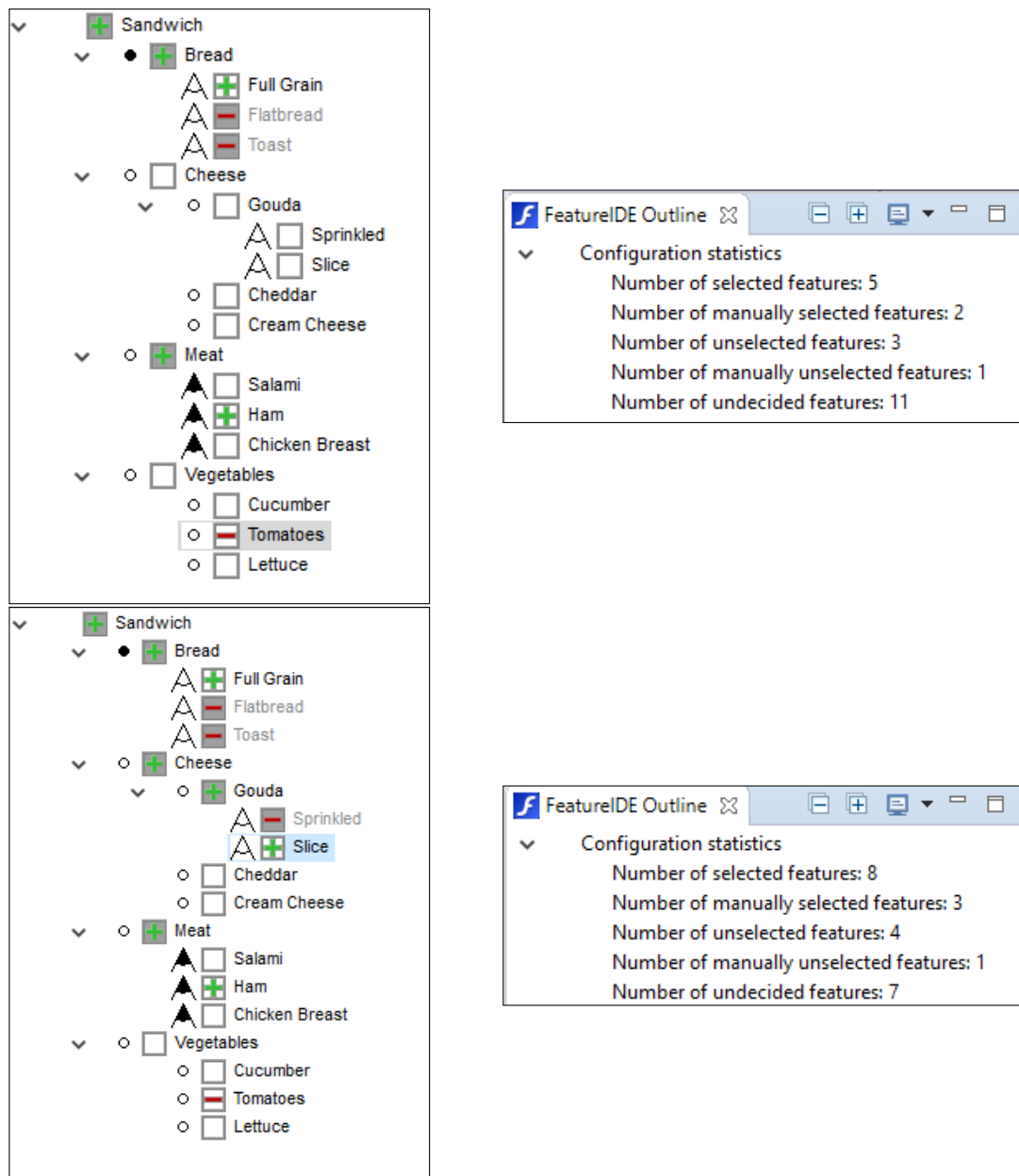


Figure 5.16: Example of the attribute entry statistics for the displayed configuration of a sandwich

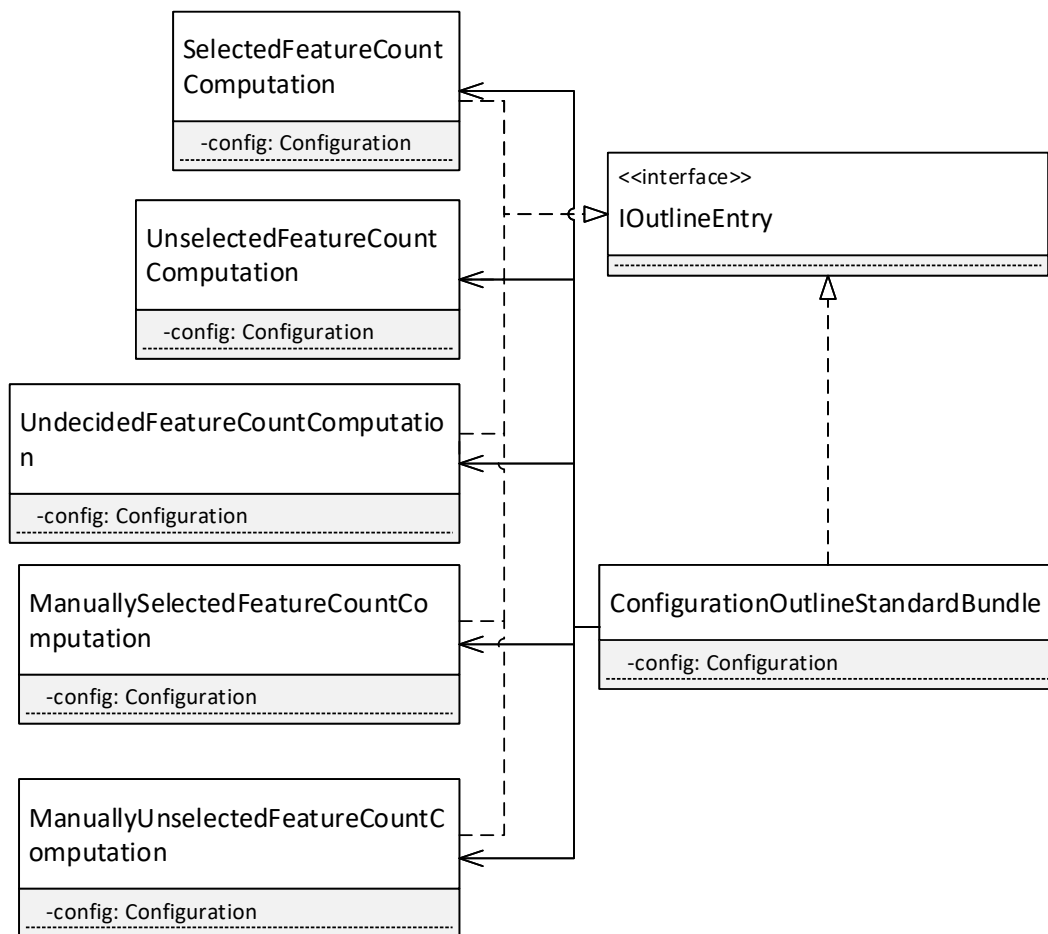


Figure 5.17: Class diagram for the default outline entries

- `ConfigurationOutlineStandardBundle` is our only default top-level entry. It functions as the header of our configuration properties. Its `getChildren()`-function returns the other instances of `IOOutlineEntry` given in Figure 5.17.
- `SelectedFeatureCountComputation` counts the number of currently selected features in the given `config`.
- `UnselectedFeatureCountComputation` counts the number of currently unselected features.
- `UndecidedFeatureCountComputation` computes the number of features that are neither selected nor unselected.
- `ManuallySelectedFeatureCountComputation` calculates the number of features that were manually selected in the configuration editor.
- `ManuallyUnselectedFeatureCountComputation` calculates the number of features that were manually unselected.

As the next step, we specify the additional extension point containing our attribute computations.

5.6 Attribute Entry Extension

This section describes the configuration outline entry displaying computations regarding the attributes of an `ExtendedFeatureModel`. This entry is supposed to be added to the outline for every attribute in the model. Figure 5.18 shows the architecture of this extension. Now, we briefly define the contained classes.

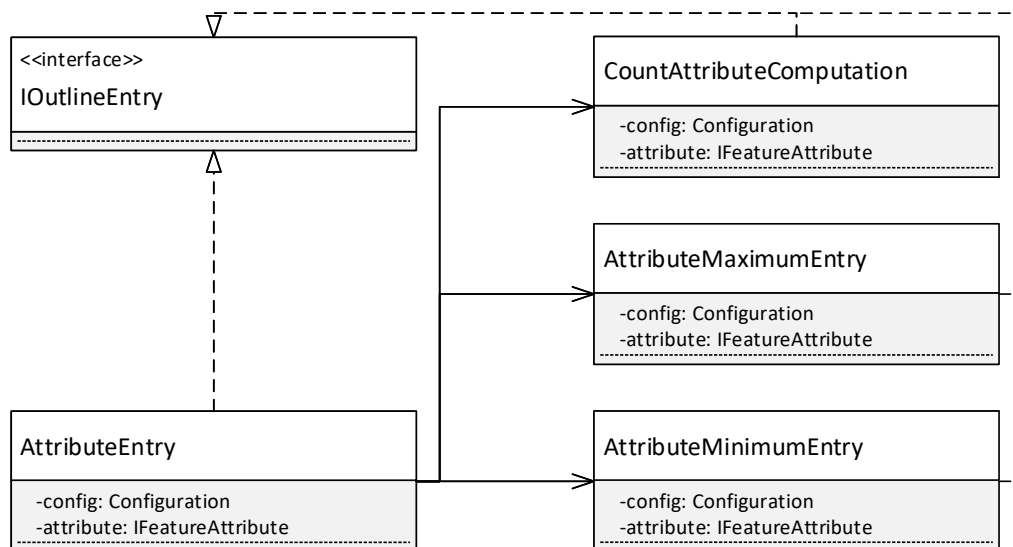


Figure 5.18: Architecture of the attribute outline entry

`AttributeEntry` is our top-level entry for attributes. Such an entry is created for each attribute in the feature model and functions as the header for our computations. Its label references the attribute's name. The function `supportsType(Object)` returns `true`, when the underlying feature model, which can be obtained through `conf`, is extended.

`CountAttributeComputation` counts the number of occurrences of the corresponding attribute. This entry is supported by every attribute type.

`AttributeMaximumEntry` and `AttributeMinimumEntry` display the possible maximum and minimum for the overall sum of the corresponding attribute respectively. On initialization, the approximated value is shown. However, the user can decide to compute the exact value with SMT and display it by double clicking the entry. Using this implementation, the user is always able to see approximated values and can obtain exact values on demand. This replacement is realized by `handleDoubleClick()`. Both entries are supported by `LongFeatureAttribute` and `DoubleFeatureAttribute`.

Figure 5.19 shows an example of our attribute entry. It represents our sandwich example while also including customer's wishes. The attribute ranges are still in their default state. Therefore, the displayed values are approximated.

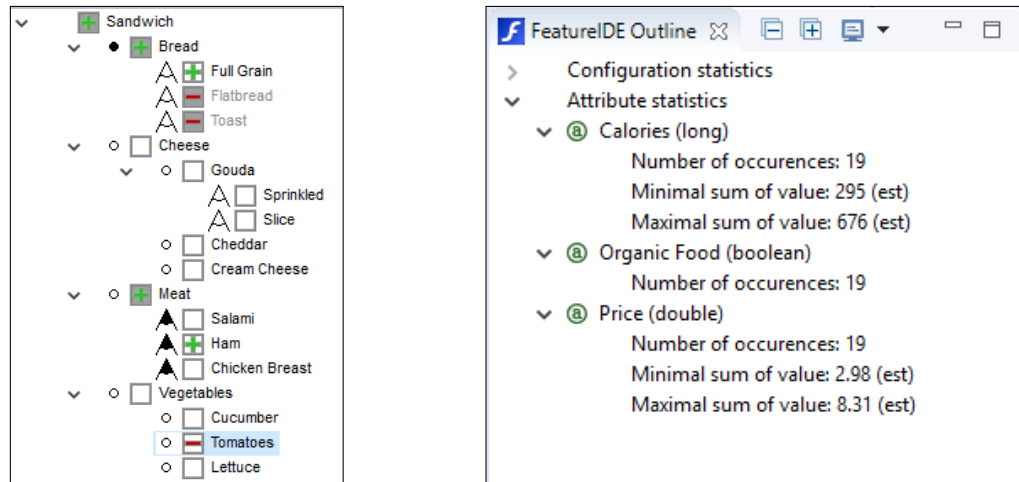


Figure 5.19: Example of attribute entry statistics for the displayed configuration of a sandwich

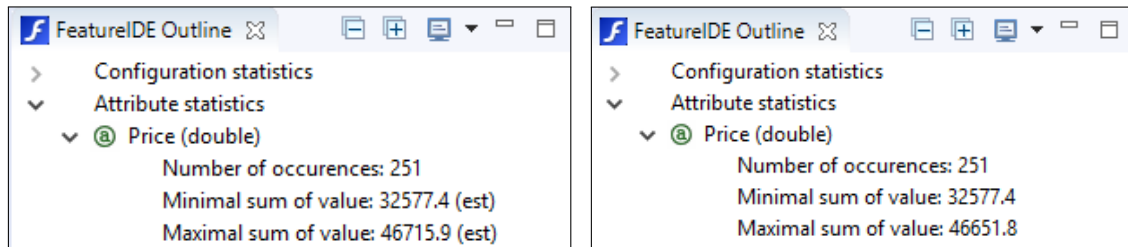


Figure 5.20: Example of the range computation with approximated on the left and exact values on the right side

Figure 5.20 shows the process of computing exact values with SMT for a double attribute. First, the approximated values are displayed. After right-clicking both entries, the exact range is shown. In the next step, the classes enabling computation of attribute ranges are described in more detail, starting with `SmtAttributeRangesComputation`.

Figure 5.21 shows the implementation of the outline entry that displays the maximal attribute value. Each `AttributeMaximumEntry` contains one `SmtMaximumComputation`, which computes the maximum with SMT, and one `EstimatedMaximumComputation`, which approximates the maximum using the estimation algorithm. On initialization, `getLabel()` returns the result computed by `getEstimatedMaximum()`. However, double clicking the entry triggers `handleDoubleClick()`, which sets the returned label to the return value of `getExactMaximum()`.

Figure 5.22 shows the implementation of the outline entry that displays the minimal attribute value. Each `AttributeMinimumEntry` contains one `SmtMinimumComputation`, which computes the minimum with SMT, and `EstimatedMinimumComputation`, which approximates the minimum using the estimation algorithm. On initialization, `getLabel()` returns the result computed by `getEstimatedMinimum()`. However, double clicking the entry triggers `handleDoubleClick()`, which sets the returned label to the return value of `getExactMinimum()`.

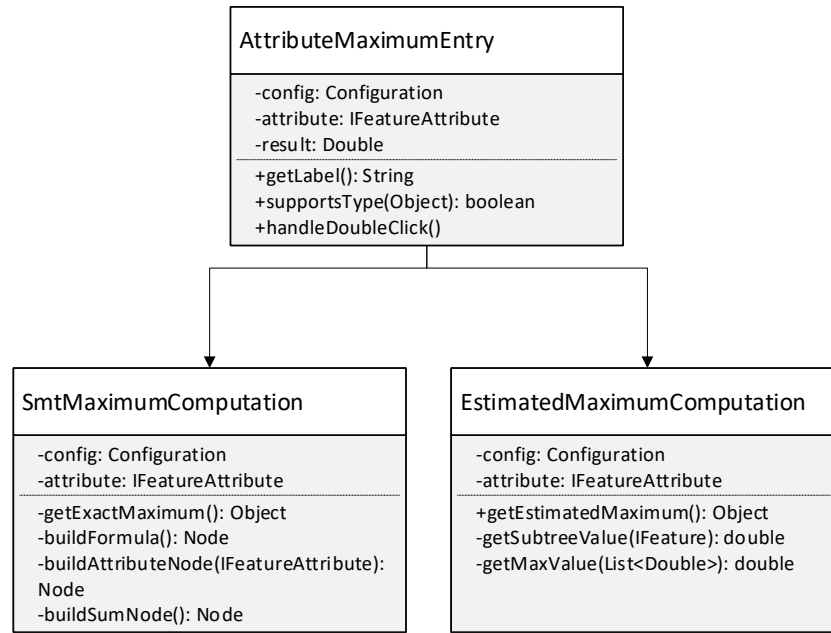


Figure 5.21: Outline entry responsible for computing the maximal sum of the attributes values

Now, we specify the implementation of the computations necessary for our outline entries. First, we describe the classes computing the ranges using SMT. Afterward, the classes which are responsible for the approximation are defined.

5.6.1 Compute Ranges using SMT

In this section, we extensively discuss the implemented solution for solving our algorithmic problem of computing attribute ranges using SMT, which was conceptually discussed in Section 4.3. The main topic to consider here is the class `SmtAttributeRangesComputation`.

Figure 5.23 represents the abstract class `SmtAttributeRangesComputation`. Every instance is defined by its input configuration `config` and the numerical attribute `attribute` to optimize for. Additionally, this class defines the shared methods between `SmtMaximumComputation` and `SmtMinimumComputation`.

`buildAttributeNode(IFeatureAttribute)` creates our formulas representing the attributes in the formula. Given an `IFeatureAttribute` owned by a feature it returns the node $(feat \Rightarrow att_{feat} = \pi(feat)) \wedge (\neg feat \Rightarrow att_{feat} = 0)$.

`buildSumNode()` builds our `Node` representing the sum variable we are optimizing for. It sets the variable equal to the sum of all attribute-variables created by `buildAttributeNode()`.

`buildFormula()` combines our different formulas to the target first-order logic formula consisting of the conjunctive normal form representing the feature model and its constraints, the literals representing selected and unselected features, the different

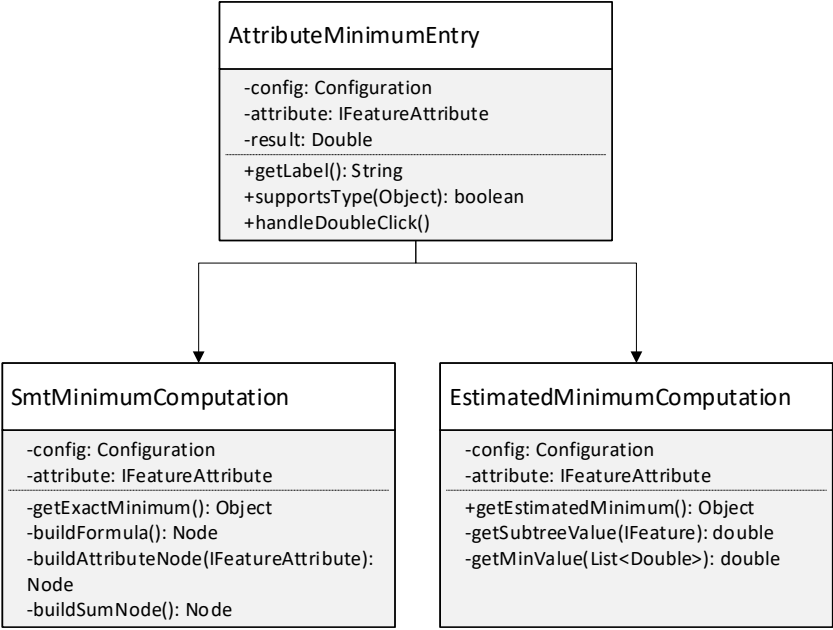


Figure 5.22: Outline entry that is responsible for computing the minimal sum of the attributes values

attribute nodes given by `buildAttributeNode(IFeatureAttribute)`, and the sum node given by `getSumNode()`. In the end, a conjunction of all the part nodes will be returned to be used in our analysis.

Both, `SmtMaximumComputation` and `SmtMinimumComputation` use the same formula returned by `buildFormula`. Additionally, both create an `SmtProblem` with the returned formula and all the variables, consisting of the literals appearing in the entire formula, at the start of `getExactMaximum()` and `getExactMinimum()`. However, different analyses are used. `SmtMaximumComputation` uses the `FeatureAttributeMaximumAnalysis` and `SmtMinimumComputation` the `FeatureAttributeMinimumAnalysis`. The exact attribute ranges consist of the return values of the `analyze()`-function of both analyses.

Next, we specify `EstimatedMinimumComputation` and `EstimatedMaximumComputation`, representing our approximated attribute sum ranges.

5.6.2 Estimation Algorithm

In this section, we describe the implementation of the approximated attribute ranges given a partial configuration. First, we discuss the basics of the implementation that are used for the minimum and the maximum. Afterward, we specify the unique aspects of both implementations.

Both classes are identified by a given `Configuration` and an `IFeatureAttribute`. Additionally, both are only usable for numerical attributes.

`EstimatedMinimumComputation` calculates the estimated minimum of our attribute value sum for a given configuration. `getSubtreeMinimum(ExtendedFeature)` is the

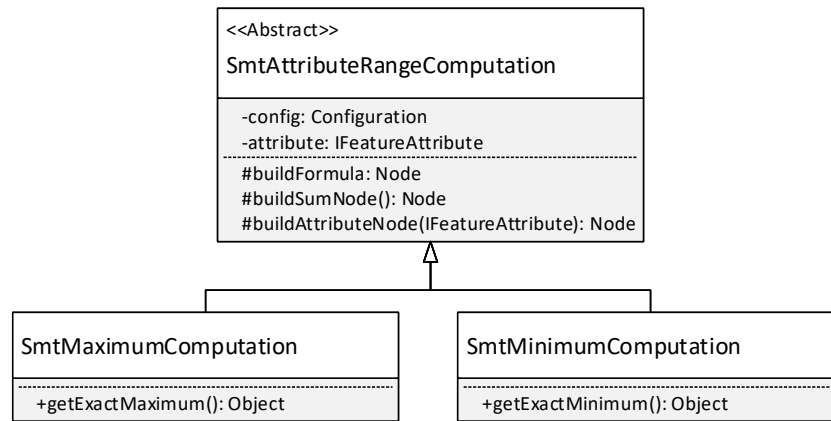


Figure 5.23: Class diagram for the range computation using SMT

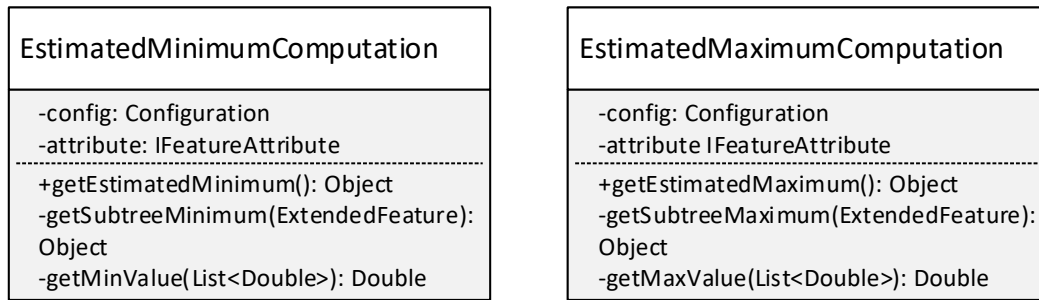


Figure 5.24: Class diagram for the range computation using the estimation algorithm

implemented equivalent of *getSubtreeMinimum(Feature)* of our minimal estimation algorithm 1 in Section 4.4. It is a recursive function that computes the minimal possible value for the sub-tree whose root is the input feature. Now, to acquire the result for the entire model *getSubtreeMinimum(r_{FM})* has to be called, which happens in *getEstimatedMinimum()*. Calling this function, with the root of the feature model as input, returns our approximated minimal sum which is then added to the label of **AttributeMinimumEntry**.

EstimatedMaximumComputation computes the estimated maximal sum of our attributes values for a given configuration. *getSubtreeMaximum(ExtendedFeature)* is the equivalent to *getSubtreeMaximum(Feature)* from our algorithm 2 in Section 4.4. The function recursively computes the maximal possible sum of attribute values for the sub-tree whose root is given by our input **ExtendedFeature**. To acquire the over-all maximal value sum, *getSubtreeMaximum(r_{FM})* has to be called, which happens in *getEstimated()*. The value returned by this call is our approximated maximal sum and is added to the label at the initialization of **AttributeMaximumEntry**.

5.7 Summary

We began by introducing the framework `FEATUREIDE` and the solvers that we were going to implement. Afterward, we introduced `PROP4J` which can be used easily to build propositional formulas. We extended `PROP4J` to create restricted first-order expressions and described the extension in detail. We concluded that `PROP4J` is not a suitable input for the solver type `IncrementalSolver` because it misses the functionality needed to find explanations.

Therefore, we implemented the data structure `ISolverProblem` representing an unmodifiable SAT or SMT problem that uses the `PROP4J` extension internally with no access from the outside. It also provides the mapping which is necessary to find explanations. Next, we implemented the abstract data type `IncrementalSolver` as a `JAVA` interface. We described the architecture of the interface in detail and described how to add new solvers. Furthermore, we implemented `SAT4J` and the `JAVASMT` solvers as an SAT solver to compare their efficiency on the automated analysis on feature models. Additionally, we implemented the `JAVASMT` solvers as SMT solvers which can be used to evaluate the efficiency for the feature attribute range computations. Last but not least, we created a factory for `SAT4J` and a factory for `JAVASMT`. Every factory controls the instantiation of the various analyses and the solvers by giving an `ISolverProblem` as input.

The other important topic we implemented are configuration statistics, including our computations of attribute ranges. We realized the statistics by creating an outline containing different properties of a configuration. `IOutlineEntry` is the interface for these properties. Additionally, we separated the statistics into calculations regarding the selection and computations regarding feature attributes. The ranges of the sum of an attribute's values are described by an entry for the minimum and maximum, respectively. These entries are defined by the configuration and one corresponding attribute and contain one exact and one approximated computation. In the default state, the outline displays approximated values computed by the heuristic. However, after a double click on the respective entry the exact value is computed with an SMT solver.

6. Evaluation of Sat4J and JavaSMT on SAT-based Tasks

The automated analysis is a vital component for modern SPLs tools. The most common tool to perform the automated analysis are SAT solvers because they are very efficient. The increasing attraction and performance for SMT solves made us interested whether SMT solvers are superior to SAT solvers in performing the automated analysis of feature models. Therefore, we created the abstract data type `IncrementalSolver` to compare both kinds of solvers regarding their efficiency to perform the various analyses. We implemented the abstract data type `IncrementalSolver` and the analyses into the framework `FEATUREIDE`. Now, we want to evaluate our implementation by comparing the `SAT4J` solver with the different solvers provided by `JAVASMT`. We begin by introducing the research questions for this thesis in [Section 6.1](#). In [Section 6.2](#), we define the setup for the evaluation including the used computer environment, the used models, and the solvers used to evaluate the data. Afterward, in [Section 6.3](#), we compare the various solvers on the automated feature model analysis and their results. In [Section 6.4](#), we compare the various solvers in finding explanation for the defects presented in [Section 3.3](#). In [Section 6.5](#) we summarize the process of the evaluation and answer the research questions with the conclusions made from the previous sections.

6.1 Research Questions

The abstract data type `IncrementalSolver` was implemented in the open-source framework `FEATUREIDE` in [Chapter 5](#). With the help of the implementation, we evaluate the following research questions.

- *RQ1*: Are SMT solvers superior to SAT solvers regarding efficiency?
- *RQ2*: Do the enhancements introduced in [Section 3.3.2](#) impact the efficiency of the core and dead feature analysis?

- *RQ3*: Do the enhancements introduced in Section 3.3.3 impact the efficiency of the false-optional features analysis?
- *RQ4*: What are the most suitable solvers for each specific analysis and do their combination speed up the whole process?
- *RQ5*: Which is the most suitable solver for finding explanations?
- *RQ6*: Is it possible to find explanations for defects on-the-fly? on-the-fly means that the explanation can be found within 200 ms.

6.2 Evaluation Setup

In this section, we clarify the most critical aspects of the evaluation of the solvers.

Evaluated Solvers

We implemented the SAT solver SAT4J¹ and the SMT solver API JAVASMT into FEATUREIDE. Hence, we evaluate SAT4J and the solvers PRINCESS², SMTINTERPOL³, and Z3⁴ provided by JAVASMT. PRINCESS and SMTINTERPOL are both JAVA-based solvers and can be installed by adding a reference to the specific libraries. For the usage of Z3, we needed to build the binaries and a static library for the use within JAVASMT. We skip the solver MATSAT5 and the extension OPTIMATSAT from JAVASMT. For the evaluation of finding explanations, we also need to skip PRINCESS because it does not currently support computations of the unsatisfiable core. Hence, we only evaluate SAT4J, SMTINTERPOL, and Z3 on finding explanations.

Evaluated Feature Models

For this chapter, we name feature models between 1,000–5,000 features medium-sized feature models and feature models with more than 5,000 features large feature models. In this chapter, we evaluate 116 medium-sized feature models. The 116 medium-sized feature models are real models and are based on different variability languages. There exist models for KCONFIG and the *component definition language* (CDL). KCONFIG is used in some software projects and was primarily designed for the configuration management of the Linux kernel [KTM⁺17b]. CDL is designed for a specific embedded system, where every model represents a configuration for a hardware system [KTM⁺17b]. Both languages are different to the format used by FEATUREIDE. Knüppel et al. created an algorithm to convert both languages to the FEATUREIDE format [KTM⁺17b]. Therefore, we evaluate all CDL and KCONFIG feature models mentioned by Knüppel et al. except the feature model for Linux [KTM⁺17b]. All 116 medium-sized feature models contain between 1,000–1,500 features and between 600–1,000 constraints.

¹<http://www.sat4j.org/>

²<http://www.philipp.ruemmer.org/princess.shtml>

³<https://ultimate.informatik.uni-freiburg.de/smtinterpol/>

⁴<https://github.com/Z3Prover/z3>

Additionally, we evaluate four snapshots of the large industrial feature model *automotive* provided by the partners of FEATUREIDE. The feature models contain between 14,000–18,500 features and between 666–1,300 constraints. Hence, all automotive feature models are large feature models. We also measured the number of clauses of the conjunctive normal form to further compare the complexity of medium-sized feature models and large feature models. We measured 3,068 clauses on average for the medium-sized feature models. In contrast, we measured 319,599 clauses on average for large feature models. Thus, the large feature models are far more complex and harder to solve than the medium-sized feature models.

To summarize, we evaluate 120 feature models. We have 116 medium-sized feature models and four large feature models.

Solving Environment

All analyses are computed on the same computer with the following specifications.

- OS: Windows 10, 64-bit system architecture
- CPU: AMD Ryzen 7 1700X, 8x3.4GHz
- RAM: 8 GB DDR4-RAM, 2400 MHz, Ballistix Sport LT

Our implementation into FEATUREIDE is forked of the version FEATUREIDE 3.5 and is available on Github.⁵ As for ECLIPSE, we used the ECLIPSE Oxygen June 2017 build. We installed JAVA 1.7 on the machine and installed the following versions of JAVASMT and the versions for the different solvers:

JAVASMT: Version: 1.0.1-219-g535f988

SAT4J: Version 2.3.5.v20130525

Z3: Version 4.6.0

PRINCESS: Binary release 2018-01-17

SMTINTERPOL: Version 2.5

6.3 Efficiency of SAT and SMT on the Feature Model Analysis

In this section, we evaluate the solvers on the task of the automated analysis of feature models. For the procedure, we compare the four implemented solvers for every analysis regarding the total runtime. The results for medium-sized feature models are shown as box plots that contain the total runtime of every solver. Additionally, we show the percentage of every solver operation as bar diagrams separately. The results for large feature models are shown in scatter plots containing the total runtime of every solver for the four large feature models. The percentages of every solver operation is displayed as bar diagrams separately. We begin with the analysis to check if a given feature model is void.

⁵https://github.com/Subaro/BachelorThesis_Sprey_Sundermann

Void Feature Model Analysis

The results of the void feature model analysis are shown in [Figure 6.1](#). We can see that the computation time for medium-sized feature models of each solver is almost the same and that all SMT solvers have outliers. We can also see that the creation of the SMT solvers is more costly than the SAT solvers creation. Additionally, the results for the large feature models show that the SAT solver is faster and the effort to create the SMT solvers rise.

The void feature model analysis is the most straightforward analysis. Therefore, we do not expect a significant difference in the runtime for all solvers. Surprisingly, we conclude from the results that the SAT solver is faster than the SMT solvers for the void feature model analysis. The gap between the SAT solver and the SMT solvers is getting bigger especially for larger feature models. Additionally, we conclude that the effort to create SMT solver is higher than the effort to create SAT solvers. SAT4J accepts the input as a set of integers. The conversion from the Prop4J nodes to the integer is fast. In contrast, the conversion from the nodes to the specific input required for JAVASMT is slower. Hence, the instantiation time for the SMT solvers is higher. For large feature models, the SAT solver performs the analysis at least 34 times faster than the other solvers. Even if we ignore the effort for the instantiation, the SAT solver is still faster than all SMT solvers.

Core and Dead Feature Analysis

We evaluate the unoptimized, optimized, and SAT4J optimized versions of the core and dead feature analysis. The results of the unoptimized version are shown in [Figure 6.2](#). We see that the unoptimized version depends almost only on checking satisfiability. The results show that the SAT solver is faster than the SMT solvers for both medium-sized and large feature models.

The unoptimized analysis iterates all features at least two times and does a satisfiability check for every iteration. The SAT solver performs the analysis for large feature models at least ten times faster than the SMT solvers. Hence, we conclude that the SAT solver is faster than the SMT solvers for the unoptimized core and dead feature analysis.

Next, we evaluate the analyses that support the filtering introduced in [Section 3.3.2](#). The results of the analysis that uses filtering is shown in [Figure 6.2](#). We see that the SAT solver can solve the analysis multiple times faster than the SMT solvers. The difference between the percentages of the solvers differ greatly between SAT and SMT. The SAT solver depends almost only on the satisfiability checks. In contrast, most of the time for SMT solvers is occupied by other operations of the analysis.

With the help of the filtering, introduced in [Section 3.3.2](#), the effort for the analysis should be reduced theoretically at least by half. The optimization reduces the solving time for the SAT solver as assumed but does not enhance the solving process for SMT solvers. In contrast, the solving time increases for the SMT solvers. The cause for the long solving times has two reasons. First, the SMT solvers do provide an efficient way to retrieve a satisfying assignment. Second, a satisfying assignment of the SMT solvers is only available in the data structure provided by JAVASMT.

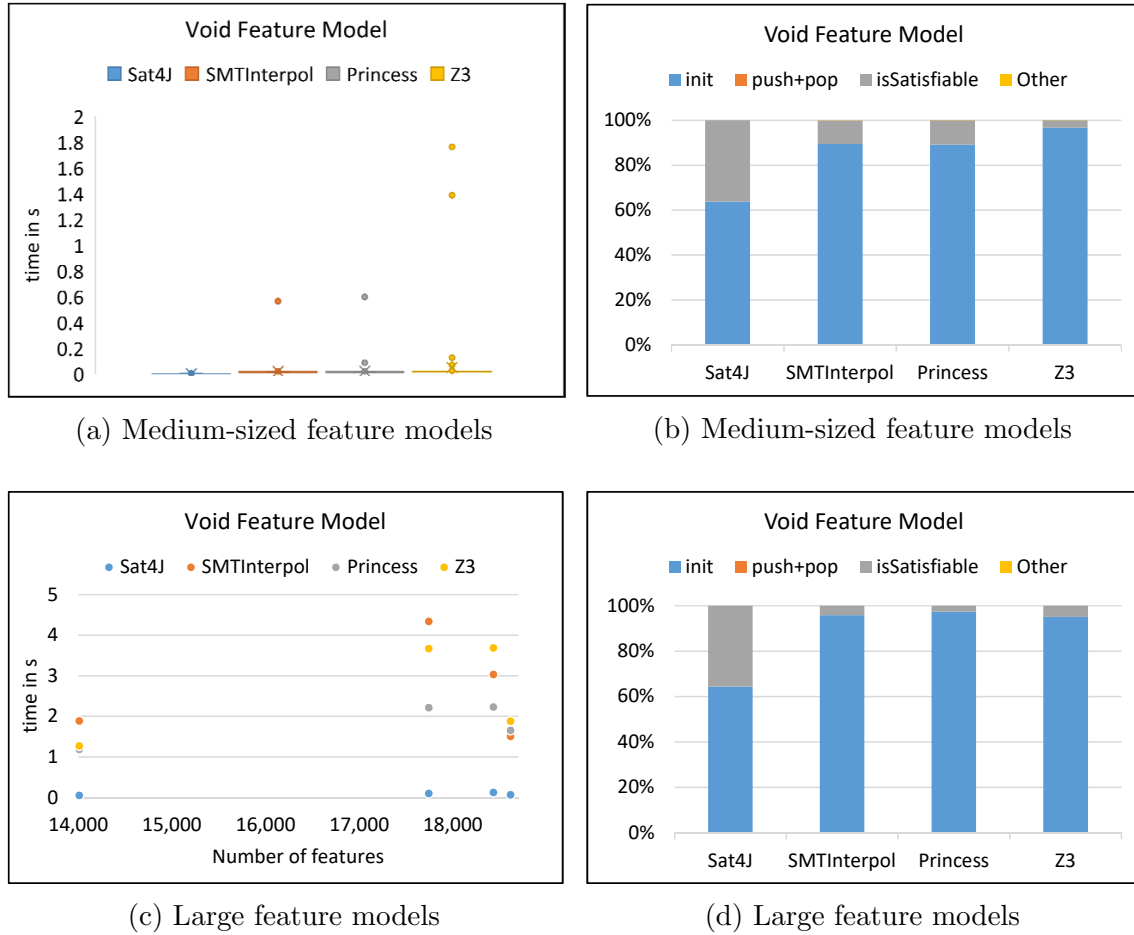


Figure 6.1: Results of the void feature model analysis. (a) and (c) show the total runtime of every solver. (b) and (d) show the percentages for the analysis steps of every solver

Therefore, we need to convert the satisfying assignment to the data structure used for the general analysis. Both reasons cover almost 100% of the percentages of the *other* analysis step. We conclude from the observations that the SAT solver is faster than the SMT solvers for the filtered core and dead feature analysis. Additionally, we conclude that the optimization impacts the performance for SAT solver positively and for the SMT solvers negatively. In future, it is possible to implement and to evaluate the optimization as a specific analysis for the SMT solver which can natively work with the data structure provided by JAVASMT. Hence, a conversion for the satisfying assignment will no longer be needed.

Last but not least, we evaluate the analysis that uses both filtering and a modified selection strategy introduced in Section 3.3.2. The analysis can only be performed on the SAT solver because the SMT solvers do not provide the functionality to modify the selection strategy. Therefore, instead of comparing the solvers, we compare the different versions of the core and dead feature analysis. The results are shown in Figure 6.4. For medium-sized feature models, we see that the solving time improves with each optimization. For large feature models, the filtering also increases the performance. In contrast, the filtering and the modified selection strategy only

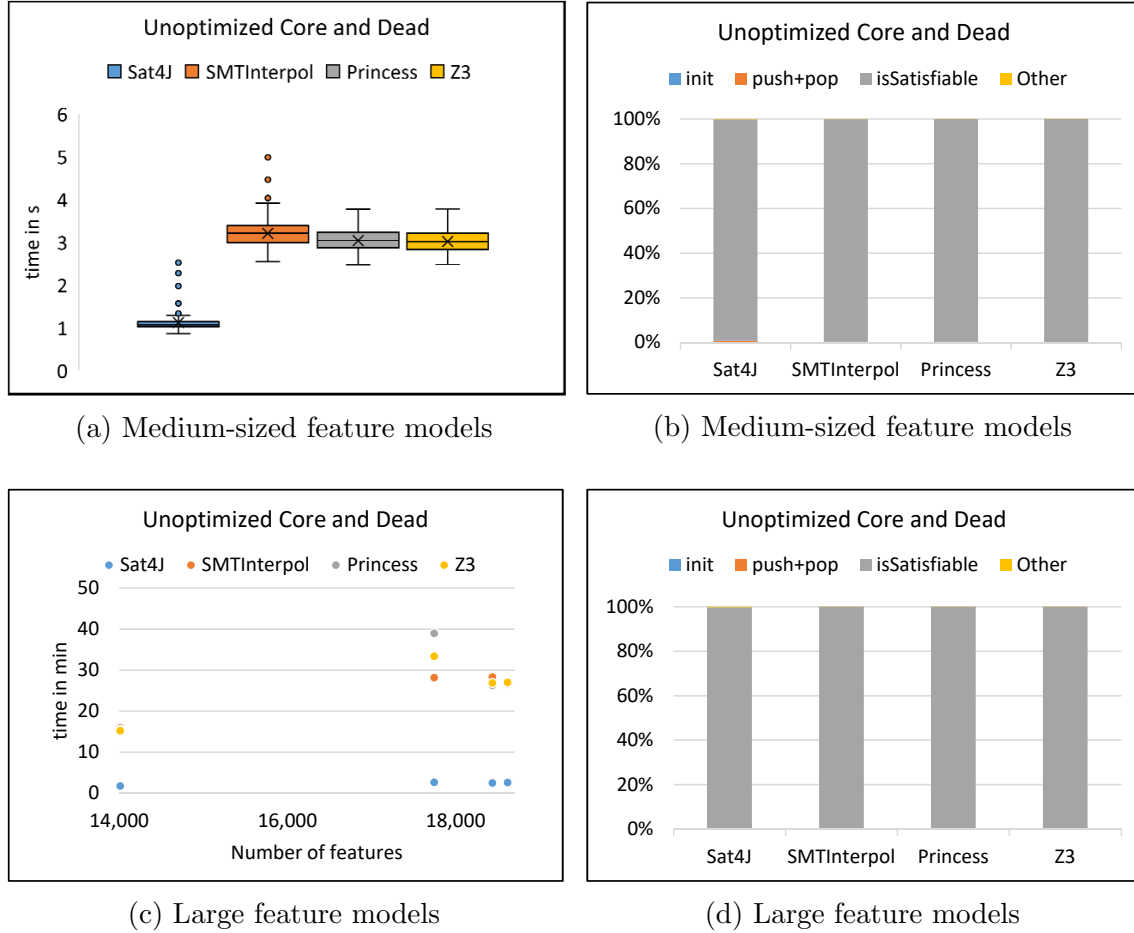


Figure 6.2: Results of the unoptimized core and dead feature analysis. (a) and (c) show the total runtime of every solver. (b) and (d) show the percentages for the analysis steps of every solver

improves the performance for the smallest automotive feature model. For the other models, the performance is the same or even worse.

The results are surprising because the analysis also does filtering and should at least be as fast as the analysis with only filtering. Hence, the modification of the selection strategy is not an improvement for large feature models. In contrast, it makes the process slower. We conclude that the modification of the selection strategy improves the performance only for medium-sized or smaller feature models and that the SAT solver is the most suitable solver for the core and dead feature analysis that uses both optimizations.

False-Optional Feature Analysis

We evaluate the unoptimized, optimized, and SAT4J optimized versions of the false-optional feature analysis. We begin with the unoptimized analysis. The results are shown in Figure 6.5. We see that the unoptimized analysis depends only on satisfiability checks. Additionally, we see that the SAT solver is faster than the SMT solvers for both medium-sized and large feature models.

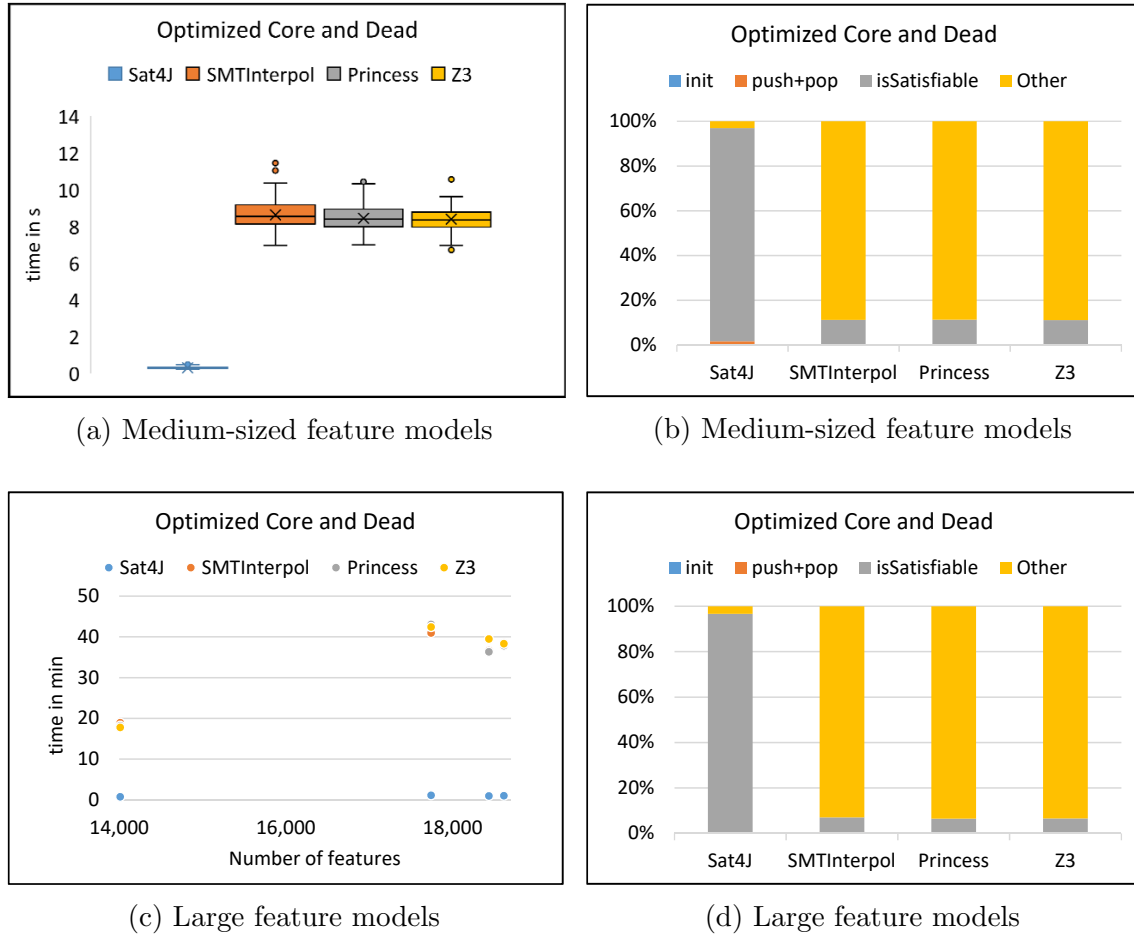


Figure 6.3: Results of the optimized core and dead feature analysis. (a) and (c) show the total runtime of every solver. (b) and (d) show the percentages for the analysis steps of every solver

The difference in performance is increasing with the size of the feature model. Therefore, we can conclude that the SAT solver is faster than the SMT solvers for the unoptimized false-optional feature analysis.

Now, we evaluate the false-optional analysis that is optimized with the filtering introduced in Section 3.3.3. The results for the analysis are shown in Figure 6.6. We see that the SAT solver is faster than the SMT solvers for both medium-sized and large feature models. As the bar diagrams show, the *Other* analysis step is the most costliest part of the analysis for the SMT solvers. In contrast, most of the analysis for the SAT solver is occupied by satisfiability checks.

The filtering for the false-optional feature analysis leads to the same problem for SMT solvers that we faced in the filtered core and dead feature analysis. We conclude with the help of the results that the most effort of the analysis is done by the filtering itself and leads to faster results for the SAT solver of both medium-sized and large feature models. In contrast, the optimization increases the solving time for the SMT solvers because we also need to convert the solutions to the data structure defined by the analysis. Nonetheless, if we ignore the conversion of the data structure, then the optimization can significantly improve the performance of the analysis also for the

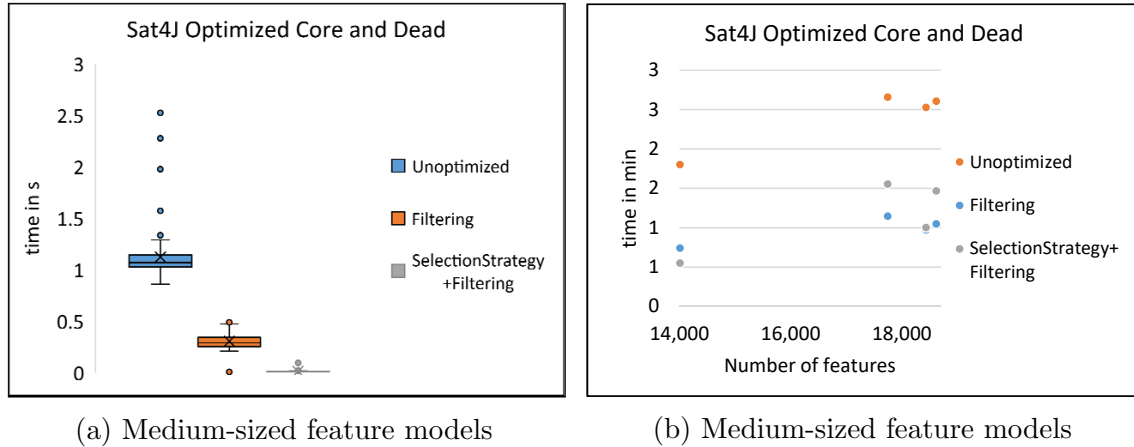


Figure 6.4: Results of the SAT4J optimized core and dead feature analysis. (a) and (c) show the total runtime of every solver. (b) and (d) show the percentages for the analysis steps of every solver

SMT solvers. In future, it is possible to create a specific analysis that can natively work with the structure provided by JAVASMT.

Last but not least, we evaluate the analysis that uses both filtering and a modified selection strategy introduced in Section 3.3.3. The analysis can only be performed on the SAT solver because the SMT solvers do not provide the functionality to modify the selection strategy. Therefore, instead of comparing the solvers, we compare the different versions of the false-optional feature analysis. The results are shown in Figure 6.7. We see for medium-sized and large feature models that every optimization can significantly improve the performance of the analysis.

We conclude from the results, that the optimizations improve the solving process significantly. The improvement also scales for large feature models allowing us to perform the false-optional analysis with the SAT solver in less than half a second for feature models with up to 18000 features. We also conclude that the SAT solver is the most suitable solver for the false-optional feature analysis.

Redundant Constraint Analysis

Next, we evaluate the redundant constraint analysis. The results for the analysis is shown in Figure 6.8. We see that the SMT solvers are faster for medium-sized feature models. For large feature models the SAT solver is faster than the SMT solvers. The percentages for the SAT solver and the SMT solvers differ greatly. While the analysis heavily depends on satisfiability checks for the SMT solvers, it is greatly influenced by the `push` and `pop` operations for the SAT solver.

The redundant constraint analysis is one of the most interesting analyses because it heavily depends on the `push` and `pop` operations. These operations are natively supported by SMT solver but not by SAT solvers. Therefore, we expect the SAT solver to be slower than the SMT solvers because we needed to implement the same behavior, especially for the SAT solver. As expected, the SMT solvers are faster than the SAT solver for medium-sized feature models. The effort of the `push` and `pop` operations slows the SAT solver down. Surprisingly, at some point, the SAT

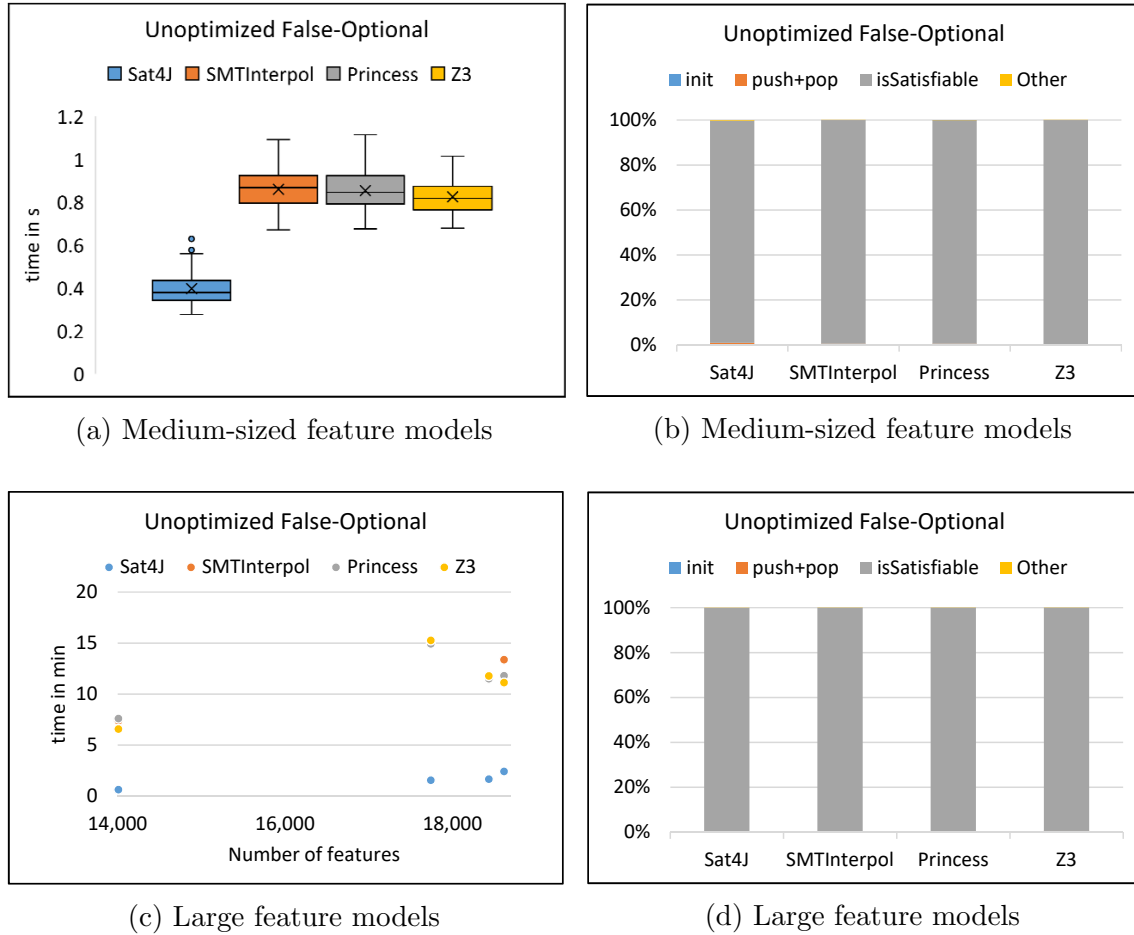


Figure 6.5: Results of the unoptimized false-optional feature analysis. (a) and (c) show the total runtime of every solver. (b) and (d) show the percentages for the analysis steps of every solver

solver catches up to the SMT solvers and becomes faster in solving the analysis. The difference in efficiency can be seen in the results for large feature models. Although that the SAT solver does not support the `push` and `pop` operations natively, it is faster than the SMT solvers for large feature models. Hence, we conclude for the redundant constraint analysis that the SMT solvers are more suitable for medium-sized or smaller feature models while the SAT solver is more suitable for large feature models. In future, it is possible to optimize the `push` and `pop` operations of the SAT solver. They are very slow because the removal of a clause from the internal formula is very slow for the SAT solver. Kanning created a method to optimize the removal of clauses for the SAT solver [Kan17]. Implementing that method into our prototype can optimize the performance of the redundant constraint analysis for the SAT solver.

Tautological Constraint Analysis

We evaluate the analysis for tautological constraints. The results for the analysis are shown in Figure 6.9. We see that the analysis heavily depends on the creation of the solvers and that the SAT solver is faster than the SMT solvers.

The analysis theoretically consists heavily of the creation of solvers and satisfiability checks. Hence, we expect the SAT solver to be faster than the SMT solver. As

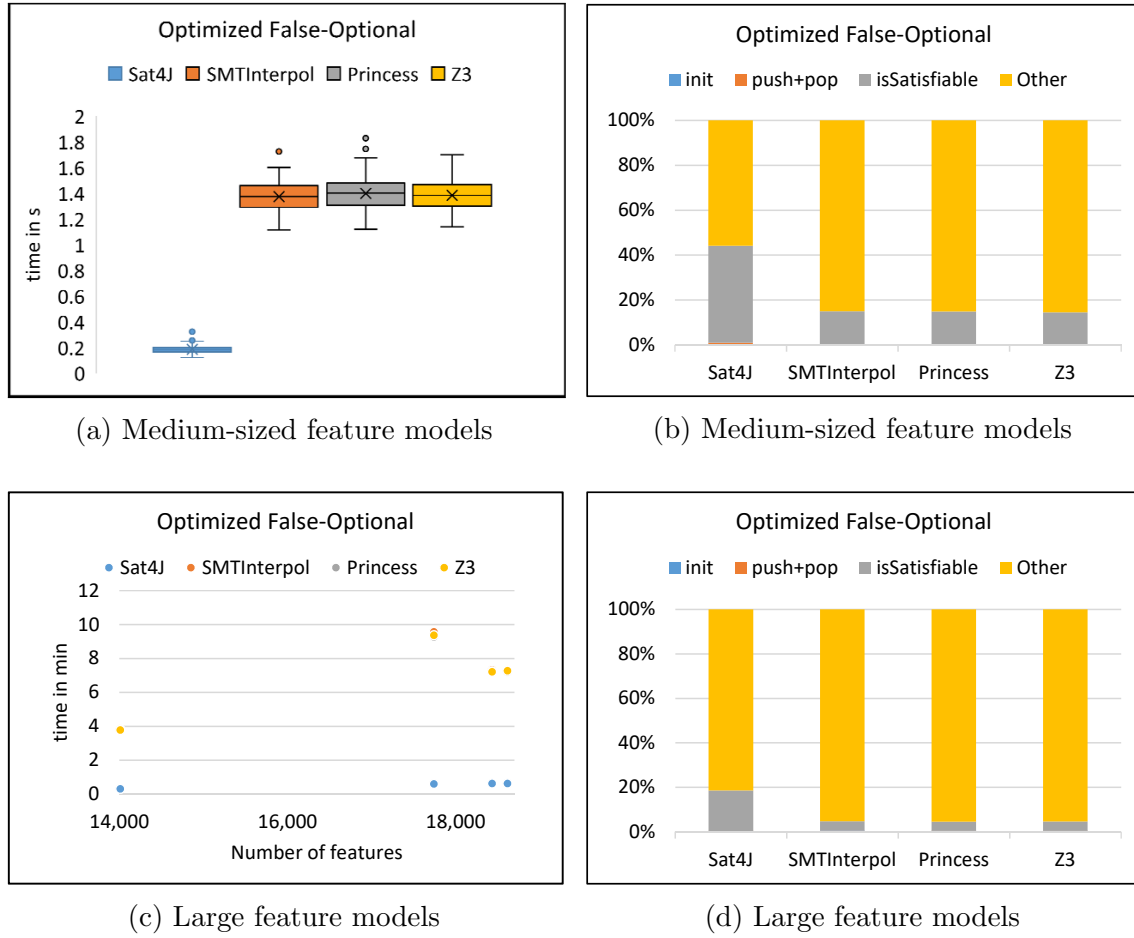


Figure 6.6: Results of the optimized false-optional feature analysis. (a) and (c) show the total runtime of every solver. (b) and (d) show the percentages for the analysis steps of every solver

expected the SAT solver is faster than the solvers from SMT. If we look at the distributions, we see that the SMT solver are heavily busy to create the solvers instead of solving. Therefore, we conclude that the SAT solver is the most suitable solver for the tautological analysis.

6.4 Explanations for the Feature Model Analysis

In this section, we evaluate explanations for dead features, false-optional features, and redundant constraints. Explanations for tautological constraints do not make sense because the constraint itself would always be the explanation. We randomly choose five of the 116 medium-sized feature models. The feature models are $\{aaed2000, adderII, viper, phycore229x, ocelot\}$. A list containing the name, number of feature, and number of constraints for all 120 feature models can be found in Table A.1. The solver used to find the explanations is created separately in the beginning and the instantiation is not considered in the evaluation for the explanations. After we detected the defects, we automatically measured the total runtime for a solver to find their explanations. The results are shown as box plot containing the runtime of every solver for medium-sized feature models and as scatter plot for large feature models. We begin with the explanations for dead features.

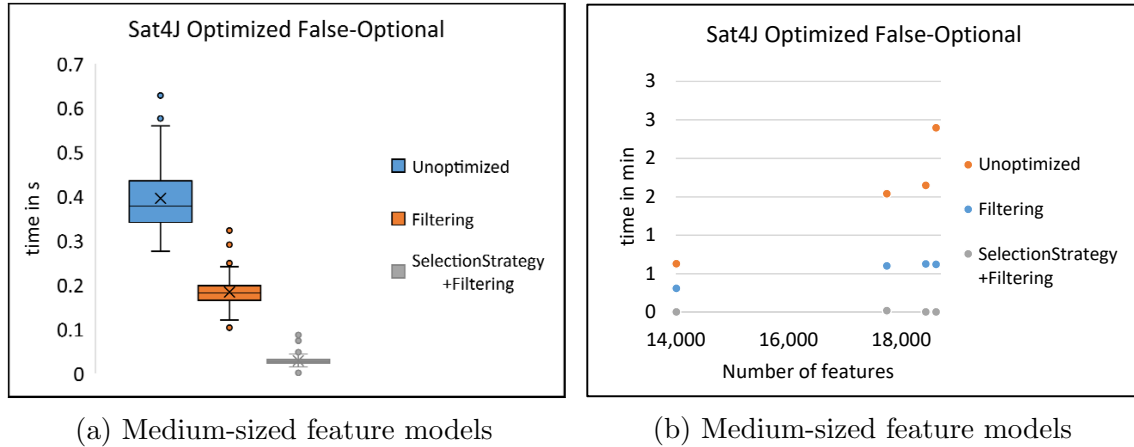


Figure 6.7: Results of the SAT4J optimized false-optional feature analysis. (a) and (c) show the total runtime of every solver. (b) and (d) show the percentages for the analysis steps of every solver

Explanations for Dead Features

The results for the evaluated explanations for dead features are shown in Figure 6.10. We see that all solver can find explanations faster than 3.5 milliseconds for medium-sized feature models and faster than 2 seconds for large feature models. The SMT solver SMTINTERPOL is the fastest solver for both medium-sized and large feature models.

Surprisingly, SMTINTERPOL and Z3 are both faster than SAT4J for large feature models. The difference between the fastest SMT solver and the SAT solver is 400 ms on average. This is a decisive advantage for the explanation because the explanations are generated on demand and should be as fast as possible. Hence, we conclude that SMTINTERPOL is the most suitable solver to find explanations for dead features. The performance with SMTINTERPOL is below 30 ms on average, even for large feature models. Therefore, it is acceptable to find the explanations for dead features on-the-fly.

Explanations for False-Optional Features

For the false-optional features, we measure the time needed to find explanations for them. The results for the explanations are shown in Figure 6.11. All solvers can find explanations faster than 3.5 ms for medium-sized feature models and faster than two seconds for large feature models. We observe that SMTINTERPOL is the fastest solver.

SMTINTERPOL is faster for medium-sized and large feature models. Hence, SMTINTERPOL is the most suitable solver in finding explanations for false-optional features. We conclude with an average runtime below 31 ms with SMTINTERPOL for large feature models that we can find the explanations for false-optional features on-the-fly.

Explanations for Redundant Constraints

We evaluate the explanations for redundant constraints. The results for the explanations are shown in Figure 6.12. SMTINTERPOL is the fastest solver for both,

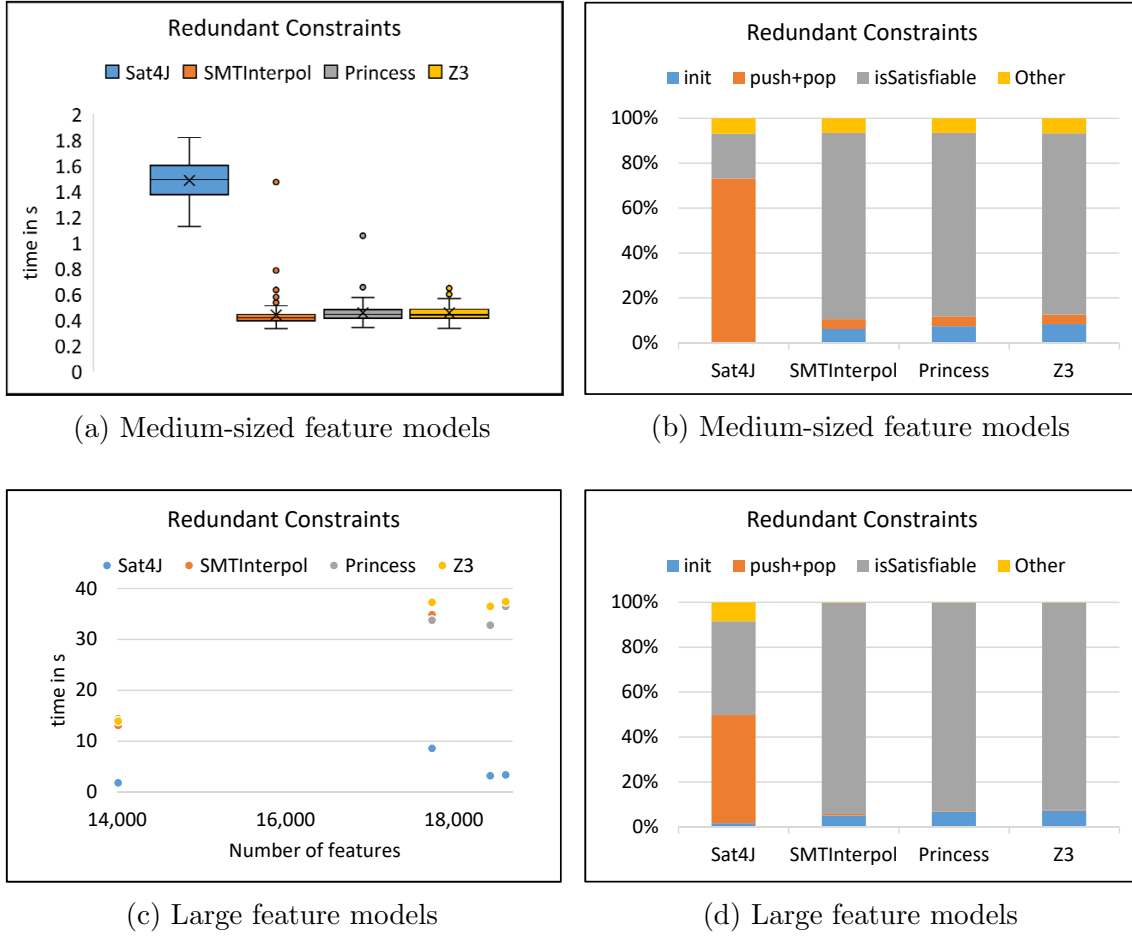


Figure 6.8: Results of the redundant constraint analysis. (a) and (c) show the total runtime of every solver. (b) and (d) show the percentages for the analysis steps of every solver

medium-sized and large feature models. All explanations are found in less than two seconds.

For both, medium-sized and large feature models, we can conclude that SMTINTERPOL is the fastest and most suitable solver in finding explanations for redundant constraints. The average runtime of SMTINTERPOL is about 49 ms. Therefore, we conclude that we can find the explanations for redundant constraints on-the-fly.

6.5 Summary

To answer the research questions, defined in Section 6.1, we summarize the conclusions gained from the evaluation. At first, we will answer whether SMT solvers are superior to SAT solvers regarding efficiency (RQ1). We concluded for all analyses, except for the redundant constraints analysis, that the SAT solver is faster than the SMT solvers. For the redundant constraint analysis, the SAT solver is slowed down because he does not support the `push` and `pop` operations natively. Therefore, all SMT solvers are faster than the SAT solver for medium-sized feature models. For large feature models, the SAT solver became faster than the SMT solvers because the `push` and `pop` operations are less important for large feature models. Hence, the SMT solvers

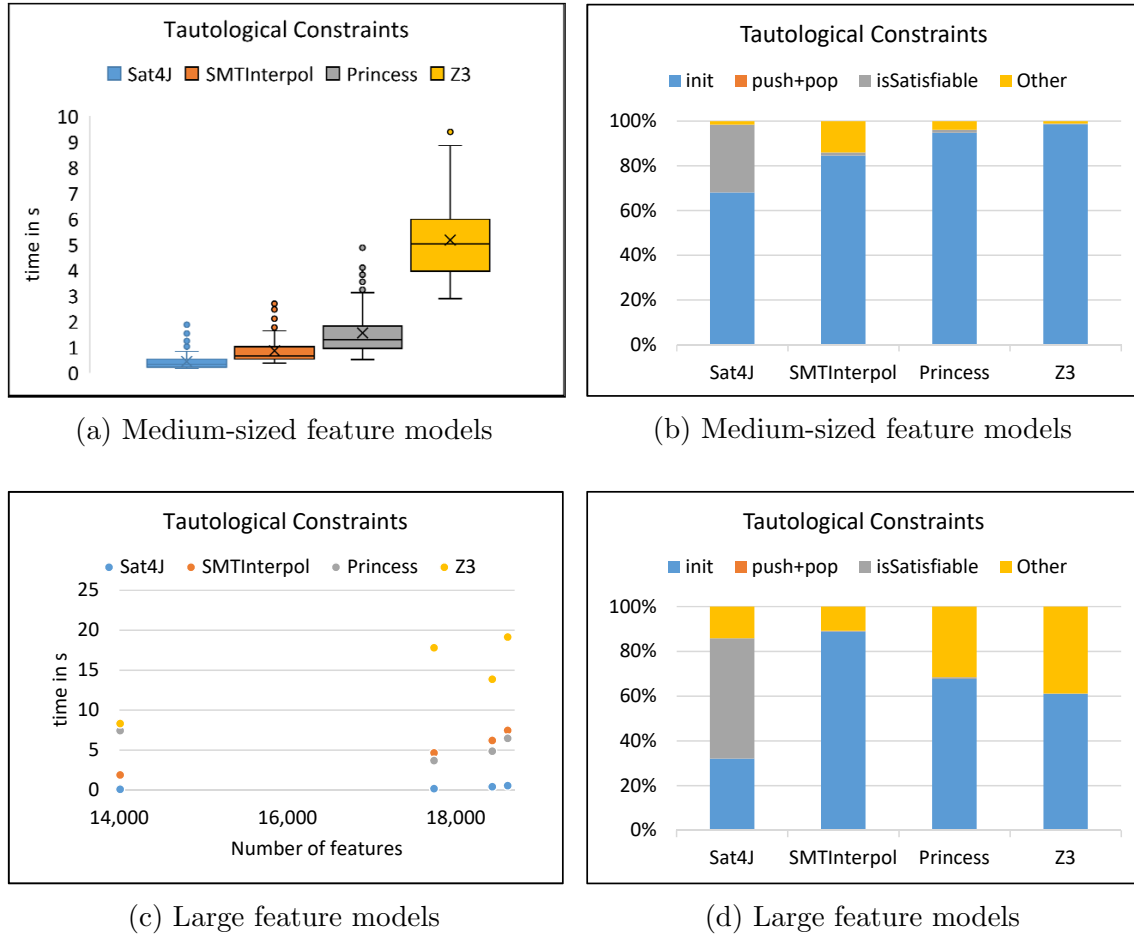


Figure 6.9: Results of the tautological constraint analysis. (a) and (c) show the total runtime of every solver. (b) and (d) show the percentages for the analysis steps of every solver

are not superior to SAT solvers regarding efficiency for the automated analysis of feature models. In contrast, we concluded that SMTINTERPOL is a lot faster than the SAT solver in finding explanations for any defect. Therefore, we can conclude that the SMT solvers are not superior to SAT solvers in general but there are problems where SMT solvers are more efficient than SAT solvers.

Next, we will answer whether the enhancements introduced in [Section 3.3.2](#) impact the efficiency of the core and dead feature analysis (RQ2). We concluded that the optimization of filtering greatly improves the core and dead feature analysis. In contrast, the combination of both filtering and modifying the selection strategy is only acceptable for medium-sized feature model. For larger feature models, the combination leads to longer solving times. Therefore, we can conclude that the filtering impacts the analysis positively, while the modification of the selection strategy impacts the analysis negatively.

For the core and dead feature analysis, we will answer whether the enhancements introduced in [Section 3.3.3](#) impact their efficiency (RQ3). We concluded that the both optimizations greatly improve the false-optional feature analysis. Therefore, both optimizations impact the analysis positively.

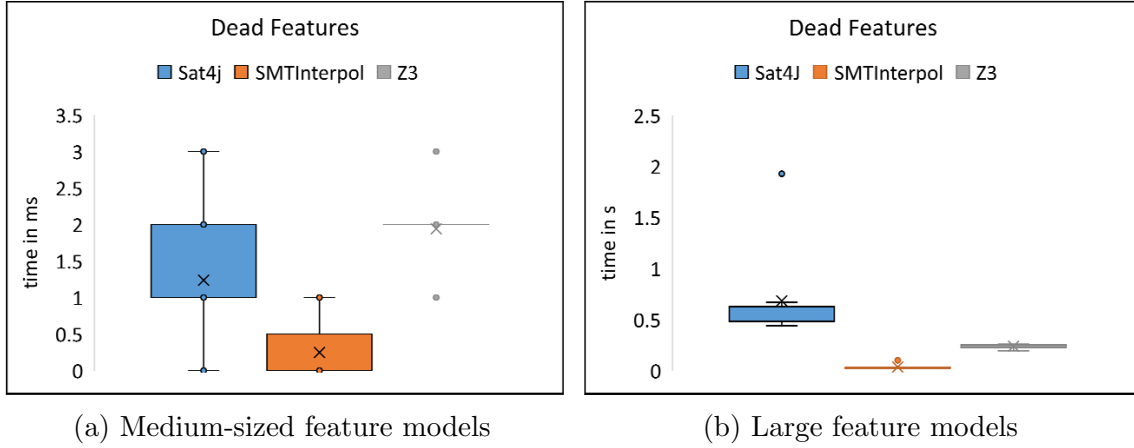


Figure 6.10: The results of finding explanations for dead features

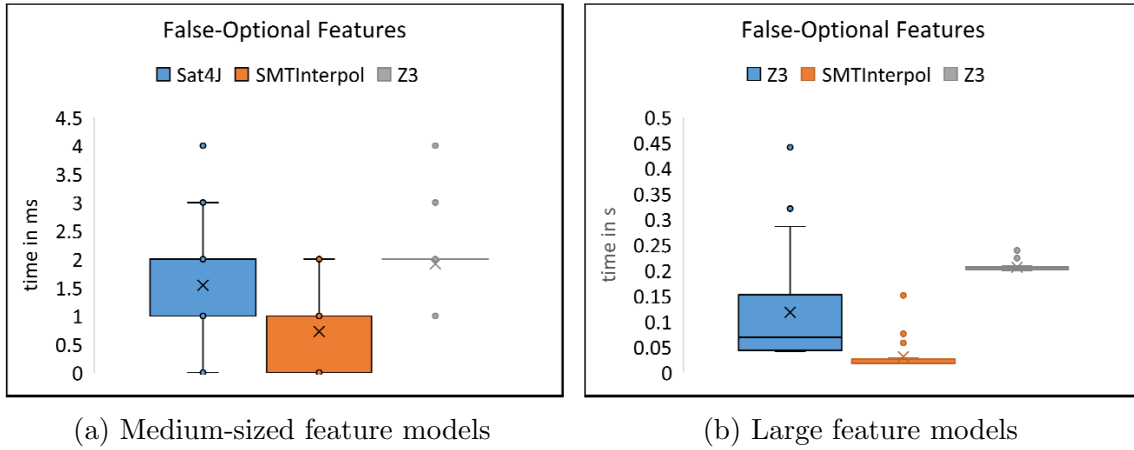
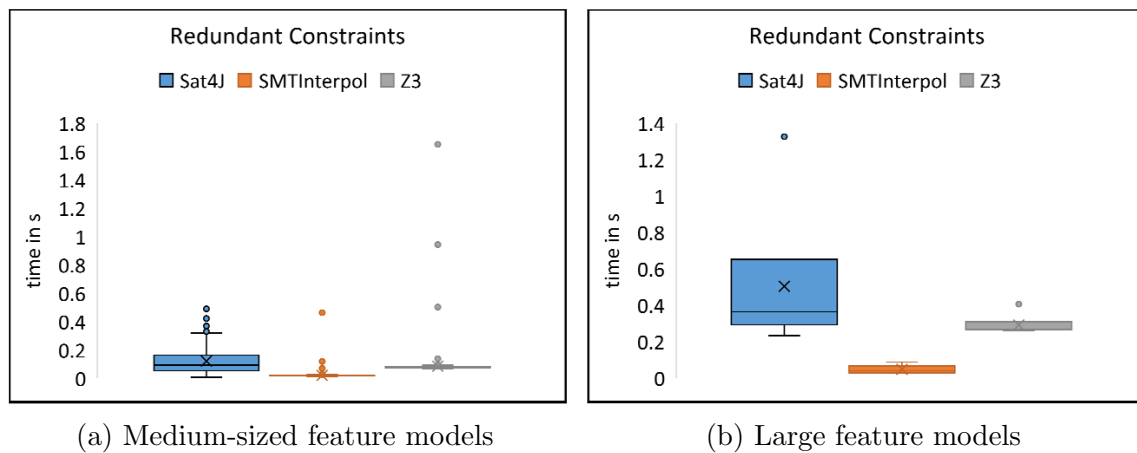


Figure 6.11: The results of finding explanations for false-optional features

Now, we want to summarize the most suitable solvers for each analysis and determine whether their combination could speed up the whole process (RQ4). We concluded that the SAT solver is the most suitable solver for all analyses, except for the redundant constraint analysis for medium-sized features. The most suitable solver for the redundant constraint analysis for medium-sized feature models is SMTINTERPOL. If we combine both solvers on the task of the automated analysis of feature models, we can slightly improve the overall process. The slight improvement of the process does not compensate the effort to implement the SMT solver.

Next, we determine the most suitable solver for finding explanations for defects (RQ5). We concluded that SMTINTERPOL is the most suitable solver for finding explanations for dead features, false-optional features, and redundant constraints because SMTINTERPOL could find the explanations several times faster than the other solvers.

Last but not least, we will answer whether it is acceptable to find explanations for defects on-the-fly (RQ6). We concluded that only SMTINTERPOL can find explanations for all kinds of defects with a runtime less than 200ms, even for large feature models. Therefore, we determine with the help of SMTINTERPOL that we can find explanations for all kinds of defect on-the-fly.



(a) Medium-sized feature models

(b) Large feature models

Figure 6.12: The results of finding explanations for redundant constraints

7. Evaluation of the Attribute Range Computation

In this chapter, we evaluate the implementation of computing attribute ranges described in [Section 5.5](#). The goal of this evaluation is to determine whether the implementations specified in [Section 5.6](#) are usable for computing attribute ranges for partial configurations of feature models. These implementations consist of the computation with an SMT solver ([Section 5.6.1](#)) and the approximation with our heuristic ([Section 5.6.2](#)). For the usability, we demand the time required for one computation to be suitable for on-the-fly computations. Therefore, a single computation should not exceed one second. The evaluation depends on the runtime of the computation using an SMT solver and using the heuristic. Additionally, we determine the quality of the approximated result by its difference to the exact result provided by the SMT solver.

This chapter is structured as follows: In [Section 7.2](#), we define the feature models used for the evaluation and the process of re-engineering them from configurators used in the industry. The technical setup used during the evaluation is described in [Section 7.3](#). In [Section 7.4](#), we analyze the run-time for both kinds of computations. In [Section 7.5](#), we present the difference between approximated and exact results and conclude about the estimation algorithm's usability.

7.1 Research Questions

This section specifies the scientific questions this evaluation is supposed to answer. These questions are separated by the method of computing the attribute ranges. *RQ1* references the computation using SMT described in [Section 4.3](#) and *RQ2* the approximation using the heuristic described in [Section 4.4](#).

- *RQ1.1*: Is the computation using SMT feasible for calculating attribute ranges of partial configurations on-the-fly?

- *RQ1.2*: How much time is spent on the computation of minimum and maximum of a range?
- *RQ2.1*: Is the computation using the heuristic feasible for calculating attribute ranges of partial configurations on-the-fly?
- *RQ2.2*: How close is the approximated minimum and maximum using the estimation algorithm to the exact results?

7.2 Creating the Models

In this section, we specify the engineering of the models used for our evaluation. The goal was to create examples that represent existing configurators to show the usability of the software in realistic use cases. First, we specify the type of configurators suitable for a conversion to a feature model. Afterward, the re-engineering process is described. In the end, the three re-engineered models used for the evaluation are presented.

The behavior of configurators we considered to use for the evaluation is now described. Most importantly, there has to be at least one numerical value that is defined for a majority of features. This value is required, as we optimize for its sum. Additionally, this value has to be static for every feature during the configuration process. This way the corresponding attribute can be modeled as non-configurable and our implementation of the range computation does not support configurable attribute values. Another important aspect is the comprehensibility of constraints, as we can only model properties of the configurator that are detectable.

Every feature model is re-engineered by trial and error. During this process, we have two main strategies. We start by selecting few features of the configurator to detect possible implications. For example, selecting a large power adapter might force us to pick one of the cases with a lot of storage. After finding such an implication, we add an according constraint to our feature model. The other strategy is to select as many features as possible. This way, we aim to detect exclusions, as picking a new feature that is not selectable any more results in a displayed conflict. For example, we include every multimedia hardware enhancement and then try to generate conflicts by including different software options. Here, we try to build enough products to include every pair of two features at least once. By using both strategies, we aim to detect the majority of constraints. However, we might overlook a few.

Now, we specify the necessary steps to re-engineer a configurator as a feature model in **FEATUREIDE**. First, every top-level category of the configurator is added as a child feature of the models root. Afterward, the elements of each category are inserted as children of the corresponding feature. Next, we want to determine the relationships between category features and their children. If exactly one element can be included per category, we create an *Alternative*-relation. Another case is the necessity of including one element but it is possible to select multiple. This implies an *Or*-relation. In the remaining cases, we use an *And*-relation. Besides the features and the parent-children relationships, the mandatory statuses of each feature are fundamental for the re-engineering of configurators. Most of the time,

certain categories and/or elements are necessary to include. In this case, we set the corresponding feature to *mandatory*. Otherwise, its status is set to *optional*.

Next, we want to create the constraints that are impossible to define within the structure of the feature model tree. Whenever the configurator indicates that the inclusion of an element leads to an invalid product, because of a previous inclusion of another element, we create an according constraint. After this step, the resulting feature model represents the product line of the configurator. Additionally, the value we are interested in has to be added to the feature model. To achieve this, we create a feature attribute and attach it to every feature of the model. Now, we set the attributes value of every feature to match the value of the corresponding configurator element. The resulting extended feature model can be used to optimize the sum of our attribute values. Now, we present the configurators we re-engineered.

Our first model represents the configurator of a Brompton bike.¹ This configurator is quite simple as it has only 54 features and no cross-tree constraints. The attribute we optimize for is the price of each component.

The second model represents the configurator of a VW Golf Trendline.² Using this configurator, the customer is able to create a specific car of the Golf Trendline family. Each feature has a price, which is the attribute we are optimizing for. The model consists of exact 250 features and 50 constraints.

Last but not least, the third model represents the configurator of the PC Richmond F.³ For this configurator, we detected 376 features and 12 constraints. Therefore, it is our largest model. It is worth noting, that every relation between a category and its elements were modeled using *Alternative*-relations. The attribute we optimize for is the price.

7.3 Evaluation Setup

In this section, we specify the technical setup used during the evaluation and describe the process of evaluating the range computations. To compute our ranges we added our implementation into a fork of FEATUREIDE version 3.5.⁴ Additionally, we used the SMT solver Z3. We were unable to compile the JAVA binaries for MATHSAT, which prevented us from using the other solvers implemented in Chapter 5 that support optimization of numerical variables.

- OS: Windows 10 64-bit
- CPU: Intel Core i5-6500, 4x3.2GHz
- RAM: 2x 8,0 GB DDR4-RAM, 2133 MHz

¹<https://www.brompton.com/Build-your-Brompton>

²<https://www.volkswagen.de/app/konfigurator/vw-de/de/der-golf/30315/38150/trendline?page=engine>

³<https://www.computerwerk.de/Extreme-PC/Extreme-Gaming-PC-Richmond-F::2836.html?XTCSid=7b4qggh0fi2irg5g7evociq4p2>

⁴https://github.com/Subaro/BachelorThesis_Sprey_Sundermann

- JAVA: 1.8
- Z3: 4.6.0

For each model, we are interested in the efficiency of computing attribute ranges using an SMT solver and using the heuristic and in the quality of the approximated values. However, we expect the results to vary for different configurations. To achieve representative measurements, we consider configurations provided by two different methods. First, we try to emulate configurations at the start of a configuration process. We aim to represent scenarios similar to our example in [Section 4.1](#), in which just a few features are included. To achieve this, we create valid configurations with only one random feature selected. Our other method for creating configurations represents the end of a configuration process, where a lot of features are already selected. These types of configurations are acquired by randomly deciding on the inclusion of each feature. However, the resulting configuration has to be valid.

The following procedure is performed for all three models we engineered in [Section 7.2](#). First, we generate our configurations which consist of 50 random and 15 with one random feature selected. Afterward, we calculate the ranges for each configuration using the Z3 solver and the heuristic and track the respective results and the runtimes.

7.4 Evaluation of Efficiency

This section describes the evaluation regarding the runtime of the computation using an SMT solver and the heuristic. We aim to answer, whether the Z3 solver and the heuristic are usable for on-the-fly computations(*RQ1.1, RQ2.1*). First, we specify the results we are interested in. Then, we analyze the measured results.

For all the created configurations, we track the time needed for initializing the SMT solver, computing the minimum using the solver and the heuristic, and computing the maximum using the solver and the heuristic. After measuring the described data for every configuration run, we want to obtain the following values for each data set:

1. The overall run-time for calculating ranges using SMT, which is the sum of the required time to create the SMT-problem, calculate the minimum, and calculate the maximum. These values are used to answer *RQ1.1*.
2. The run-time distribution of creating the formula and calculating the minimum and maximum using SMT. This is used to answer *RQ1.2*.
3. The overall run-time for approximating the attribute ranges using our heuristic. This is supposed to answer *RQ2.1*.

7.4.1 Runtime of Computing Attribute Ranges

In this section, we analyze the time required to compute attribute ranges of a partial configuration. For each model, we tracked the runtime to compute ranges for configurations at the start of the configuration process(one random feature selected)

and configurations the end of the configuration process (random). The results are used to answer *RQ1.1*, *RQ2.1*.

Figure 7.1 shows the runtime of computing the attribute ranges for our feature model representing the bike. The time required for computing the approximated value does never exceed one millisecond in both states of the configuration process. On the other side, calculating the ranges using the SMT solver never exceeds half a second. However, the higher effort to calculate ranges of configurations with only one feature selected is already noticeable, as most runtimes of the computations for the start of the configuration progress lie between 75–150 ms and between 20–50 ms for the end of the configuration progress.

In general, we consider a runtime of less than one second as suitable for on-the-fly computations. Therefore, for the feature model representing the bike configurator the computation using the SMT solver and using our heuristic are suitable for both states of the configuration process for on-the-fly computations.

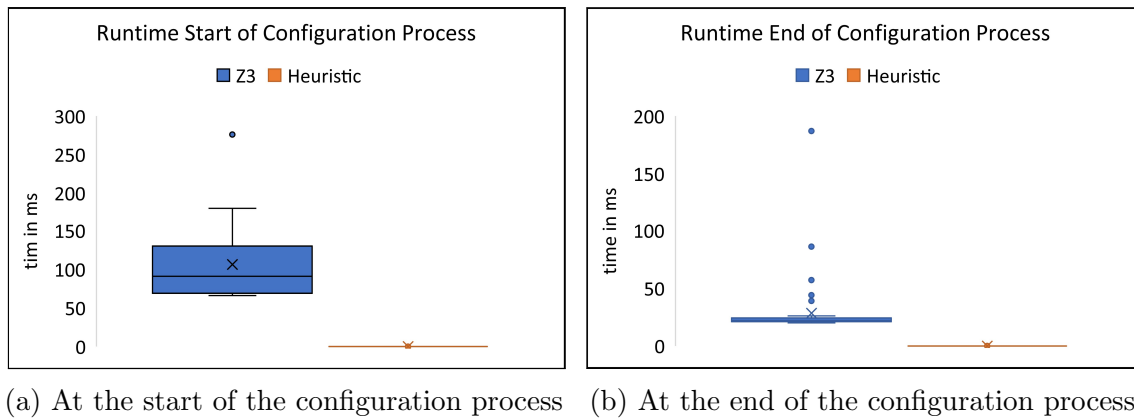
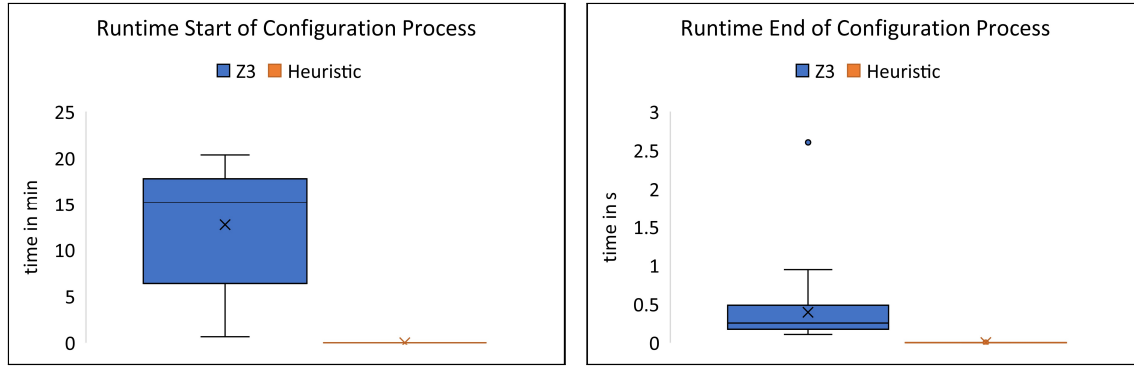


Figure 7.1: Runtime of the computation using Z3 and the heuristic for the bike feature model

Figure 7.2 shows the time required to compute the attribute ranges of our car feature model using the SMT solver and the heuristic. Even though the runtime for SMT is significantly higher than for the bike product line, approximating the value still does not exceed one millisecond. The runtime of computing the attribute ranges using the SMT solver even exceeds 20 minutes in one case. Additionally, the difference between the time required to compute ranges at the start and at the end of the configuration process is immense, as the median of the latter is around 250 milliseconds while the runtime median at the start of the process is about 15 minutes.

The measured runtimes for computing the attribute ranges at the start of the configuration process for the car feature model are too high to consider them suitable for on-the-fly computations. Therefore, we conclude that an approximation is necessary. Additionally, the required time to approximate the ranges with our heuristic implies the usability of it. At the end of the configuration process, there are a fewer possible products resulting from the leftover choices. This explains the difference in computing ranges at the start and at the end of the configuration process. As only 1 out of 50 configurations representing the end of the configuration process exceeded, we consider the SMT solver to be suitable for these configurations.



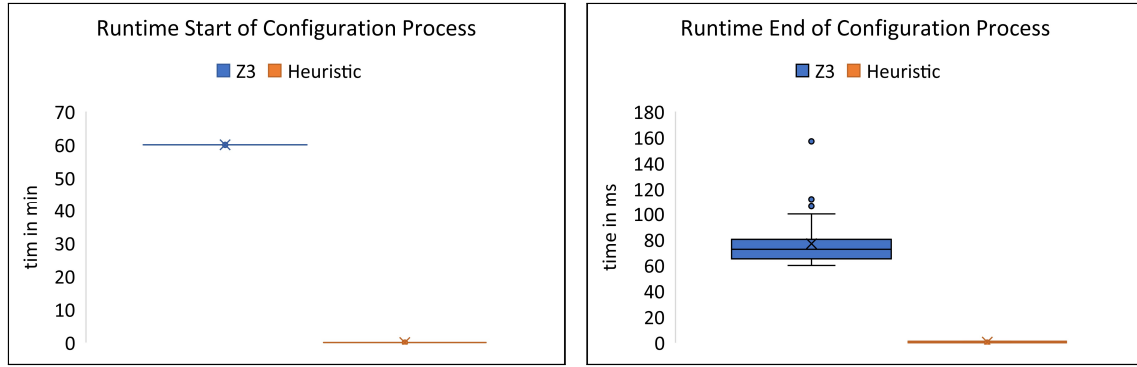
(a) At the start of the configuration process (b) At the end of the configuration process

Figure 7.2: Runtime of computing attribute ranges using Z3 and the heuristic for the car feature model

Figure 7.3 displays the runtime of computing attribute ranges for the feature model representing the PC configurator. For this model, the heuristic had two outliers requiring 16 milliseconds to compute the attribute ranges. All other measured run-times of the heuristic did not exceed one millisecond. The time required to compute exact values with the SMT solver further increased, compared to the car model, for the computations at the start of the configuration process, displayed in 7.2a. For these configurations, we decided to include a timeout after one hour. Every measured time for the configurations representing the start of the configuration process exceeded this timeout. Furthermore, we attempted to compute the attribute ranges without a timeout three times. In all three attempts, the computation was interrupted by an out-of-memory exception after 10–15 hours, even though the entire RAM (16 GB) was available. However, the necessary time to compute ranges for configurations at the end of a configuration process is shorter than for the car feature model.

The results, regarding the time required to compute attribute ranges for the partial configurations of the PC feature model, imply similar conclusions as the measurements for the car feature model. Computing ranges at the start of the configuration process with the SMT solver is definitely not suitable for the on-the-fly computations, as we could not acquire results for the maximum at all. However, the heuristic still almost instantly delivers results. For configurations representing the end of the configuration progress, the computation using the SMT solver is suitable for on-the-fly computations, as the run-time does not exceed 0.2 seconds.

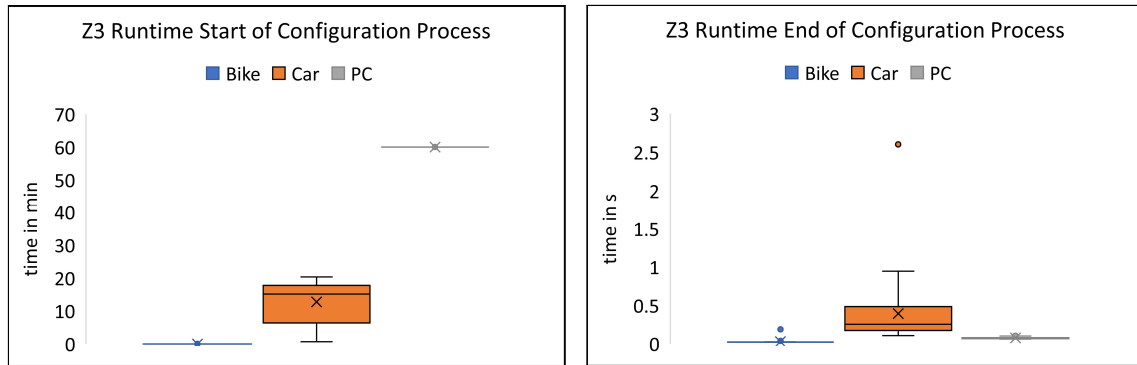
Figure 7.4 shows a comparison of the required time to compute the attribute ranges for each model with the SMT solver. For every feature model, the time required to compute attribute ranges at the start of the configuration process is greater. This discrepancy is significant for the car and PC models. While the results for the start of the configuration process could not even be acquired within one hour for the PC feature model, the largest amount of time spent to compute ranges at the end of the configuration process does not reach three seconds. Another thing to notice, is that the runtime of the computation for the PC is much greater than the computation for the car in 7.4a but, noticeably shorter in 7.4b.



(a) At the start of the configuration progress (b) At the end of the configuration progress

Figure 7.3: Runtime of computing attribute ranges using Z3 and the heuristic for the PC feature model

Even though all of our three feature models are comparatively small (far below 1,000 features each), the computation using the SMT solver is not suitable for the car and PC models (*RQ1.1*). Therefore, the heuristic is definitely necessary to obtain results on-the-fly (*RQ2.1*). Furthermore, the measured runtimes of the approximation imply that the estimation algorithm is definitely suitable for on-the-fly computations, which answers *RQ2.2*.



(a) At the start of the configuration process (b) At the end of the configuration progress

Figure 7.4: Runtime comparison of the computation of attribute ranges using Z3 for the three feature models

7.4.2 Distribution of the Runtime Needed for Computing Attribute Ranges with an SMT Solver

In this section, we analyze the runtime of the three necessary steps to compute attribute ranges with the SMT solver separately. The three steps consist of the initialization, calculating the minimum, and calculating the maximum. During the initialization, the formula is built, the SMT problem is created and configured, and the analysis is instantiated. We are interested in these results, because there might be scenarios in which only the minimum or the maximum is requested. We aim to answer whether its advantageous to only compute the requested minimum or maximum respectively in these scenarios (*RQ1.2*).

The distribution of the runtime of calculating attribute ranges with the Z3 solver for the bike feature model is shown in Figure 7.5. In both cases, 7.5a and 7.5b, the amount of time required for the initialization is similar (the median for computations at the start of configurations is 16 seconds for computations at the end 17 seconds). In 7.5a the longer time required to compute the maximum than to compute the minimum is noticeable. In 7.5b, both computations take a similar short amount of time, while the highest proportion of the run-time is spent for the initialization.

The initialization is very similar in the different states of the configuration process, as the formula only differs in the literals added for the selected and unselected features. This explains the similar runtimes of the initializations in 7.5a and 7.5b. Additionally, for our bike model we can observe that the initialization requires a big proportion of the overall runtime.

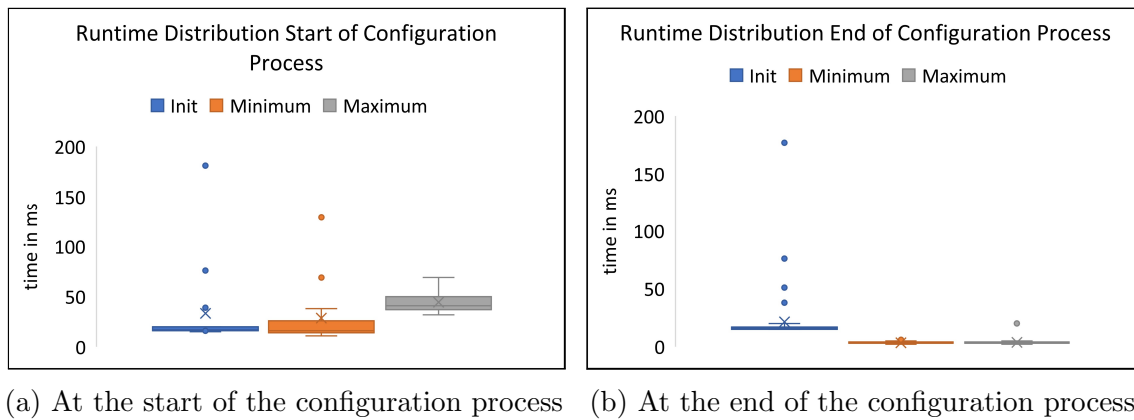
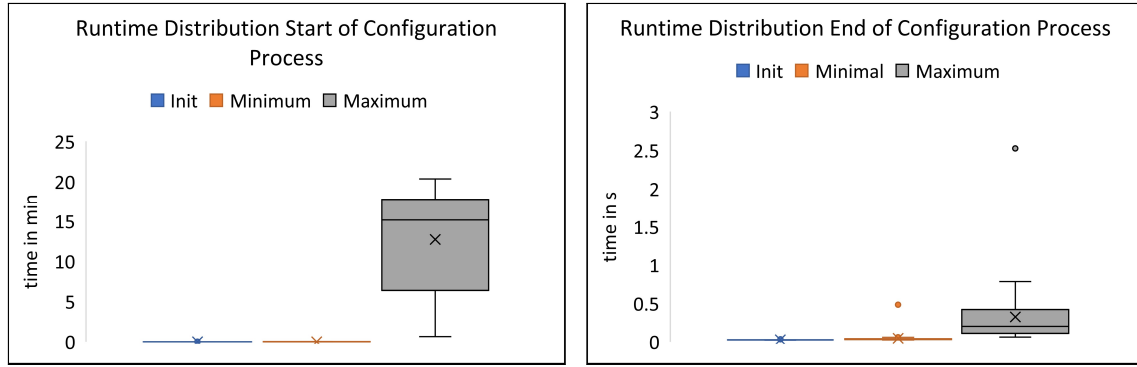


Figure 7.5: Required time for each step of the attribute range computation using Z3 for our bike feature model

Figure 7.6 show the proportion of the required steps to compute attribute ranges for the car feature model using the SMT solver. Especially for the computations at the start of the configuration process, the computation of the maximum takes much longer (median: 15.15 minutes) than computing the minimum (median: 0.218 seconds) or initializing (median: 0.028 seconds). Additionally, the initialization only needs slightly longer than the initialization for the bike feature model, while the time required to compute the minimum took about ten times longer on average at the start of the configuration process.

The most outstanding observation is the difference in time required to compute the maximum and the minimum. For the car feature model, it is much easier to compute the minimum. Therefore, if you are only interested in the minimum, calculating the ranges is a huge overhead. Additionally, we conclude that the effort for the initialization does not increase as much as the computations for larger feature models.

The distribution of the time required to compute attribute ranges for the PC feature model is displayed in Figure 7.7. The measured proportions at the start of the configuration process are similar to the measurements for the car feature model. For both models, the computation of the maximum takes significantly longer than computing the maximum or the initialization. However, acquiring the ranges at

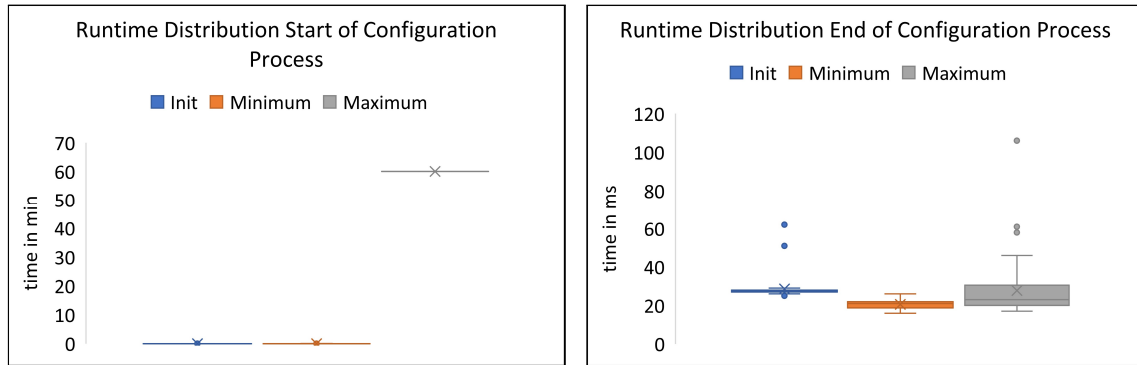


(a) At the start of the configuration process (b) At the end of the configuration process

Figure 7.6: Required time for each step of the attribute range computation using Z3 for our car feature model

the end of the configuration process took significantly less time. Additionally, the time needed for the initialization increased noticeably (from 0.028 seconds of the car model to 0.293 seconds as the median).

For the PC feature model, we come to similar conclusions as for the car. If a user is only interested in the minimum, computing the ranges is unnecessary and too costly. Every relation in the PC feature model is modeled as alternative. An Alternative-relation requires a larger propositional formula than an And- or Or-relation [BSRC10]. Additionally, the PC feature model has the highest amount of features. These two properties explain the longer time for the initialization.



(a) At the start of the configuration process (b) At the end of the configuration process

Figure 7.7: Required time for each step of the attribute range computation using Z3 for our PC feature model

Based on the results of this section, we now answer our research question *RQ1.2*. Especially, for the car and PC feature model, there is a huge difference between the time required to compute the minimum and maximum with the SMT solver. Therefore, the possibility of computing the range separately, leads to much shorter runtimes, when you are only interested in the minimum.

7.5 Evaluation of Precision

In this section, we analyze the quality of result given by our heuristic to find an answer for *RQ2.2*. We determine the quality by the percentage difference between the exact result given by the SMT solver and the approximated result.

Figure 7.8 shows the percentage difference between the exact results given by the SMT solver and the approximated values given by our heuristic for the bike feature model. For every measured configuration, the approximated values are equal to the exact values.

The bike feature model only contains tree constraints. Therefore, the results from this measurement support our claim that the heuristic is always correct for feature models without cross-tree constraints.

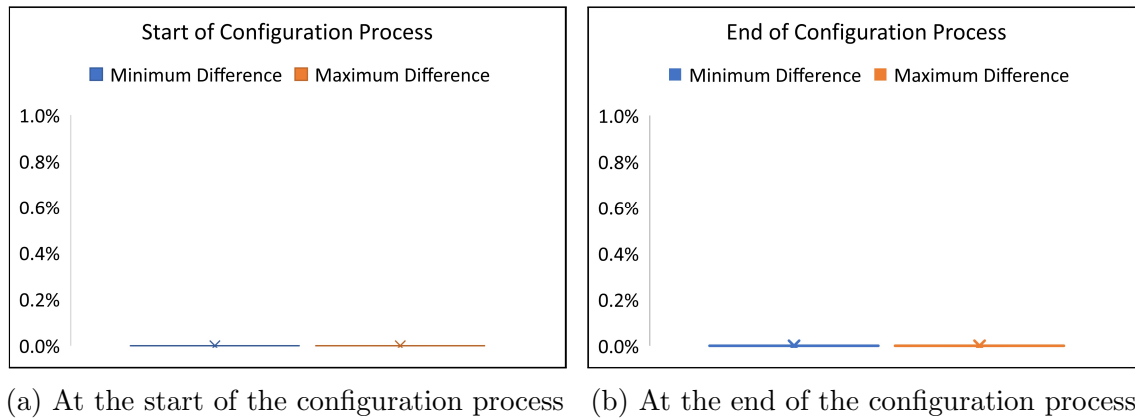
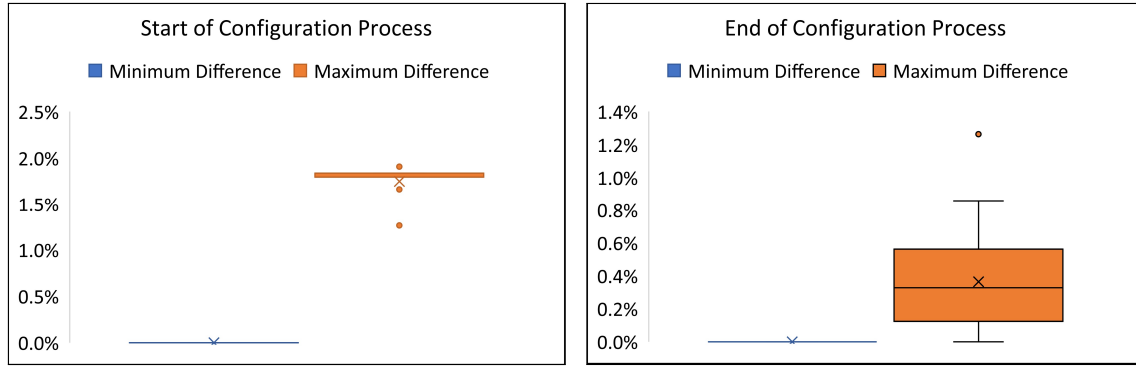


Figure 7.8: Percentage difference between exact and approximated values for our bike feature model

Figure 7.9 show the percentage difference between the exact results given by the SMT solver and the approximated values given by our heuristic for the car feature model. For every measured configuration, the exact and approximated minimum are equal. The percentage difference between maximal values given by the SMT solver and the heuristic never exceed 2%. In the maximal percentage difference the results were 50,774.60 for the exact value and 5,1762.60 for the approximation (ca. 1.91% difference in this case). Additionally, the approximated value for the maximum was exact in 6 out of the 65 measured configurations. Another thing to note for the computed maxima is that every approximated value is at least as large as the exact one.

The main conclusion of the measured percentage differences between minimum and maximum for the car feature model is that the approximated result are really close to the exact values. Therefore, the heuristic is usable for computations for this model. Additionally, the results support our claim in Section 4.4 that the approximation is conservative, as every approximated maximum and minimum is higher and smaller respectively than the exact counterpart.

The percentage difference between the results for the PC feature model given by the SMT solver and given by our heuristic is displayed in Figure 7.10. As we could not

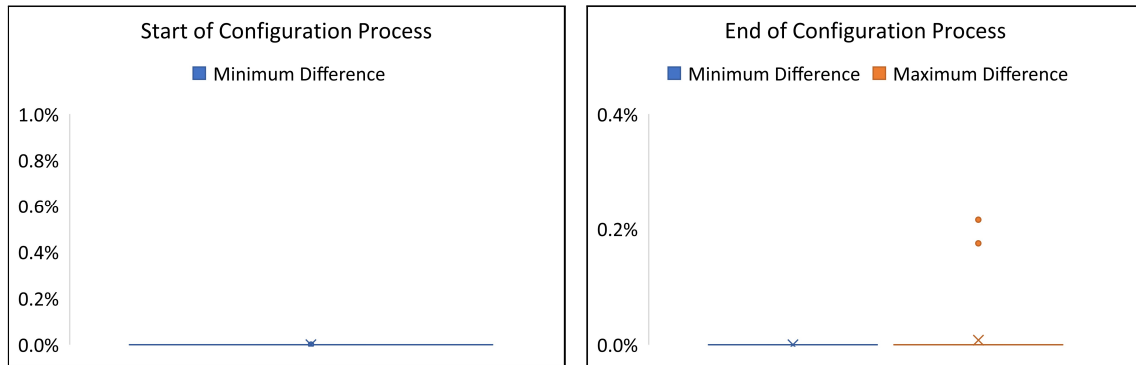


(a) At the end of the configuration process (b) At the start of the configuration process

Figure 7.9: Percentage difference between exact and approximated values for our car feature model

measure exact values for configurations with one feature selected of the PC feature model, only the percentage difference between exact and approximated ranges for random configuration is shown in 7.10b. For every measured configuration, the approximated minimum is equal to the exact one. The approximated maximums are exact in 48 out of 50 measured configurations. In the other two configurations the percentage differences were around 0.175% and 0.215%.

The measured differences were even lower than for the car feature model, as even the majority of maximum was exact. This follows from the lower amount of cross-tree constraints.



(a) At the start of the configuration process (b) At the end of the configuration process

Figure 7.10: Percentage difference between exact and approximated values for our PC feature model

With the measured results from this section, we answer the research question *RQ2.2*. For the evaluated feature models, our approximation is always very close to the exact value. Overall, even though approximating optimized attribute sums for other types of feature model may be off by a larger margin, the estimation algorithm is usable for quick results.

7.6 Summary

In this section, we summarize the conclusions we came to during this evaluation. In particular, we address our research questions given in [Section 7.1](#).

First, we discuss the usability of the given SMT solver Z3 for on-the-fly computations of attribute ranges ([RQ1.1](#)). Even though we only evaluated small feature models (54, 250, and 376 features), the two models with 250 and 376 features had a huge difference to our demanded time of one second required for one computation. For the PC feature model with 376 features and 12 constraints, we were not even able to acquire a single result within one hour. Therefore, we argue that the computation of attribute ranges using the SMT solver is not feasible for on-the-fly results. Additionally, we conclude that the approximation using our heuristic is necessary to ensure results on-the-fly.

Next, we analyze the time that is spent on the computation of the minimum and maximum separately ([RQ1.2](#)). With this question, we aim to answer whether its highly advantageous to compute only one side of a range, if you are only interested in one. There are two main factors to consider for this research question. First, the difference of time required for the computation between the minimum and maximum. Second, the necessary time for the initialization. Especially, for the car and PC feature models, we observed a high discrepancy between the time required to compute the maximum and minimum. In the PC feature model, we could not obtain a single result for the maximum within one hour of computing. However, the computation of the minimum never exceeded two seconds. The initialization only exceeded a runtime of one second in one case. Following from these observations, we conclude that if you are only interested in the minimum, computing it separately is advantageous for our models.

As we concluded that the SMT solver is not suitable for on-the-fly computations of attribute ranges and an approximation is required, we are now interested in the runtime of our heuristic to answer [RQ2.1](#). The measured runtimes for the heuristic show the usability, as only in 2 out of 195 cases the required time to compute approximated ranges exceeded 2 milliseconds (16 milliseconds in both outliers). Therefore, every computation of the heuristic is far below a runtime of one second.

For the next question ([RQ2.2](#)), we analyzed the difference between the approximated result given by our heuristic and the exact ones computed by the SMT solver. The goal of this is to determine the quality of our approximations. First, we observed that for every configuration our approximation was conservative, as every approximated maximum was at least as high as the actual and every approximated minimum was even equal to the exact minimum. Furthermore, the highest difference between an approximated and an exact maximum was ca. 1.91%. For the inclusion of less features, its less likely to violate a cross-tree constraint. This explains The closer proximity of the minimum results. Overall, we conclude that the results of our heuristic are so close to the actual ranges that the approximation is a good indicator for the minimum and maximum regarding our feature models.

8. Related Work

In this chapter, we compare works similar to our topic. First, we present similar publications and compare these to our result.

Comparison of SAT and SMT Solvers for Software Product Lines

The work from Michel et al. presents an approach to automate the configuration process for software product lines with the help of an SMT solver [MHGH12]. They created a process to translate a given feature diagram into the input of their SMT solver STP. STP is a state-of-the-art SMT solver that supports background theorems for bit-vectors and arrays. They translate their feature diagram using the background theorems for bit-vectors and arrays and compare the results between the SMT solver STP and CRYPTOMINISAT [Kul09], STP's default backend SAT solver. Unlike our work, Michel et al. translate their feature diagram into first-order logic using only the background theorems for bit-vectors and arrays. Our input into the SMT solvers is entirely in propositional logic for the automated analysis of feature models. Additionally, we compared separated solvers instead of comparing the SMT solver with its backend SAT solver. We concluded that the native SAT solver SAT4J is more efficient than every compared SMT solver from JAVASMT. In contrast, Michel et al. concluded from their relatively simple examples that STP is approximately twice as fast as CRYPTOMINISAT. If the examples get more complicated, they expect an even larger margin between both solvers.

Feature Model Analysis

Now, we discuss the work from Benavides et al. [BSRC10]. They present the state-of-the-art for the automated analysis of feature models by summarizing the various analyses that are available for them. In their work, they summarize 30 analyses for feature models. They categorize the proposals for the different analyses based on the notation of feature models. The proposals are grouped by basic, cardinality-based, and extended feature models. Additionally, the analyses are classified into propositional logic, constraint programming, description logic, and the category others. The analyses we evaluated during our thesis are only a small subset of all

analyses. They depend on propositional logic and we used SAT and SMT solvers to perform them. But SAT and SMT solvers are not the only tools for propositional logic based feature model analysis. Benavides et al. show that other tools like Alloy, BDD solver, and SMV can also be used. Furthermore, they give an overview about the tools used for the constraint programming based analyses.

Related Work regarding the Optimization of Feature Attributes

Benavides et al. also offered automated analyses on extended feature models, including the optimization of numerical attributes [BTRC05]. Their optimization takes an extended feature model and an objective function, which may include attribute values, as input. Then, the feature model is mapped into a constraint satisfaction problem. The implemented computation uses a CSP solver. Benavides et al.'s optimization differs from our result in several aspects. We support the configuration process by computing the ranges for partial configurations, instead of optimizing once for the feature model itself. Additionally, we offer an approximation to obtain results on-the-fly. Furthermore, instead of using CSP, we use SMT to optimize feature attributes.

White et al. optimized resources in feature models, which work similar to feature attributes [WDS09]. Instead of acquiring exact results their implementation used filtered cartesian flattening, which approximates the maximized or minimized value of a resource in polynomial time. Additionally, constraints on other resources are possible. For example, it is possible to maximize variable A while B can be at most 100. The main difference between White et al.'s and our [WDS09] is that we provide support during the configuration process. Additionally, we offer flexible switching between exact and approximated results. *[results vergleiche]*

9. Conclusion

In this thesis, we wanted to improve the interactive configuration process by calculating ranges for feature model attributes. The calculation of these ranges is not possible for SAT solvers because it is currently impossible to express numerical attributes with propositional logic. Therefore, we decided to use an SMT solver which is capable of optimizing numerical variables. Furthermore, we were interested whether SMT solvers are superior to SAT solvers regarding the automated analysis of feature models.

We created the abstract data type `IncrementalSolver` and used it to formally define analyses for feature model inconsistencies to compare SAT with SMT solvers. By proposing multiple enhancements for some of the analyses, we wanted to optimize their performance. Also, with the help of the abstract data type, we formally defined the analysis of finding explanations for feature model defects to further compare both kinds of solvers.

The evaluation for the comparison of SAT and SMT solvers shows, that SAT solvers are more efficient than SMT solvers for every analysis except for the redundant constraint analysis. Furthermore, SMT solvers are more efficient than SAT solvers at finding explanations for feature model defects. Additionally, we concluded that three out of four optimizations greatly improve the performance of the respective analysis.

We realized the computation of attribute ranges for partial configurations with an SMT solver, by creating a first-order logic formula, representing the configuration and the sum of the attributes values. Furthermore, we implemented an heuristic which conservatively approximates the ranges in linear time.

The evaluation for the computation of attribute ranges shows that using an SMT solver is feasible, but not realistic for the support of the interactive configuration process, as the runtime increases immensely with the amount of features. We could not acquire a single result within one hour for a feature model with 367 features. However, for every evaluated request the heuristic immediately provided results, which differed at most 2% from the exact values.

Based on our results, we determine the following recommendations regarding SMT solvers. SMT solvers cannot replace SAT solvers for the automated analysis of feature models. However, they can significantly improve the finding of explanations for any kinds of defects. Finally, we do not recommend the usage of SMT solvers to compute attribute ranges for partial configurations. Instead we advocate to approximate the attribute ranges with our heuristic.

10. Future Work

In this chapter, we present possible future work regarding the topic of our thesis. First, we introduce cross-tree constraints including feature attributes. Second, we propose future work for the automated analysis of feature models. Third, we speak about adding more solvers to the implementation to compare them. Fourth, we present a method to find the minimal unsatisfiable core using an SMT solver. Fifth, we discuss possible improvements of the interactive configuration process. Last but not least, we propose further optimizations for the range computation.

Feature Attributes

Cross-tree constraints describe relations between feature which are not directly connected. In future, one might extend the cross-tree constraints for extended feature models to include the attributes from features. This would greatly benefit the variability of the extended feature model but would also increase the effort to analyze it. Additionally, more expressive solvers than SAT solvers are required.

Feature Model Analysis

As part of our thesis, we introduced, implemented, and evaluated analyses for various defects of feature models. The introduced analyses are not the only analyses used for feature models. Benavides et al. present the state-of-the-art for the automated analysis of feature models by summarizing the various analyses that are available for feature models [BSRC10]. Our implemented analyses only cover a small set of the analyses which are currently available. Hence, in future one might extend our implementation with the missing analyses.

One of the challenges of Benavides et al. work was to create a framework to describe the operations of all analyses formally [BSRC10]. We realized the framework by creating and implementing the abstract data type `IncrementalSolver`. With the help of the `IncrementalSolver`, we already defined the analyses for the feature model defects formally. We can further define the missing analyses for the automated

feature model analysis. Hence, future work is to extend the abstract data type `IncrementalSolver`, if needed, and to formally define the missing analyses.

In our evaluation, we detected some unoptimized analyses for certain solvers. We saw that the optimizations cannot be performed for the SMT solvers because the general analyses require a general data structure, which is natively not provided by SMT solvers. In future one might create analyses that natively work with the data structure provided by the SMT solvers to omit the conversion from the `JAVASMT` data structure to the general data structure. Additionally, we saw that removing clauses from the SAT solver is very slow. Kanning faces the same problem for SAT4J in his work and customizes the solvers to optimize the removal for clauses [Kan17]. If we apply his customization into our implementation, we could further improve the performance for SAT4J.

Comparison of SMT and SAT

During this thesis, we implemented the SAT solver SAT4J and the solver API `JAVASMT` and compared all solvers on the task of the automated analysis and on the task of finding their explanations. The implemented solvers are not the only solvers available. Future work is to add more SAT and SMT solvers to our implementation and to compare the solvers with the `IncrementalSolver` on the different analyses.

Unsatisfiable Core

With the current state of our implementation, it is possible to retrieve the unsatisfiable core of a formula. The unsatisfiable core is used as an explanation to help the user fixing a defect. The explanation can be understood more easily if the unsatisfiable core is minimal. Hence, it is desired to retrieve the minimal unsatisfiable core. SAT4J can retrieve all minimal unsatisfiable cores. In contrast, SMT solvers can also retrieve the unsatisfiable core, but generally, the computed unsatisfiable core is not minimal. Therefore, we present the work of Guthmann et al., which shows an algorithm for SMT solvers to extract a minimal unsatisfiable core of an unsatisfiable formula [GST16]. Their algorithm is based on a well known deletion-based minimal unsatisfiable cores extraction, which is also widely used for propositional based minimal unsatisfiable cores extraction. Theory-rotation and several other optimizations further enhance the algorithm. In future one might extend our implementation with the algorithm of Guthmann et al. to retrieve minimal unsatisfiable cores for the SMT solvers.

Computing Attribute Ranges

The validity of computing the attribute ranges with the Z3 solver and our heuristic has to be inspected further. During this thesis, our evaluation focused on feature models re-engineered from publicly available product configurators. However, computing attribute ranges for other types of extended feature models might be interesting as well. Even though our implementation supports all types of extended feature models, further evaluation is necessary.

Another interesting research topic are other SMT solvers. During this thesis, we exclusively used the Z3 solver. Other SMT based solvers or APIs might result in a

better performance when computing attribute ranges. Therefore, a comparison of different solvers and APIs is a relevant research topic.

Another direction for future work might be improving the methods we implemented. First, optimizing the first-order logic formula used for computing the attribute ranges with SMT might improve the performance of this approach. On the other hand, the quality of the results approximated by the estimations algorithm could be improved. For example, the algorithm could account simple cross-tree constraints like an implication between two features. This would improve the approximations in several cases.

Support for the Interactive Configuration Process

One possibility to improve the configuration process might be to acquire sets of features whose attribute values build the optimized sum. These sets could be used to automatically build configurations. Using such a functionality enables the user not only to see the resulting minimum and maximum from a selected partial configuration, but also to build this product.

Additionally, more specific computations of attribute ranges could be implemented. For example, the possibility of adding further constraints may be interesting (e.g. optimizing a price of a sandwich while not exceeding a certain amount of calories).

A. Appendix

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <featureModel>
3   <properties />
4   <struct>
5     <and abstract="true" mandatory="true" name="ExampleFeatureModel">
6       <feature name="Base" />
7     </and>
8   </struct>
9   <constraints />
10  <calculations Auto="true" Constraints="true" Features="true" Redundant="true"
    Tautology="true" />
11  <comments />
12  <featureOrder userDefined="false" />
13 </featureModel>

```

Listing A.1: XML file of a feature model with the root feature *ExampleFeatureModel* and the child *Base*

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <extendedFeatureModel>
3   <properties />
4   <struct>
5     <and abstract="true" mandatory="true" name="ExampleFeatureModel">
6       <attribute name="Attribute0" type="string" unit="" value="Example" />
7       <feature name="Base" />
8     </and>
9   </struct>
10  <constraints />
11  <calculations Auto="true" Constraints="true" Features="true" Redundant="true"
    Tautology="true" />
12  <comments />
13  <featureOrder userDefined="false" />
14 </extendedFeatureModel>

```

Listing A.2: XML file of an extended feature model with the root feature *ExampleFeatureModel* and the child *Base*. The root feature also has the attribute *Attribute0* assigned

Name	Features	Constraints
aaed2000	1298	904
adder	1286	890

adderII	1289	890
aeb	1226	875
aim711	1277	912
aki3068net	1220	851
am31_sim	1178	819
asb	1244	851
asb2305	1255	857
assabet	1279	927
at91sam7sek	1309	917
at91sam7xek	1332	928
atlas_mips32_4kc	1229	859
atlas_mips64_5kc	1222	857
brutus	1234	876
calm16_ceb	1186	830
calm32_ceb	1186	830
ceb_v850	1202	843
cerf	1289	935
cerfpda	1304	946
cma230	1225	868
cma28x	1217	849
cme555	1279	863
cq7708	1252	858
cq7750	1281	864
csb281	1246	871
dreamcast	1266	867
e7t	1280	883
ea2468	1408	956
eb40	1253	890
eb40a	1254	890
eb42	1247	884
eb55	1283	914
ebsa285	1258	895
ec555	1280	863
edb7xxx	1259	900
edosk2674	1207	841
excalibur_arm9	1247	877
fads	1200	829
flexanet	1278	925
frv400	1251	862
gps4020	1226	876
grg	1254	889
h8300h_sim	1195	828
h8max	1215	849
h8s_sim	1196	828
hs7729pci	1312	886
innovator	1270	886
integrator_arm7	1272	904

integrator_arm9	1281	907
ipaq	1271	921
iq80310	1271	897
iq80321	1269	898
ixdp425	1260	886
jmr3904	1212	843
jtst	1254	891
linux	1245	859
lpcmt	1262	882
m5272c3	1336	877
mac7100evb	1247	867
mace1	1247	867
malta_mips32_4kc	1245	865
malta_mips64_5kc	1245	865
mb93091	1263	876
mb93093	1243	859
mbx	1308	889
mcb2100	1262	882
moab	1291	917
mpc50	1226	863
nano	1278	911
npwr	1271	898
ocelot	1279	872
olpce2294	1287	904
olpch2294	1274	890
olpcl2294	1286	902
p2106	1262	882
pati	1261	866
pc_i82544	1272	885
pc_i82559	1272	885
pc_rltk8139	1270	886
pc_usb_d12	1294	910
pc_vmWare	1268	884
phycore	1287	892
phycore229x	1373	932
picasso	1261	899
pid	1242	891
prpmc1100	1236	875
psim	1194	826
rattler	1324	894
ref4955	1231	862
refidt334	1276	870
sa1100mm	1235	878
sam7ex256	1345	935
se7751	1309	883
se77x9	1333	911
sh4_202_md	1272	885

sh7708	1267	876
skmb91302	1230	840
sleb	1201	827
smdk2410	1279	884
snds	1231	872
sparc_erc32	1181	816
sparc_leon	1181	816
sparc-lite_sim	1181	816
stb	1265	860
stdevall	1203	844
stm3210e_eval	1283	873
ts1000	1310	886
ts6	1253	865
tx39_sim	1192	822
uE250	1262	900
vads	1253	865
viper	1314	894
vrc4373	1260	860
vrc4375	1271	870
XSEngine	1273	886
Automotive2_V1	14010	666
Automotive2_V2	17742	914
Automotive2_V3	18434	1300
Automotive2_V4	18616	1369

Table A.1: List containing the name, number of features, and number of constraints for all 120 feature models that were evaluated for the comparison of SAT and SMT solvers

Bibliography

- [ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. 2013. (cited on Page 1, 5, 6, 13, and 14)
- [Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 7–20, 2005. (cited on Page 1)
- [Ber] Daniel Le Berre. SAT4J: A Satisfiability Library for Java. Website. Available online at <http://www.sat4j.org/>; visited on November 9th, 2010. (cited on Page 40)
- [BRCTS06] David Benavides, Antonio Ruiz-Cortés, Pablo Trinidad, and Sergio Segura. A Survey on the Automated Analyses of Feature Models. *Jornadas de Ingeniera del Software y Bases de Datos*, pages 367–376, 2006. (cited on Page 15)
- [BRN⁺13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A Survey of Variability Modeling in Industrial Practice. pages 7:1–7:8, 2013. (cited on Page 1 and 40)
- [BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–708, 2010. (cited on Page 1, 8, 13, 14, 17, 18, 19, 21, 22, 93, 97, and 101)
- [BSS⁺09] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, Cesare Tinelli, et al. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009. (cited on Page 2 and 11)
- [BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *International Conference on Advanced Information Systems Engineering*, pages 491–503. Springer, 2005. (cited on Page 98)
- [CESS08] Koen Claessen, Niklas Een, Mary Sheeran, and Niklas Sörensson. Satisfiability in practice. In *Proceedings of the International Workshop on Discrete Event Systems (WODES)*, pages 61–67. IEEE, 2008. (cited on Page 10)

- [CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration through Specialization and Multi-Level Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005. (cited on Page 1)
- [CSHL13] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Beyond boolean product-line model checking: dealing with feature attributes and multi-features. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 472–481. IEEE Press, 2013. (cited on Page 8)
- [DHN06] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 36–41. Springer, 2006. (cited on Page 11)
- [DMB11] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011. (cited on Page 11 and 15)
- [EPAH09] Abdelrahman Osman Elfaki, Somnuk Phon-Amnuaisuk, and Chin Kuan Ho. Using first order logic to validate feature model. 2009. (cited on Page 1)
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. (cited on Page 46)
- [GST16] Ofer Guthmann, Ofer Strichman, and Anna Trostanetski. Minimal unsatisfiable core extraction for smt. In *Formal Methods in Computer-Aided Design (FMCAD), 2016*, pages 57–64. IEEE, 2016. (cited on Page 102)
- [Gün17] Timo Günther. Explaining Satisfiability Queries for Software Product Lines. Master’s thesis, Technische Universität Braunschweig, 2017. (cited on Page 14, 21, 44, and 45)
- [Jan10] Mikoláš Janota. *SAT solving in interactive configuration*. PhD thesis, Citeseer, 2010. (cited on Page 17, 18, 19, 20, and 24)
- [Kan17] Frederik Kanning. Presence Condition Reasoning with Feature Model Interfaces. Master’s thesis, Technische Universität Braunschweig, 2017. (cited on Page 77 and 102)
- [KAT16a] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. Explaining Anomalies in Feature Models. pages 132–143, 2016. (cited on Page vii, 13, 18, and 19)
- [KAT16b] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. Explaining Anomalies in Feature Models. Technical Report 2016-01, TU Braunschweig, Germany, August 2016. (cited on Page 17, 18, 19, and 21)

- [KFB16] Egor George Karpenkov, Karlheinz Friedberger, and Dirk Beyer. Javasmmt: A unified interface for smt solvers in java. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 139–148. Springer, 2016. (cited on Page xi, 40, and 41)
- [KK01] Andreas Kaiser and Wolfgang Küchlin. Detecting inadmissible and necessary variables in large propositional formulae. In *University of Siena*. Citeseer, 2001. (cited on Page 17, 18, 19, 20, and 24)
- [KOD13] Ahmet Serkan Karataş, Halit Oğuztüzün, and Ali Doğru. From extended feature models to constraint logic programming. *Science of Computer Programming*, 78(12):2295–2312, 2013. (cited on Page 5 and 8)
- [KPK⁺17] Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. Featureide: Empowering third-party developers. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume B*, pages 42–45. ACM, 2017. (cited on Page 46)
- [KTM⁺17a] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. Is there a mismatch between real-world feature models and product-line research? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 291–302. ACM, 2017. (cited on Page 5)
- [KTM⁺17b] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. Is There a Mismatch Between Real-World Feature Models and Product-Line Research? In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESECFSE)*, 2017. To appear. (cited on Page 70)
- [Kul09] Oliver Kullmann. Theory and applications of satisfiability testing. *SAT*, 2009. (cited on Page 97)
- [Man02] Mike Mannion. Using First-Order Logic for Product Line Model Validation. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 176–187, 2002. (cited on Page 15)
- [MHGH12] Raphaël Michel, Arnaud Hubaux, Vijay Ganesh, and Patrick Heymans. An smt-based approach to automated configuration. *SMT@ IJCAR*, 20:109–119, 2012. (cited on Page 97)
- [MTS⁺17] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. 2017. To appear. (cited on Page 3 and 14)
- [MWC09] Marcílio Mendonça, Andrzej Wąsowski, and Krzysztof Czarnecki. SAT-Based Analysis of Feature Models is Easy. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 231–240, 2009. (cited on Page 1)

-
- [RMH12] Vijay Ganesh Raphael Michel, Arnaud Hubaux and Patrick Heymans. An SMT-based Approach to Automated Configuration. *SMT@ IJCAR*, pages 109–119, 2012. To appear. (cited on Page 2)
- [TBK09] Thomas Thum, Don Batory, and Christian Kastner. Reasoning about edits to feature models. In *Proceedings of the 31st International Conference on Software Engineering*, pages 254–264. IEEE Computer Society, 2009. (cited on Page 31)
- [vdML04] Thomas von der Maßen and Horst Lichter. Deficiencies in feature models. In *Workshop on Software Variability Management for Product Derivation-Towards Tool Support*, volume 44, 2004. (cited on Page 21)
- [WDS09] Jules White, Brian Dougherty, and Douglas C Schmidt. Selecting highly optimal architectural feature sets with filtered cartesian flattening. *Journal of Systems and Software*, 82(8):1268–1284, 2009. (cited on Page 8 and 98)

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

Braunschweig, den 03. April 2018

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

Braunschweig, den 03. April 2018