TU Braunschweig

Department of Computer Science

Master's Thesis

Explaining Defects and Identifying Dependencies in Interrelated Feature Models

Author:

Sofia Ananieva

September 01, 2016

Advisors:

Prof. Dr.-Ing. Ina Schaefer

Institute of Software Engineering and Automotive Informatics

M.Sc. Matthias Kowal & Dr.-Ing. Thomas Thüm

Institute of Software Engineering and Automotive Informatics

http://publikationsserver.tu-braunschweig.de/get/64218

Ananieva, Sofia:
Explaining Defects and Identifying Dependencies in Interrelated Feature Models
Master's Thesis, TU Braunschweig, 2016.

http://publikationsserver.tu-braunschweig.de/get/64218

Acknowledgements

Main thanks are due to my supervisors, Dr.-Ing. Thomas Thüm and M.Sc. Matthias Kowal, for giving me the opportunity to work on this thesis. With Thomas and Matthias, I had two very dedicated supervisors who always took the time to discuss questions and carefully red the drafts of this work. I greatly appreciated their confidence, our productive meetings and the scientific freedom to conduct this work.
Finally, I am very thankful to my family for their constant support.

http://publikationsserver.tu-braunschweig.de/get/64218

http://publikationsserver.tu-braunschweig.de/get/64218

Abstract

Software product line engineering has proven to be a successful develop approach for variable software in different industrial applications, especially in the automotive area. The variability of a product line is at the center of software product lines engineering. By modeling variability on a high abstraction level, software product lines enable a wide variety of a product. Nevertheless, modeling variability is not a trivial task. It gets more complex with a growing number of variants and, hence, becomes more error-prone. For instance, modeling errors may result in product configurations being not possible anymore. The detection of such defects is well researched. A user-friendly explanation of the defect causes - the subject of the present work - is still a challenge and is becoming increasingly important.
In the scope of this thesis, we elaborate a generic algorithm which is able to generate explanations for any kind of modeling defects based on predicate logic. A resulting explanation is generated in a user-friendly and structural manner and displayed within a tool tip during the modeling phase.
Additionally, the basic algorithm is improved to find the shortest explanation und to compute and visualize relevant parts of the explanation. Furthermore, we detect hidden dependencies among interrelated product lines and apply the generic explanation algorithm mentioned above to explain such dependencies.
In a quantitative and qualitative analysis, we evaluate the explanation algorithm and resulting explanations concerning their correctness, understandability, performance impact and length using existing examples. For hidden dependencies, we additionally inspect situations in which such dependencies occur most often. By analyzing respective explanations, we can furthermore determine the number of involved product lines in a hidden dependency.
To summarize the results, we demonstrate the correctness and understandability of explanations and show the scalability of the explanation algorithm for different sizes of product lines. Generating a first explanation approximately doubles the computational time of the former model analysis while the improved algorithm (which searches for a shortest explanation) approximately triples the computational time. It is notable that the first explanation is most often already the shortest one, otherwise it is usually shorter by 25% - 50%. Explanation length slightly increases compared to the size of a product line. Finally, we observe that up to five interrelated feature models may lead to a hidden dependency based on the evaluated product lines.

http://publikationsserver.tu-braunschweig.de/get/64218

http://publikationsserver.tu-braunschweig.de/get/64218

Zusammenfassung

Software-Produktlinien gelten als anerkanntes Entwicklungskonzept für variable Software in vielen industriellen Anwendungsbereichen, insbesondere im Automobilbereich. Im Zentrum von Software-Produktlinien steht die Variabilität einer Produktlinie, die eine Variantenvielfalt des Produktes durch die Modellierung auf einer hohen Abstraktionsebene ermöglicht. Jedoch ist die Modellierung von Variabilität eine nicht triviale Aufgabe, die mit der wachsenden Anzahl der Varianten komplexer und dadurch fehleranfälliger wird. Modellierungsfehler können beispielsweise dazu führen, dass Produktkonfigurationen nicht mehr möglich sind. Die Detektion von Modellierungsfehlern einer Softwareproduktlinie ist ein weit verbreitetes Forschungsfeld. Die nutzerfreundliche Erklärung von Fehlerursachen - das Thema der vorliegenden Arbeit - stellt dabei weiterhin eine Herausforderung dar und wird immer wichtiger.
Im Rahmen dieser Arbeit wird ein generischer Algorithmus ausgearbeitet, um Modellierungsfehler aller Arten auf Basis der Prädikatenlogik zu erklären. Die resultierende Erklärung wird in der natürlichen Sprache nutzerfreundlich und strukturiert generiert und als Tooltip während der Modellierung angezeigt. Der Basisalgorithmus wird zusätzlich erweitert, um kürzeste Erklärungen zu finden und die Relevanz der einzelnen Teile einer Erklärung zu berechnen und visuell hervorzuheben.
Weiterhin detektieren wir versteckte Abhängigkeiten in zusammenhängenden Produktlinien und verwenden den oben erwähnten generischen Algorithmus zur Erklärung dieser Abhängigkeiten. In einer qualitativen und quantitativen Analyse evaluieren wir den Algorithmus und resultierende Erklärungen anhand der vorhandenen Beispiele im Hinblick auf ihre Korrektheit, Verständlichkeit, Performance und Länge.
Für versteckte Abhängigkeiten untersuchen wir zusätzlich, in welchen typischen Situationen diese Abhängigkeiten am häufigsten entstehen. Durch die Inspektion von Erklärungen können wir darüber hinaus feststellen, wie viele zusammenhängende Produktlinien in einer versteckten Abhängigkeit involviert sind.
Zusammenfassend stellen wir fest, dass Erklärungen korrekt und verständlich sind und die Generierung von Erklärungen für unterschiedlich große Produktlinien skaliert. Die Generierung der Erklärung mit dem Basisalgorithmus verdoppelt dabei ungefähr die Rechenzeit der ursprünglichen Modellanalyse, während der erweiterte Algorithmus (welcher nach der kürzesten Erklärung sucht) die Rechenzeit in etwa verdreifacht. Auffällig ist, dass die erste Erklärung meistens die Kürzeste ist, andernfalls ist die kürzeste Erklärung üblicherweise um 25% - 50% kleiner. Die Erklärungslänge wächst nur geringfügig im Vergleich zu der Größe einer Produktlinie. Zum Schluss zeigen wir, dass

http://publikationsserver.tu-braunschweig.de/get/64218

anhand der analysierten Produktlinien bis zu fünf zusammenhängende Produktlinien zu einer versteckten Abhängigkeit führen können.

http://publikationsserver.tu-braunschweig.de/get/64218

Contents

http://publikationsserver.tu-braunschweig.de/get/64218

http://publikationsserver.tu-braunschweig.de/get/64218

List of Figures

http://publikationsserver.tu-braunschweig.de/get/64218

http://publikationsserver.tu-braunschweig.de/get/64218

http://publikationsserver.tu-braunschweig.de/get/64218

http://publikationsserver.tu-braunschweig.de/get/64218

List of Tables

http://publikationsserver.tu-braunschweig.de/get/64218

http://publikationsserver.tu-braunschweig.de/get/64218

1. Introduction

With a growing interest in variant diversity for industrial products, variability has become a key characteristic to many software systems. It enables a system to adapt to different environments or customer requirements and, hence, allowing fully customized product variants. To manage variability, the paradigm of *software product lines* (SPL) has been introduced [16]. SPLs make use of a common base platform for all product variants. Reusing parts of the base platform enables a reduced time-to-market and improves the cost-efficient distribution of individualized products compared to classic development methods.

1.1 Motivation

A wide-spread approach to model variability in SPLs are feature models [10]. In a feature model, features represent permanent or variable product characteristics. Constraints express relationships among features such as the selection of one feature may require or exclude the selection of another feature. Nevertheless, feature models might become a subject to uncontrolled software evolution due to constantly changing requirements and dependencies between features [49]. This can result in erroneous (defect) feature models. V.d. Maßen et al. differ between three types of such defects namely redundancy, anomaly and inconsistency [67]. A feature model contains redundancy, if semantic information is modeled in multiple ways. Anomalies appear, if a feature model represents an incorrectly modeled domain, e.g., a permanent feature excludes another feature, thus making it impossible to select. Finally, inconsistencies appear in a feature model, if semantic information is modeled contradictory.

In order to avoid the described defects, an adequate tool support is needed to assist the user during the development and maintenance of a feature model. In literature, this process is called *automated analysis of feature models* and represents one of the

http://publikationsserver.tu-braunschweig.de/get/64218

current challenges in SPLs [9, 10]. An important part of automated analysis operations are *explanations*, which are usually related to defects in the feature model and make the occurrence of a certain defect comprehensible for the user [10]. Explanations are most commonly expressed in terms of features and relationships among them. In order to help the modeler find the source of a defect, explanations should be as detailed and understandable as possible [9].

Although several tools for feature modeling manage to detect defects, e.g., FeatureIDE [63] or pure::variants[1], the generation of explanations suffers from several limitations: existing explanation algorithms are either not open-source [40], roughly described in literature [7] or are disadvantageous in terms of explanation length, scalability, understandability and evaluation [10, 25, 55, 66]. Furthermore, explanations for redundant cross-tree constraints are often completely missing [7, 40, 55].

An additional use case for explanations emerges from explaining implicit constraints in feature models. An implicit constraint is a hidden dependency between features, which occurs in interrelated feature models. Such feature models are connected by cross-tree constraints, while every feature model might represent a subset of the large model or depict a different view on the SPL, e.g., mechanical, electrical and software components of a product. With growing feature models consisting of thousands of features and constraints, splitting large models into a set of interrelated submodels reduces the complexity of an SPL. To assure the quality of interrelated feature models, the automated detection and explanation of hidden dependencies becomes even more important.

1.2 Goals

With the limitations concerning the explanation of defects and hidden dependencies in feature models mentioned above in mind, we aim at achieving two overall **research goals (RG)**:

RG1 Developing a generic algorithm, which provides explanations for redundant cross-tree constraints within a feature model. Explanations may be also generated for further defects. Explanations must be informative with respect to size and completeness. A tool tip containing an explanation for a selected defect should be provided.

RG2 Explain implicit constraints in interrelated feature models by reusing the explanation approach from **RG1**.

The thesis must include a critical assessment of the developed methods. As proof of concept, the presented feature models by Feldmann et al. along with several test feature models should be used to assess the applicability of both methods [21] .

[1] https://www.pure-systems.com/

http://publikationsserver.tu-braunschweig.de/get/64218

1.3 Contribution

Realizing **RG1** includes the following contributions:

RG1.1 *Developing a generic explanation algorithm.*
We develop a generic algorithm, which is able to explain all defects in a feature model.

RG1.2 *Providing an open-source implementation.*
We provide an open-source implementation of the explaining algorithm within FeatureIDE. The implementation is publicly available on GitHub.

RG1.3 *Refine the explanation algorithm to increase the quality of explanations.*
We aim to find shortest explanations, express explanations in a user-friendly manner and emphasize most relevant parts leading to a defect.

RG1.4 *Evaluating the scalability of the generic algorithm.*
We show the scalability of the explanation algorithm by applying it on industrial-sized feature models.

Realizing **RG2** leads to the following contributions:

RG2.1 *Detecting implicit constraints for a feature model.*
We develop an approach, which detects and presents implicit constraints for a subtree of a feature model.

RG2.2 *Explaining implicit constraints.*
Using the explanation approach from **RG1**, we compose and visualize an explanation for implicit constraints in order to understand hidden dependencies.

RG2.3 *Providing an open-source implementation.*
We provide an open-source implementation of the explaining algorithm within FeatureIDE. The implementation is publicly available on GitHub.

To realize both research goals, the following steps are performed iteratively for **RG1** and **RG2**:

1. Literature research:
   Collect knowledge in the context of automated analysis, detection and explanation of defects.

2. Concept development:
   In the first phase, a generic explanation algorithm will be worked out in order to find causes for redundant cross-tree constraints and provide respective explanations. The algorithm will be applicable to explain further defects. In the second phase, a conceptual framework will be developed to detect and explain implicit constraints in interrelated feature models.

http://publikationsserver.tu-braunschweig.de/get/64218

3. Prototypical implementation:
   The implementation of an explanation algorithm is supported by using existing tools, i.e., the Eclipse Modeling Framework and FeatureIDE [63]. FeatureIDE is an Eclipse-based IDE and supports single development steps of SPL, e.g., domain analysis and domain implementation. In the first phase, FeatureIDE will be extended to display explanations of defects to the user designing feature models.
   In the second phase, FeatureIDE will be extended with the functionality to present and explain hidden dependencies in a feature model.

4. Conducting a case study:
   To explore the applicability of the developed explanation approach, multiple test models and interrelated feature models from Feldmann et al. will be used [21]. If redundant cross-tree constraints or implicit constraints are missing, the feature models will be artificially extended.

1.4 Structure

The remainder of this thesis is structured as follows.

In Chapter 2, we present relevant background information on SPLs in general. This includes feature models, their mapping to propositional formulas and an overview about interrelated feature models. Next, we describe common defects in feature models using suitable examples. The section ends with an overview about automated analysis of feature models and the state-of-the-art of explanations.
Chapter 3 deals with the theoretical background of explanations. We introduce the basics of an explanation algorithm, which meets the specified requirements. Next, we discuss the adaption of the algorithm for defects and hidden dependencies in a feature model. The chapter ends with improvements of the algorithm.
Chapter 4 covers the workflow and architecture of the implementation. Additionally, we provide information about the availability of the source code.
Chapter 5 presents an evaluation of the generic explanation approach, divided into a quantitative and a quantitative analysis of the explanation algorithm and resulting explanations. In this chapter, we present and discuss the results of the evaluation.
In Chapter 6, we give an overview about related work.
Chapter 7 deals with a brief conclusion on this thesis and summarized findings.
Eventually, we present future work in Chapter 8.

http://publikationsserver.tu-braunschweig.de/get/64218

2. Background

The subject of this thesis is the generation of explanations for defects which occur in a feature model. Therefore, we provide relevant background information in this chapter. In Section 2.1, we give an overview about fundamental aspects of SPLs including feature models and their mapping to propositional formulas. In Section 2.2, we explain different kinds of defects in a feature model and give small examples for every defect. In Section 2.3, we refer to the automated analysis of feature models and give an overview about the detect of defects in a feature model and their explanation.

2.1 Software Product Lines

Software product line engineering has established itself as a successful approach for reusability and expandability of software. The idea of software product line engineering was inspired by the *individualized mass production* back in the 70's, which enabled a cost-efficient and faster production due to *common platforms* [52]. Such platforms represented commonalities between all product variants, while further development added distinguishing features to every product variant. This development paradigm was successfully adapted by the automotive industry, e.g., a common platform for different vehicles comprised the chassis and components of a car body. A recent example covers the Volkswagen Group MQB platform (*Modularer Querbaukasten* in German). The MQB platform represents a construction system for automobiles with standardized and transverse front-engines [1]. Key advantages of a common platform are reduced production costs, a rapid time-to-market and a wide variety of products. In the automotive industry, a *product line* represented a series of vehicles, e.g., a VW Golf VII and VW Golf Sportsvan, which both are based on the MQB platform.

Software Systems get increasingly more complex nowadays, which is why software engineers applied the idea of product lines to software development. This has replaced a

[1] http://www.volkswagenag.com/content/vwcorp/info_center/en/themes/2012/02/MQB.html

http://publikationsserver.tu-braunschweig.de/get/64218

conventional software development focusing on producing software for single customers or a certain purpose. The main idea of SPLs enables software manufacturers to tailor products according to the needs of customers while reusing common and variable parts for generating a software product [4]. The Institute of Software Engineering at the Carnegie Mellon University provides a common definition of SPLs [16]:

"A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."

The definition covers software-intensive systems, i.e., hardware systems whose software component mainly drives the characteristics of a system encompassing similar, yet distinguishable software products based on *features*. A feature represents a characteristic of a software system, which can be adapted to changing customer needs and environments and which is visible to customers [4].
Figure 2.1 illustrates an SPL with three exemplary product variants. A common, managed set of features involves *left button, right button, scroll wheel, ball, scanner, USB, bluetooth*. The first product comprises a left and right button, an USB connection and a ball, which is used to register mouse movement. Based on the first product, a second product is enhanced with a scroll wheel between the buttons and a scanner instead of a ball to process mouse movement. A third product also uses a scanner and a bluetooth connection instead of USB. Core assets are available in every product configuration and provide basic functionality, e.g., the left button and the right button. An example for the reusability of variable parts in an SPL is the scroll wheel feature. It extends the SPL by an optional functionality and is used in the second and third product. Hence, all products are similar yet different and based on a common, managed set of features.

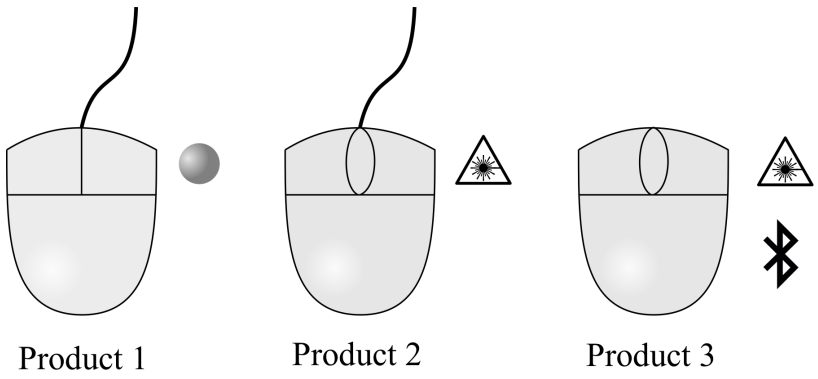

Figure 2.1: Mouse example SPL

During the development of an SPL, diversity and commonality of the artifacts represent a key factor. Differences are realized by so called *variation points*, which represent configuration options that are specified by a user. Hence, products of an SPL differ with respect to decisions made by a user at these points. In order to define commonalities and differences of a product line, software product line development is divided into two subsequent processes: *Domain Engineering* and *Application Engineering* [53].

http://publikationsserver.tu-braunschweig.de/get/64218

- The domain engineering process focuses on modeling an SPL in order to fulfill requirements of multiple customers in a domain. It is performed once for an SPL and covers the definition of strategic goals, requirements, architecture modeling and developing and testing reusable code. A core task of domain engineering is the creation of a *feature model*, which comprises characteristics of a product line, dependencies and configuration options (cf. Section 2.1.1)

- The application engineering process is based on the domain engineering. It comprises the definition of concrete applications, i.e., product variants, which are tailored to requirements of a single customer. The derivation of concrete instances includes a refinement of requirements, architecture and code.

Czarnecki and Eisenecker distinguish between *problem space* and *solution space* in SPL engineering [19]. While the problem space represents domain-specific artifacts, e.g., requirements, the solution space comprises implementation-oriented artifacts. In order to achieve a correct variant handling, artifacts of both spaces and engineering processes are linked. For instance, changes in a domain artifact lead to adaptions in respective application artifacts. This essentially contributes to a successive evolution of an SPL.

In the following, we explain feature models as a common way of modeling variability in an SPL during domain engineering.

2.1.1 Feature Models

The variability of software systems is a key factor of SPLs and encompasses features and their relationships. It allows to distinguish between product variants of a software system and to maintain variants in a centralized manner. Additionally, it supports changing requirements with respect to the environment, customer requests and legal issues.

During the domain engineering process, variability is implemented in different artifacts. Kang et al. introduce a *feature model*, which is commonly used to express all valid combinations of features, i.e., *product variants* of an SPL [35]. A feature model is a hierarchically arranged set of features represented by a tree structure. Subordinate feature are entitled as child features. Respectively, superordinate features are called parents features. The element at the top of the tree is the *root* feature, which represents the product line. A graphical representation of a feature model is called a *feature diagram*. In Figure 2.2, we illustrate a feature model for the mouse example SPL.

In a feature model, features are categorized as:

- Mandatory
  A feature, which always appears together with its parent feature in a product configuration (*left button, right button, sensor, connection*).

http://publikationsserver.tu-braunschweig.de/get/64218

- Optional
  A feature, which may appear in a product configuration. Hence, it is not obligatory to appear in a product configuration if its parent feature does (*scroll whell*).

- Alternative
  Exactly one child feature must be present in every product configuration. Feature groups in an alternative relationship exclude each other (a *sensor* can either consist of a *ball* or a *scanner*, but not both).

- Or
  At least one child feature must be present in every product configuration. Feature groups in an or relationship can be realized simultaneously (a mouse connection uses *USB* or *Bluetooth* or both).

Feature models might also comprise *core* features, which are present in every configuration. Consequently, core features represent a commonality of an SPL. Every core feature is mandatory and so are all of its predecessors in the feature tree topology (*left button, right button, sensor, connection*).

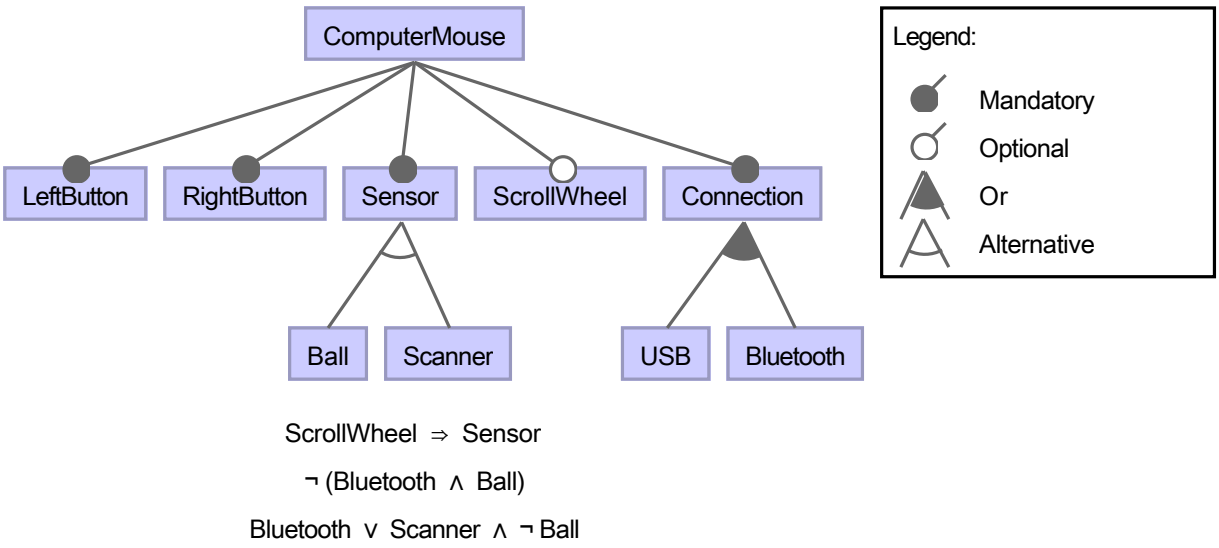

Figure 2.2: Feature model for a computer mouse example SPL.

Besides the feature tree topology, a feature model comprises *cross-tree constraints*. Cross-tree constraints define relationships between features that are not related by the tree structure. Propositional formulas are commonly used to express such constraints. For the mouse feature model, three exemplary cross-tree constraints are presented below the feature model. According to Batory [7], tools often approve only *simple* constraints representing either inclusions or exclusions between features. Such constraints are illustrated by the first and second cross-tree constraint in Figure 2.2. Nevertheless, *complex*

http://publikationsserver.tu-braunschweig.de/get/64218

constraints exist, which can represent any combination of features leading to an arbitrary propositional formula. The third cross-tree constraint presents an example for a complex constraint.

The variability modeling of a software system is not restricted to one single feature model. Several feature models can be used in order to decrease the complexity of an SPL for individual developers. In the next subchapter, we provide an overview about interrelated feature models for SPLs.

2.1.2 Interrelated Feature Models

A single feature model representing a large-scale system might become extremely complex, if the number of features and constraints reaches the hundreds or thousands. The maintenance of such feature models is a tedious and error-prone task, which gets even more difficult due to involved developers from different domains. To tackle this problem, large-scale systems often use multiple dedicated, interrelated feature models for different purposes, scopes and different levels of granularity. Regarding the purpose of a feature model, modeling spaces of feature models are distinguishable into problem space, solution space and configuration space. Problem space refers to system specifications while solution space refers to a concrete product variant. Feature models of the configuration space represent configuration options. A scope of a feature model is a complete SPL (a monolithic feature model) or a part of it. Furthermore, feature models can be defined on different levels of granularity, e.g., features on high-level or low-level system descriptions [41].
Multi software product line (MSPL) approaches have been proposed in order to support the modularization of feature models [31]. MSPLs are an emerging research topic and represent a special kind of SPLs. They are used to manage variability for large-scale systems and can represent interrelated feature models. Rosenmüller and Siegmund were the first to define MSPL with *composition models* integrating multiple SPLs and their dependencies [56].

In Figure 2.3, we demonstrate two interrelated feature models, which exemplary model hardware and software variability of a garage door. The garage door hardware comprises a door, an optional light barrier and lock sensor. Furthermore, additional functionalities include a lock LED signaling if the door is locked and an actuation motor for an automatic garage door. The garage door software consists of a software safety module, which interrupts an automating closing if a person crosses its path and a remote key recognition. A dependency between the two models origins from the actuation motor requiring the software safety module, which on its part requires a light barrier.

According to Lettner et al. [41], a need exists to manage dependencies between different feature models. The authors argue that it is often unclear how features from different feature models are related to each other, e.g., which feature from the configuration space is linked to its respective configuration option or which high-level feature refers to a feature implementing the low-level functionality. Additionally, cross-tree constraints between interrelated feature models can lead to *implicit constraints*. Implicit constraints

http://publikationsserver.tu-braunschweig.de/get/64218

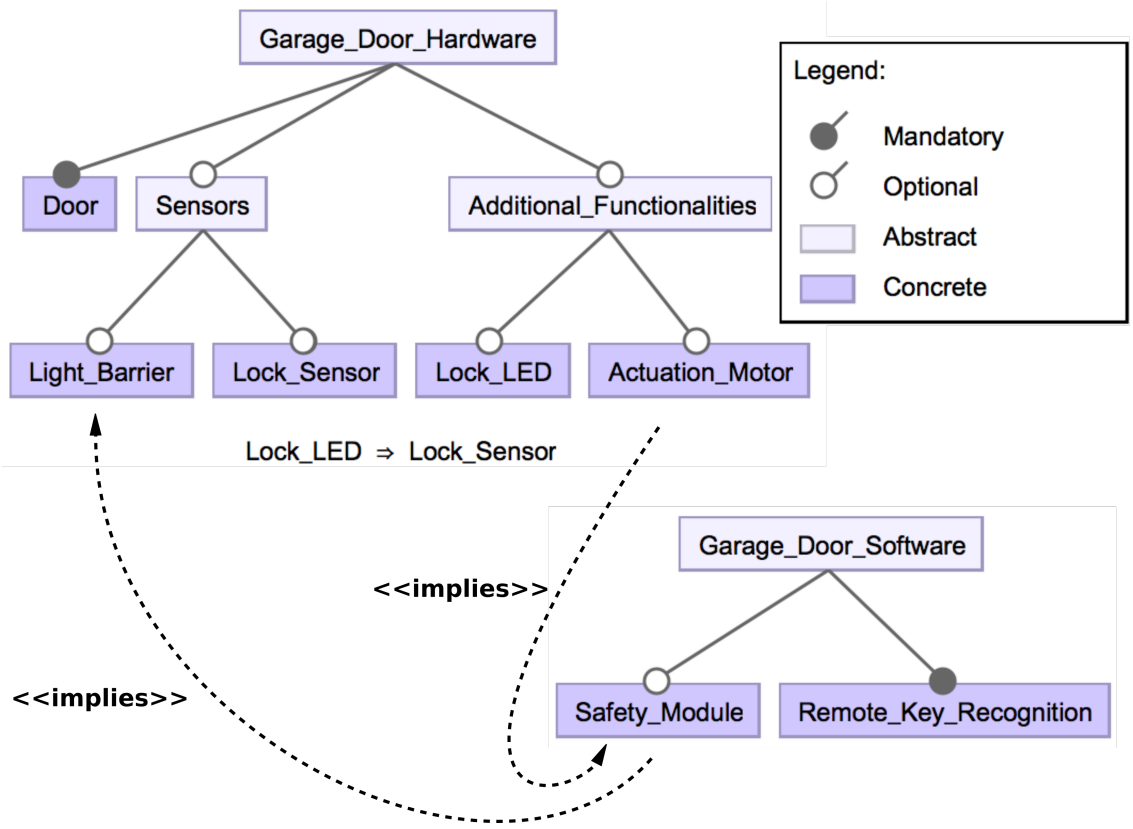

Figure 2.3: Example of interrelated feature models for a garage door SPL.

represent hidden dependencies caused by interrelated feature models. Such constraints represent a major challenge for maintenance and development of interrelated feature models. In Figure 2.3, a hidden dependency occurs in the feature model representing the garage door hardware:

Actuation_Motor $\Rightarrow$ ***Light_Barrier***

The implicit constraint results from the two cross-tree constraints *Actuation_Motor* $\Rightarrow$ *Safety_Module* and *Safety_Module* $\Rightarrow$ *Light_Barrier* and reveals a transitive relationship between Actuation_Motor and Light_Barrier.

While the visualization of feature models helps developers to built and maintain an SPL, a feature model itself can be expressed in terms of a propositional formula for analysis purposes, which we explain in the next subsection.

2.1.3 Propositional Formulas

The validation of a feature model encompasses a configuration of products that satisfy the constraints of the model. Validation is a difficult task due to the size of a feature model and dependencies, which increase its complexity. To tackle this problem, Mannion et al. were the first to propose the representation of a feature model using

http://publikationsserver.tu-braunschweig.de/get/64218

propositional formulas [43]. In detail, a propositional formula is a logical expression that consists of a set boolean variables and logical operators $\wedge, \vee, \neg, \Leftrightarrow, \Rightarrow$. Boolean variables are assigned a truth value, which is either *true* or *false*. A minimal example of a propositional formula is of the form: $a \wedge b \Rightarrow c$.

The representation of feature models by propositional formulas is as follows. Every feature is considered a variable, usually with the same name, and assigned *true* if the corresponding feature is selected. Logical operators represent relations between features. In Table 2.1, we present a mapping of the computer mouse feature model to propositional logic (cf. Figure 2.2). The conjunction of the resulting propositional formulas represents a feature model.

Relationship	Propositional Formula
Mandatory	$Mouse \Leftrightarrow LeftButton \wedge RightButton \wedge Sensor \wedge Connection$
Optional	$ScrollWheel \Rightarrow Mouse$
Or	$Connection \Leftrightarrow USB \vee Bluetooth$
Alternative	$(Ball \Leftrightarrow (\neg Scanner \wedge Sensor)) \wedge (Scanner \Leftrightarrow (\neg Ball \wedge Sensor))$
implies	$ScrollWheel \Rightarrow Sensor$
Excludes	$\neg(Ball \wedge Scanner)$

Table 2.1: Mapping feature models to propositional formulas.

A valid product configuration requires the complete propositional formula to be *true*. A *conjunctive normal form* (CNF) is commonly used as a notation of a propositional formula. It consists of a set of conjunct clauses. A clause consist of disjunct literals. A literal is a variable or its negation.

Besides the validation of a feature model, the representation of a feature model with a propositional formula enables different analyses purposes: retrieve statistics on feature models, ensure a correct implementation and serve as base for the explanation of defects [7, 39].

2.2 Defects in Feature Models

Requirements allocation during domain and application engineering leads to a great amount of requirements and dependencies between features. An approach to deal with this complexity are *feature models* (cf. Section 2.1.1). Nevertheless, a feature model might be subject to inconsistency, anomaly and redundancy caused by cross-tree constraints.

Von der Maßen and Lichter classify defects into three major groups [67]:

- Inconsistency: If contradictions between modeled relationships appear, a feature model is inconsistent. Inconsistencies are regarded as a severe issue, since they lead to inconsistent product configurations. Von der Maßen and Lichter identified

http://publikationsserver.tu-braunschweig.de/get/64218

inconsistencies at the domain level and at product configuration level. Inconsistency at domain level comprises contradictions within a feature model, e.g., a mutual exclusion between two core features. Inconsistency at product configuration level results from conflicting or incomplete product configurations, e.g., core features have not been selected for a product configuration.

- Anomaly: If at least one potential configuration is not possible to build although it should be, the feature model contains an anomaly. Anomalies are regarded as a medium issue. An example for an anomaly involves a core feature, which implies an optional feature. This results in the optional feature become a core feature as well.

- Redundancy: If at least one relationship is modeled in multiple ways, a feature model contains redundancy. Redundancy decreases maintainability, since changes must be applied to all redundant occurrences. However, redundancy can also be used to increase readability and understandability of a feature model, therefore it has not always a negative impact and is regarded as a light issue by the authors. An example for redundancy can be illustrated by an implication of a core feature by an optional feature. Since the core feature already appears in every configuration, the implication is unnecessary.

In the following subsections, we discuss defects in a feature model with respect to inconsistency, anomaly and redundancy. Additionally, we introduce implicit constraints as another form of anomaly in interrelated feature models.

2.2.1 Void Feature Models

A *void* feature model is an inconsistency as no product can be derived from the SPL. Figure 2.4 depicts a simple example of a void feature model. Constraint $\neg(B \wedge C)$ leads to features B and C being mutual exclusive and adds a contradiction to the feature model, since both features are core features. A void feature model is a severe issue, because it is not possible to derive any product variant of the SPL.

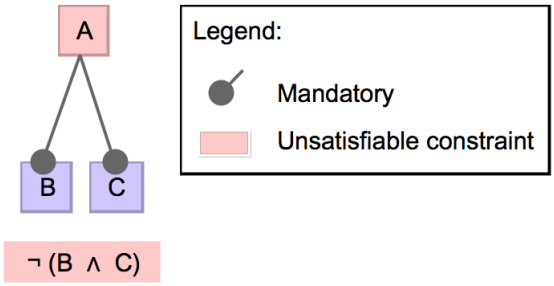

Figure 2.4: Example of a void feature model.

http://publikationsserver.tu-braunschweig.de/get/64218

2.2.2 Dead Features

Regarding the defect categorization of von der Maßen and Lichter, a dead feature represents an anomaly in a feature model [67]. A feature is dead, if it does not appear in any valid configuration of an SPL. Figure 2.5 illustrates common examples for dead features. In the first case, feature D is dead, because it is alternative to feature E, which is implied by a core feature B. Therefore, feature B and E will appear in every product configuration while D cannot appear in any product variant. In the second case, feature B and C are mutual exclusive to each other, while feature B is a core feature. Since feature B is a core feature, feature C cannot appear in any product of the SPL.

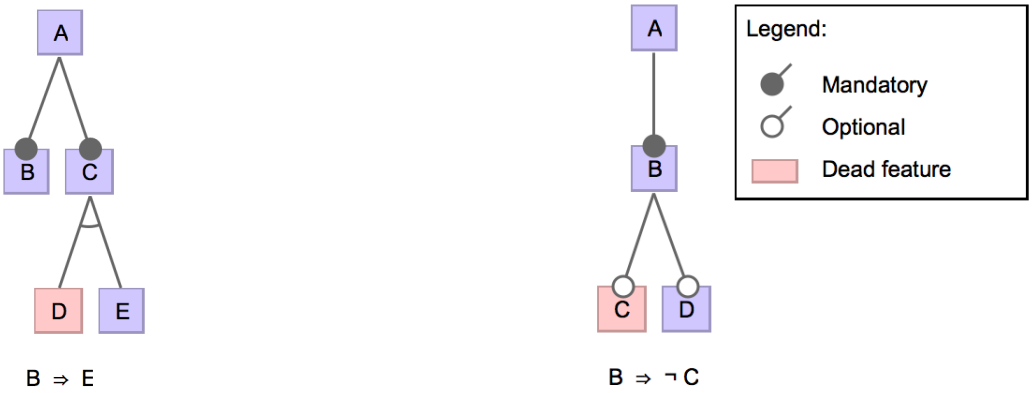

Figure 2.5: Two examples of dead features.

2.2.3 False-Optional Features

Same as dead features, *false-optional* features also represent anomalies in a feature model. A feature is false-optional, if it is available with its parent feature in all products of the SPL although it is modeled as optional. Figure 2.6 illustrates common examples for false-optional features. In the first case, the feature B is false-optional, because it is implied by core feature C. In the second case, core feature B implies feature D. Since an alternative relationship allows exactly one feature selection, feature E becomes dead and feature D must be selected. This results in the parent feature C becoming false-optional.

2.2.4 Redundant Constraints

Generally speaking, redundancy occurs if the removal of information does not change the semantic of a feature model, e.g., the removal of a redundant cross-tree constraint does not affect the validity of configurations. Redundancy in a feature model origins from different sources. First, redundancy occurs if a feature appears multiple times in a feature model with different parent features. Since FeatureIDE forbids the creation of identical-named features, we ignore this kind of redundancy at this point. Second, redundancy occurs due to combinations of domain relationships within a feature tree and dependencies, which arise through cross-tree constraints [67]. Von der Maßen and Lichter identified five such cases of redundant cross-tree constraints:

http://publikationsserver.tu-braunschweig.de/get/64218

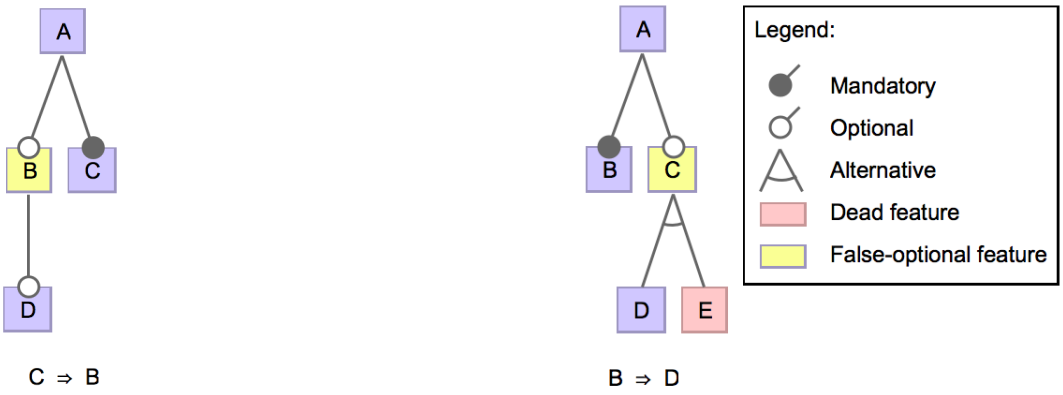

Figure 2.6: Two examples of false-optional features.

1. Mandatory and Implication:
   Figure 2.7 presents two redundant cross-tree constraints concerning a mandatory feature and its implication. In the first case, core feature C is implied by feature B. This relationships is superfluous, since C is a core feature and therefore appears in every configuration regardless from further implications. In the second case, a mandatory child feature D is implied by its mandatory parent feature C. This implication is also superfluous, since both features are relative-mandatory to each other and will always appear together in a configuration.

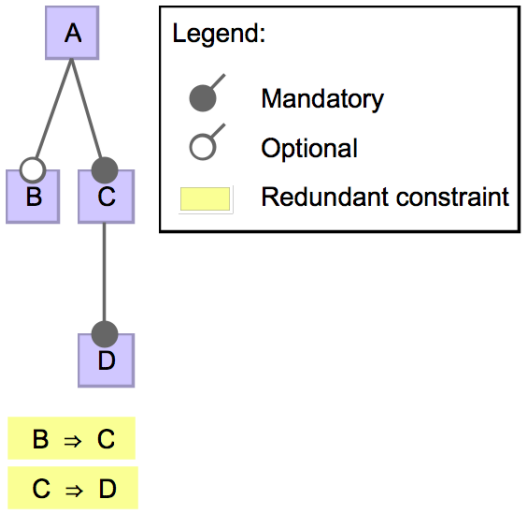

Figure 2.7: Redundant cross-tree constraints: mandatory and implication

2. Alternative and Exclusion:
   Figure 2.8 illustrates a redundant cross-tree-constraint concerning an alternative feature-group *B, C* and its mutual exclusion. This is superfluous, because an alternative relationship already includes a mutual exclusion.

http://publikationsserver.tu-braunschweig.de/get/64218

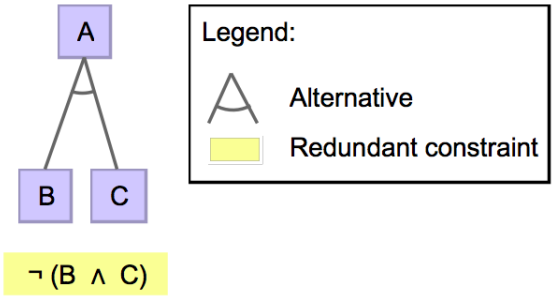

Figure 2.8: Redundant cross-tree constraints: alternative and exclusion

3. Multiple Implications:
   Figure 2.9 presents a case of redundancy due to multiple implications. Constraint $\neg(B \Rightarrow E)$ models an implication of feature E by feature B. Since the mandatory feature C is a child feature of B, its implication of feature E is superfluous and therefore the constraint $(C \Rightarrow E)$ becomes redundant.

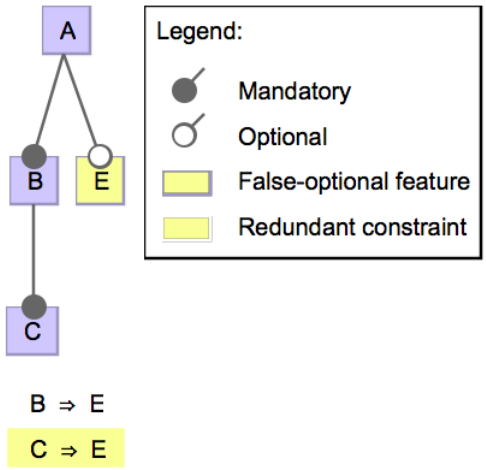

Figure 2.9: Redundant cross-tree constraints: multiple implications

4. Multiple Exclusions:
   Figure 2.10 presents a case of redundancy due to multiple exclusions. Constraint $\neg(B \wedge E)$ models a mutual exclusion between features B and E. Since the mandatory feature C is a child feature of B, its mutually exclusiveness to feature E is superfluous and therefore the constraint $\neg(C \wedge E)$ becomes redundant.

5. Transitive Implications:
   Figure 2.11 illustrates a case of a transitive redundancy. Since feature B is is a mandatory child of feature A and feature C is a mandatory child of feature B, a transitive relationship between feature A and feature C is created. This leads to a superfluous implication of feature C by feature A.

http://publikationsserver.tu-braunschweig.de/get/64218

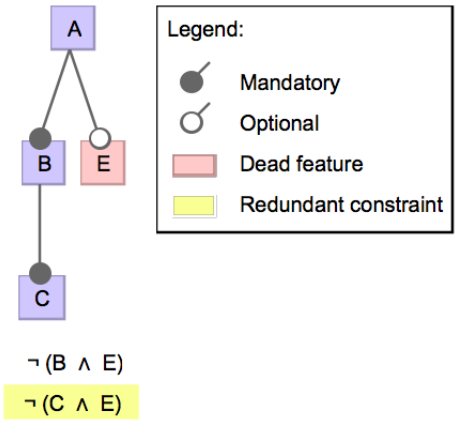

Figure 2.10: Redundant cross-tree constraints: multiple exclusions

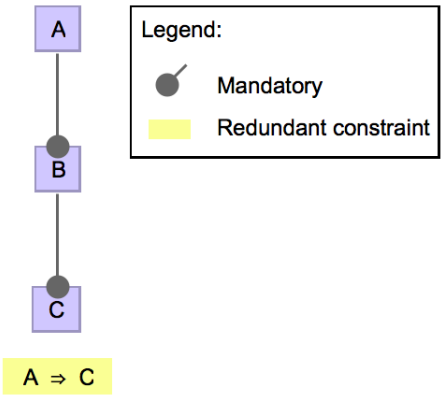

Figure 2.11: Redundant cross-tree constraints: transitive implications

Redundant cross-tree constraints also comprise trivial cases of redundancy by modeling domain relationships and dependencies multiple times with the identical logical expression.

2.2.5 Implicit Constraints

A rarely considered defect is an implicit constraint, which we classify as an anomaly. An implicit constraint represents a hidden dependency caused by interrelated feature models (cf. Section 2.1.2). Implicit constraints may lead to erroneous configurations and have a negative effect on the maintenance of interrelated feature models.

Figure 2.12 presents two interrelated feature models FM1 and FM2. Feature C from FM1 implies feature G from the FM2. Feature G, on its part, implies feature D from FM1. Both cross-tree constraints result in a transitiv, hidden implication between Feature C and D in FM1:

$C \Rightarrow D$

http://publikationsserver.tu-braunschweig.de/get/64218

Figure 2.13 presents another use case of an implicit constraint in FM1. Feature C from FM1 implies feature G from FM2. Feature D from FM1 implies feature H from FM2. An alternative relationship between feature G and H in FM2 results in a hidden mutual exclusion between feature C and D in FM1.

$\neg(C \wedge D)$

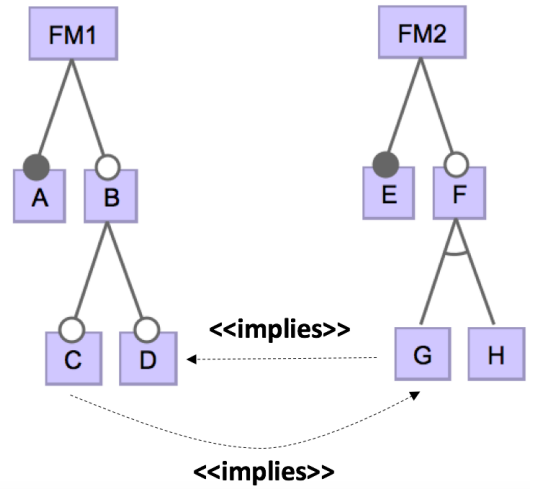

Figure 2.12: Hidden implication

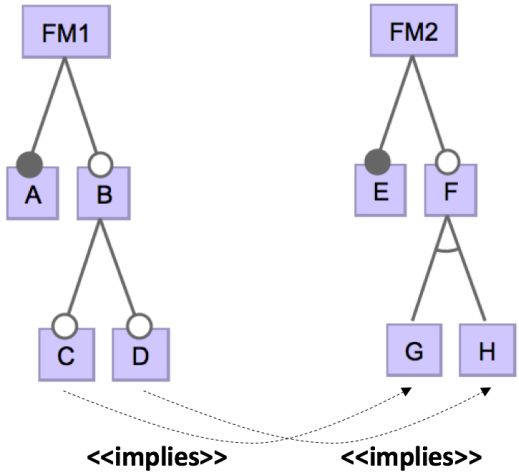

Figure 2.13: Hidden exclusion

To assure the quality of interrelated models, the automated detection of hidden dependencies has become an important field of research [39, 41].

2.3 Automated Analysis of Feature Models

The automated analysis of feature models takes a feature model as an input, performs an analysis operation and returns its result. A usual purpose of an analysis operation is the detection of defects. In Section 2.2, we presented different kinds of defects in a feature model. Next, we explain how these defects are detected in a feature model.

2.3.1 Detection of Defects

First, we consider the detection of defects in single feature models, i.e., void feature models, dead and false-optional features as well as redundant cross-tree constraint. Afterwards, we provide information on how to reveal implicit constraints in interrelated feature models.

Defects in Single Feature Models

Propositional formulas are a useful representation of feature models in order to perform automated analyses (cf. Section 2.1.3). Every propositional formula can be transformed into a CNF, which is used by most of satisfiability *SAT* solvers [10]. A SAT solver determines for a given formula whether it is satisfiable, i.e., a truth value assignment

http://publikationsserver.tu-braunschweig.de/get/64218

exists that makes the formula true. If such a value assignment exists, at least one product configuration can be derived from the feature model proving that the feature model is not void. In Table 2.2, we define how defects can be detected with a SAT solver.
In order to detect a dead feature, the SAT solver determines whether a satisfying value assignment exists, which includes a certain feature to be selected. If not, the respective feature is considered to be dead. The detection of a false-optional feature involves the selection of its parent feature. If this leads to a selection of the feature itself and the SAT solver cannot determine a satisfying truth value assignment, the feature is considered to be false-optional. In order to prove that a constraint is redundant, two feature models must be equivalent to each other, while the one feature model contains the redundant cross-tree constraint and the other does not.

$$
\begin{aligned}
void(FM) &:= \neg SAT(FM) \\
dead(f) &:= \neg SAT(FM \wedge f) \\
falseOpt(f_{opt}) &:= TAUT(FM \wedge p(f_{opt}) \Rightarrow f_{opt}) \\
redundant(c) &:= TAUT(FM' \Leftrightarrow FM' \wedge c) \\
with\ TAUT(x) &:= \neg SAT(\neg x)
\end{aligned}
$$

Table 2.2: Defect detection with a SAT solver. FM = feature model, f = feature, f_{opt} = optional feature, p = parent of feature f_{opt}, c = cross-tree constraint, $FM = FM' \wedge c$, and x = propositional formula [37].

Although SAT solving is NP-complete [17], algorithms using efficient heuristics lead to an tremendously improved performance [45]. Within FeatureIDE, the detection of defects defined in Table 2.2 is performed using a SAT solver.

It is also possible to use other solving strategies not relying on CNF such as the *binary decision diagram* (BDD) solver. A BDD solver transforms the propositional formula of a feature model into a binary decision diagram. One the one hand, a BDD allows to determine if the formula is satisfiable. On the other hand, the paths of a BDD represent all product configurations thus revealing the number of possible variants. Finding a variable ordering reducing the size of a BDD is known to be NP-complete.
Benavides et al. present an overview about studies proposing the usage of different *off-the-shelf* solvers like SAT or BDD for feature model analyses [10]. Additionally, Benavides et al. introduce a combination of solvers depending on the kind of analysis and particular advantages of the different solvers [11, 12].

Implicit Constraints

The automated analysis of feature models also comprises the classification of *edits* between two feature models [62]. An edit can be a *specialization* of a feature model, i.e., feature model containing a smaller amount of features compared to the original model.

http://publikationsserver.tu-braunschweig.de/get/64218

When deriving specialized models, the elimination of features must not change dependencies between the remaining features. A state-of-the-art approach to remove features while maintaining dependencies between other features is *feature model slicing* [3, 62]. Feature model slicing can be applied for different scenarios, i.e., feature model evolution, removing abstract features and for the decomposition of feature models [2, 64]. Krieter et al. present an efficient algorithm for feature model slicing in FeatureIDE [39]. Inputs to the algorithm comprise a feature model in CNF and a set of features. The feature model parameter represents a complete feature model before the slicing operation. After performing the feature model slicing, the algorithm returns a sliced feature model in CNF without the specified set of features while maintaining dependencies between features in the sliced model. The core of the slicing algorithm is *logical resolution*. The main idea of logical resolution involves the construction of a new clause, which represents a relationship between features of the sliced feature model. The authors refer to this clause as *resolvent*, represented by a new cross-tree constraint of the sliced feature model. The construction of a resolvent requires two clauses such that the first clause contains the literal to remove in its positive form and the second clause contains the literal to remove in its negated form. The authors derive the resolvent by combining the two clauses and removing the respective literal. The resolvent, on its part, represents a transitive relationship between the two clauses. We consider the resolvent as an **implicit constraint**.

In 2010, a literature review revealed up to 30 analysis operations using up to 10 different solvers [10]. Nowadays, the number of analysis operations increased to 40 and is still an ongoing research in the SPL community [65].

2.3.2 Explanation of Defects

Generating explanations for defects in feature models is one of the several analysis operations. Usually, an explanation informs a user about features and/or relationships which lead to a defect [10].

Batory introduced one of the first explanation approaches based on a *logic truth maintenance system* (LTMS), a boolean constraint propagation-based approach, which is independent from any solver [7]. An LTMS derives assumptions about variable truth values in a propositional formula and maintains the reason for an inference. Based on all stored reasons, the generation of an explanation takes place (cf. Section 3.2.1). Batory implemented a tool called *guidsl*, which supports the development of SPLs including the generation of explanations based on an LTMS. For the dead feature C in Figure 2.14, *guidsl* generates the following explanation:

A has a contradiction
C because set by user
not B because (B) implies (not C)
not A because (A iff B)
not A because it is root of grammar
But A

http://publikationsserver.tu-braunschweig.de/get/64218

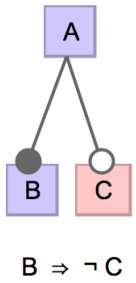

Figure 2.14: Example of a feature model with a dead feature C.

By assuming D to be set by a user for a product configuration, *guidsl* reveals a contradiction within the feature model. This is due to an inference resulting in the root feature to be excluded from the product configuration, which on its part must always be part of a product.

Another way to express explanations is introduced by Trinidad [10–12, 66]. Trinidad uses the *Theory of Diagnosis* by Reiter, which has been widely used in order to detect a minimal set of faulty components leading to an abnormal behavior [54]. In order to apply the theory of diagnosis on a feature model, Trinidad expresses a feature model in terms of a *constraint satisfaction problem* (CSP). Components are represented as features and constraints. A CSP contains a set of features, i.e., variables, and constraints restricting the values of the variables. Finding a value assignment, which satisfies all constraints simultaneously is a solution to a CSP. Applying the theory of diagnosis to a feature model containing a defect leads to at least one minimal set of faulty constraints. If several minimal sets are detected, one explanation is generated per set. The explanations consist of the set of faulty constraints leading to the defect. For the dead feature C in Figure 2.14, two minimal explanations exist: First, the mandatory relationship to B and, second, the first cross-tree constraint (CTC-1) comprising an excludes-relationship between feature B and C. The explanations in FAMA are expressed in the following manner:

```
1: Dead Feature: C (2 explanations) -> [to-B-rel], [CTC-1]
```

In his early work, Trinidad focuses on the explanation of dead features [66]. Nowadays, the framework is used to explain all kind of defects, implemented in the FAMA tool suite [11, 12].

Rincón et al. present an ontological rule-based approach to detect and explain dead and false-optional features in natural language [55]. Therefore, the authors construct an ontology that represents a specification of concepts in a feature model and which is used to identify dead and false-optional features. Based on the ontology and formalizing rules leading to dead and false-optional features, the authors are able to assign every defect feature to a respective rule. Depending on the rule, a generation of an explanation in natural language takes place. Below, an excerpt of the results after analyzing a feature

http://publikationsserver.tu-braunschweig.de/get/64218

model with dead features is shown. For every defect, the authors provide a classification, name the corresponding defect feature and give an explanation in natural language.

Analyzing the defects...
Defect:.....DEAD_FEATURE
Feature:.....AF2
Causes: FULL MANDATORY FEATURE EXCLUDES AN OPTIONAL FEATURE
Explanation: Optional feature AF2 is dead because it is excluded by the full mandatory feature AF3 with the dependency AD19

To summarize, several approaches concentrate on generating explanations within feature models. Explanations range from presenting significant constraints leading to a defect up to generating explanations in natural and user-friendly language. Within this thesis, we extend the work of Batory using an LTMS to generate explanations for all described defects [7] (cf. Section 2.2). In the next chapter, we explain the concept of our approach in detail.

http://publikationsserver.tu-braunschweig.de/get/64218

http://publikationsserver.tu-braunschweig.de/get/64218

3. A Conceptual Framework to Generate Explanations

Finding contradictions in a feature model is challenging. In order to ensure the quality of a software product line, quality assurance measures have to be provided to the model designer. Generating dynamic answers in the form of *why* a certain defect has occurred during usage, leads to a significantly simplified and faster error repairing. Finding errors in feature models is important to both model designers and customers of the SPL. The overall goal of explanations is to extract only relevant parts of the feature model causing a defect and present those in a human-understandable form afterwards.

In this chapter, we present a generic explanation approach for defects in feature models contributing to **RG1** and **RG2**. In Section 3.1, we first define requirements for an explanation algorithm. Next, we describe the basics of a suitable algorithm found in literature in Section 3.2. By reasoning on the adaptions of the algorithm to explain all kinds of defects (cf. Section 2.2) and providing minimal use-cases as working examples, we achieve **RG1.1**. Additionally, we present how to detect and explain implicit constraints in interrelated feature models to reach **RG2.1** and **RG2.2**. In Section 3.4, we suggest improvements covering the generation of shortest explanations along with highlighting parts in explanations, which have a higher probability to cause the defect. This is essential to realize **RG1.3**. Parts of this chapter are available in [37].

3.1 Requirements of the Explanation Algorithm

The starting point is the selection of an appropriate algorithm to explain defects in feature models, which shall meet the following requirements:

- *generic*
  The algorithm shall be able to explain all previously mentioned defects (cf. Section 2.2).

http://publikationsserver.tu-braunschweig.de/get/64218

- *efficient*
  Variability modeling can include an enormous number of features and constraints, i.e., several thousands. The algorithm shall be economic in terms of resources, computing time and memory and, hence, scale to large feature models.

- *informative*
  In order to increase the usability of explanations, they shall be as short as possible and significant parts, which are more likely to cause a defect, should be highlighted.

Batory implemented a tool called *guidsl*, which serves for product-line development and that is part of the AHEAD Tool Suite [7]. It successfully adapts basic ideas of the *boolean constraint propagation* (BCP) algorithm in order to provide justifications for the selection or deselection of features during a configuration process. BCP is an efficient algorithm, which fulfills the listed requirements (cf. Section 3.2.2). It is used frequently for implementing an LTMS and represents its inference engine [27]. Consequently, *guidsl* serves as a model example to apply BCP for explanations in FeatureIDE.

3.2 Basic Algorithm

In this section, we propose a generic explanation algorithm, which is able to explain all defects in a feature model only by varying two input parameters. We describe the functionality of an LTMS and its internal inference engine BCP. Additionally, we reason on the benefits and limitations of the explanation algorithm.

3.2.1 Logic Truth Maintenance System

Truth maintenance (TM) is a wide spread approach in the area of artificial intelligence for implementing inference systems. The core of a *truth maintenance system* (TMS) is an *inference engine* (IE), which derives assumptions about variable values and maintains the reason for its belief. A TMS can be used to perform a range of activities such as explanation capabilities, reasoning and deductions [22].

Different types of implemented TMSs exist, i.e., a *justification-based TMS* (JTMS), an *assumption-based TMS* (ATMS) and an LTMS [36, 46, 57]. They differ in their fundamental structure, implementation and functions. For explanation capabilities, we chose the LTMS due to several reasons. In contrast to an ATMS and a JTMS, an LTMS carries out logical operations and understands propositional semantics, i.e., it can't represent a positive and negated value of the same variable simultaneously [18]. Additionally, an LTMS allows any arbitrary propositional formula as input whereas both JTMS and LTMS are restricted to horn clauses (i.e., formulas of the form $A \wedge ... \wedge D \Rightarrow Z$) [27]. Since we operate with arbitrary logical formulas representing feature models, we require these properties.

An LTMS is a boolean constraint propagation-based approach, which can be used to infer an explanation. As an LTMS is based on logical constructions, the following logical specification is needed:

http://publikationsserver.tu-braunschweig.de/get/64218

- a set of boolean variables
- a propositional formula consisting of clauses constraining these variables
- premises, which are permanent assignments of truth values to variables
- a set of assumptions, which are assignments of truth values to variables that may be later revoked

The basic principle of an LTMS for generating explanations is depicted in Figure 3.1. A formula is derived from the feature model (cf. Section 2.1.3) and a premise is assumed which leads to a contradiction during propagation. Whenever an inference is made, an LTMS stores the reason for an inference. The occurrence of a contradiction, for example $root = false$, reveals an inconsistency in the feature model. After the detection of a contradiction, an explanation is generated based on the stored reasons. As a result, an LTMS can be used to assist on error repairing during SPL modeling.

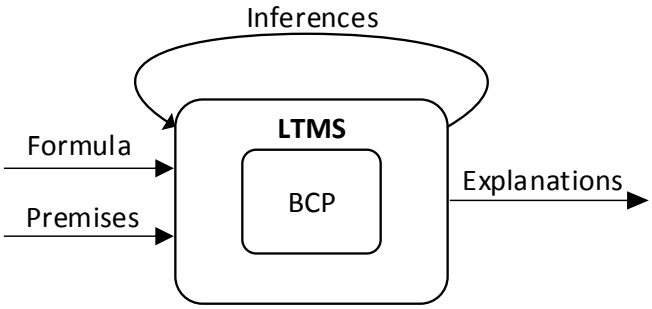

Figure 3.1: LTMS

3.2.2 Boolean Constraint Propagation

BCP is considered as a sound and efficient algorithm for implementing the logical specification of an LTMS [27]. Boolean constraints are represented by means of boolean formulas and are a special case of CSPs [5]. They use typical connectives such as AND, OR and NOT to combine variables. To reason about boolean constraints, rules can be applied to propagate known values for boolean variables.

1. $X \wedge Y = Z$: If Z = *true*, then X and Y must be true.
2. $X \vee Y = Z$: If Z = *true* and X = *false*, then Y must be true.

BCP is also known as *Unit Resolution* and makes use of such rules to conclude inferences [20]. Input to a BCP is usually specified as a set of variables defined by a three-value logic (true, false, unknown) and a formula in CNF. A formula is *satisfied*, if

http://publikationsserver.tu-braunschweig.de/get/64218

at least one literal in every clause is true. Consequently, a truth value assignment that satisfies the formula represents a product in the SPL [7].

The basic idea of a BCP assigns a type to every clause:

- Satisfied: at least one literal is true
- Violated: all literals are false
- Unit-Open: one literal is unknown (unbound) while remaining literals are false
- Non Unit-Open: some literals are unknown, the rest is false

Hence, a unit-open clause can be satisfied by setting its unbound literal to true and a violated clause is considered to be a contradiction.

Example. Regarding the clause $\neg A \vee B \vee C$, the different types are demonstrated:

- If A is false, the clause is **satisfied**.
- If A is true, B is false and C is false, the clause is **violated**.
- If A is true, B is false and C is unknown, the clause is **unit-open**. C is derived as true.
- If A is true and B and C are unknown, the clause is **non unit-open**.

Figure 3.2 presents a general overview of the BCP algorithm: BCP is invoked on initial truth value assignments, which represent premises. In the first iteration, the algorithm processes every clause in the CNF and pushes unit-open clauses it encounters on stack. After processing the complete CNF, the latest unit-open clause is removed from the stack. Depending on the unit-open clause, BCP infers and updates a truth value for the unbound literal and restarts the propagation process. This is generally defined as *unit-propagation*. BCP terminates and reports a contradiction as soon as it detects a violated clause. Hence, detecting a contradiction on the basis of premises is a substantial part for generating explanations with BCP.

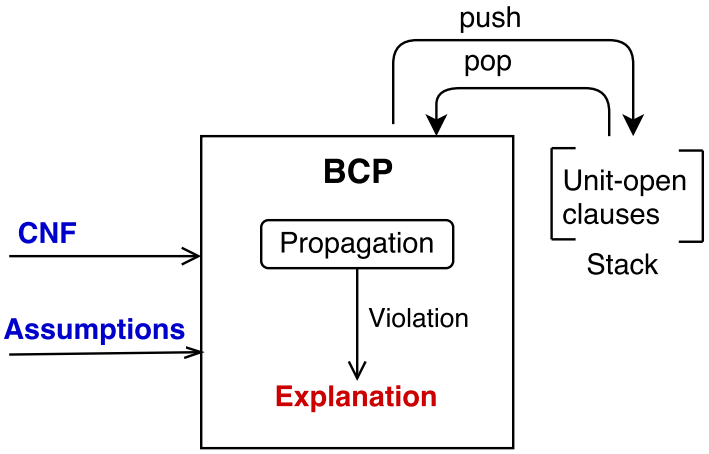

Figure 3.2: Overview of BCP.

http://publikationsserver.tu-braunschweig.de/get/64218

The bookkeeping of a BCP algorithm consists of a 3-tuple <*conclusion, reason,* {*antecedents*}> for every derived value assignment. *Conclusion* represents a value assignment to a variable. *Reason* is the predicate or unit-open clause that lead to the derived value. *Antecedents* are the remaining variables in the unit-open clause whose values were referenced and for which the algorithm also maintains a 3-tuple.

Example. Consider the formula of a feature model: $(A \Rightarrow B) \wedge (B \Rightarrow \neg A)$
The formula is transformed to a CNF: $(\neg A \vee B) \wedge (\neg B \vee \neg A)$

As presented in Table 3.1, BCP sets A=*true* and remembers its reason to be a *premise*. A premise does not have antecedents. BCP pushes respective unit-open clauses from the CNF on stack. After examining $(\neg B \vee \neg A)$, BCP infers B=*false* and stores its unit-open clause as reason. It refers to variable A as antecedent, since its value was referenced. The BCP algorithm discovers the violated clause $(\neg A \vee B)$ and generates an explanation, which is based on the violated clause and stored reasons [37].

CNF: $(A) \wedge (\neg A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee A) \wedge (\neg C \vee A) \wedge (\neg B \vee \neg C)$				
ID	Conclusion	Reason	Antecedents	Stack
#1	A=1	premise		$(\neg A \vee B), (\neg B \vee \neg A)$
#2	B=0	$(\neg B \vee \neg A)$	#1	$(\neg A \vee B)$
Violated clause: $(\neg A \vee B)$				

Table 3.1: Stepwise example of the BCP process.

The described algorithm holds the specified requirements in Section 3.1, which we put in concrete terms in the following.

Benefits

BCP comprises the following beneficial characteristics:

- *generic*
  BCP works on the basis of a propositional formula. Since every feature model can be mapped to a propositional formula, described in Section 2.1.3, BCP can process every kind of feature model regardless of the kind of defect.
- *efficient*
  The BCP process requires a quadratic time depending on the sum of literals contained in every clause [24].
- *informative*
  Since BCP only maintains reasons leading to a contradiction, explanations comprise relevant information for a defect. Clauses, which do not result in a violation, are ignored.

Along with beneficial characteristics, BCP suffers from several limitations, which we present next.

http://publikationsserver.tu-braunschweig.de/get/64218

Limitations

BCP comprises the following limitations:

- *order-sensitive*
  Assuming that a CNF is processed from the left to the right and a stack is used to maintain unit-open clauses, the explanation depends on the order of clauses. Processing a CNF the another way may lead to a different explanation. This characteristic also implies that the BCP algorithm per se does not always find an explanation with a minimal length.

- *incomplete*
  BCP cannot infer all truth values. Consider two clauses:

  $$(A \Rightarrow B) \wedge (B \Rightarrow \neg A)$$

  The first clause selects B if A is selected, while the second clause deselects A. Therefore, selecting A leads to its deselection. BCP is not able to infer that A cannot be present in a product (dead feature) and only the selection of A reveals the contradiction [30]. However, this characteristic is not restrictive to explain defects in feature models, since FeatureIDE already detects that A is a dead feature.

All in all, LTMS and its internal inference engine BCP are able to generate explanations consisting of propositional formulas. For the tool *guidsl*, BCP is used to inform a user why the selection of a specific feature is not able during the configuration process of an SPL [7]. To the best of our knowledge, BCP was never used to explain all defects in a feature model.

3.3 Explaining Defects

In this section, we reason on how to vary the input parameters for the BPC algorithm in order to explain void feature models, dead and false-optional features as well as redundant and implicit constraints. Minimal use cases for every defect serve as examples to demonstrate the applicability of BCP. Additionally, we suggest improvements for explanations concerning their length and emphasis of parts that are more likely to cause a defect. The detection of defects is already performed by FeatureIDE.

3.3.1 Void Feature Models

A void feature model represents an inconsistency in a feature model. An inconsistency is regarded as a severe form of a defect, since no valid product configurations can be derived from the SPL (cf. Section 2.2.1). Consider the feature model illustrated in Figure 3.3. Feature B and C exclude each other mutually resulting in a void feature model.

http://publikationsserver.tu-braunschweig.de/get/64218

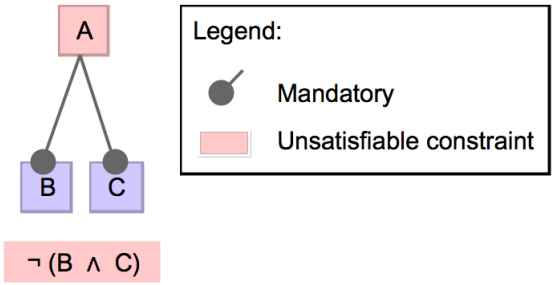

Figure 3.3: A void feature model.

To explain why the feature model is void, BCP requires two input parameters. First, the CNF of the feature model. Second, a premise which leads to a contradiction. A contradiction occurs, if the root feature is assumed to be *true* which is not the case in a void feature model. An explanation consists of the violated clause and the stored reasons during propagation. Table 3.2 demonstrates the single steps of explaining the void feature model depicted in Figure 3.3. First, we build the CNF of the feature model. During the creation of the CNF, we extend every literal with tracing information, i.e., every literal belongs to a clause, which either represents a relationship within the feature-tree topology or a cross-tree constraint. Together with the CNF, we pass a premise $root = true$ to the BCP algorithm. Since the root feature is initially bound, it does not have any antecedents. Note that all other literals except for premises are initially unbound. BCP processes the CNF and pushes all unit-open clauses on stack. Afterwards, BCP removes the last occurred unit-open clause $(\neg A \vee C)$ and concludes C to be *true* in order to satisfy the clause. Propagation terminates with detecting the violated clause $(\neg A \vee B)$. The tracing information allows us to express explanations in a user-friendly manner. Every explanation comprises the violated clause first, followed by all stored reasons. To reduce the length of an explanation, premises are omitted in explanations.

CNF: $(A) \wedge (\neg A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee A) \wedge (\neg C \vee A) \wedge (\neg B \vee \neg C)$				
ID	Conclusion	Reason	Antecedents	Stack
#1	root=1	premise		$(\neg A \vee B), (\neg A \vee C)$
#2	C=1	$(\neg A \vee C)$	#1	$(\neg A \vee B), (\neg B \vee \neg C)$
#3	B=0	$(\neg B \vee \neg C)$	#2	$(\neg A \vee B)$
Violated clause: $(\neg A \vee B)$				
Explanation: *Feature Model is void, because: B is mandatory child of A (violated clause),* $\neg(B \wedge C)$ *is Constraint (#3), C is mandatory child of A (#2).*				

Table 3.2: Explaining a void feature model.

Similar to explaining a void feature model, BCP explains a dead feature, which we present in the following.

http://publikationsserver.tu-braunschweig.de/get/64218

3.3.2 Dead Features

Another defect might occur in the form of dead features (cf. Section 2.2.2). The explanation for dead features is similar to the explanation of a void feature model, namely also interpreting a dead feature to be *true* as we did for the root feature of a void feature model. This will lead to a contradiction, because a dead feature cannot appear in any configuration. Hence, we pass BCP the CNF of the feature model and a premise including the dead feature to be *true*.

An example to explain dead features is demonstrated with the feature model illustrated Figure 3.4. An alternative Feature E is implied by a core feature B resulting in an alternative Feature D to be dead. Since B will always appear in every configuration, D will never appear in any configuration of the SPL.

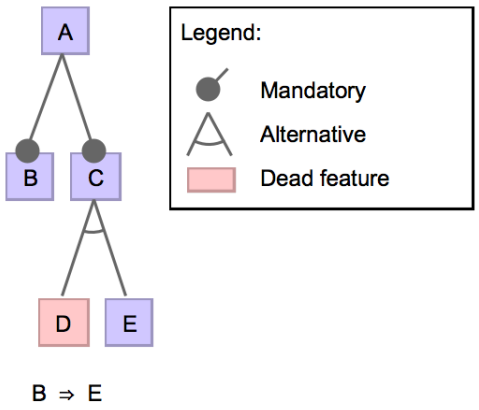

Figure 3.4: Feature model with dead feature D.

In Table 3.3, we demonstrate the BCP process. We pass the CNF and a premise involving the selection of the dead feature D (D=*true*). After putting all unit-open clauses on stack, BCP removes the last unit-open clause along with concluding and updating the truth value for literal A. In the following, BCP infers truth values for the literals E, B, and A. The last inference sets the root-feature A to *false* leading to a violated clause *(A)*.

3.3.3 False-Optional Features

Another form of defects in a feature model are false-optional features [67]. Such features are modeled as optional, but are present in every product of the SPL together with their parent feature (cf. Section 2.2.3).

We pass to the BCP algorithm the following two parameters: The CNF of the feature model and premises, which comprise the false-optional feature to be *false* and its parent feature to be *true*. With the premises, BCP will detect a contradiction, since the false-optional feature will always be selected together with its parent feature for every configuration.

http://publikationsserver.tu-braunschweig.de/get/64218

CNF: $A \wedge (\neg A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee A) \wedge (\neg C \vee A) \wedge (\neg C \vee D \vee E) \wedge (\neg D \vee C) \wedge (\neg E \vee C) \wedge (\neg D \vee \neg E) \wedge (\neg B \vee E)$				
ID	Conclusion	Reason	Antecedents	Stack
#1	D=1	premise		$(\neg D \vee C), (\neg D \vee \neg E)$
#2	E=0	$(\neg D \vee \neg E)$	#1	$(\neg D \vee C), (\neg B \vee E)$
#3	B=0	$(\neg B \vee E)$	#2	$(\neg D \vee C), (\neg A \vee B)$
#4	A=0	$(\neg A \vee B)$	#3	$(\neg D \vee C)$
Violated clause: (A)				
Explanation: *Feature D is dead, because: A is root (violated clause), B is mandatory child of A (#4), B ⇒ E is Constraint (#3), E is alternative child of C (#2), D is alternative child of C (#2).*				

Table 3.3: Explaining a dead feature.

An example for a false-optional feature is illustrated in Figure 3.5. Feature D is false-optional, because a mandatory feature C implies D. In Table 3.4, we demonstrate the BCP process in order to generate an explanation. By setting the false-optional feature D to *false* and its parent feature B to *true*, BCP concludes feature C to be *false*. This results in a violation of the clause $(\neg B \vee C)$.

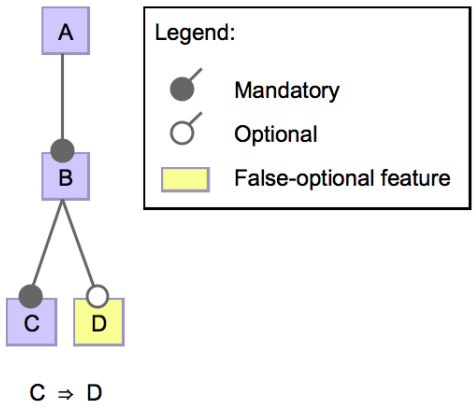

Figure 3.5: Feature model with false-optional feature D.

3.3.4 Redundant Constraints

Redundancy in feature models occurs if semantic information is modeled in multiple ways (cf. Section 2.2.4). Redundant information origins from already modeled relationships within the the feature-tree or in cross-tree constraints.

To explain redundant cross-tree constraints, we need to reason on the two input parameters for BCP, i.e., the CNF and premises. For the defects described previously, we pass the CNF of the feature model and a premise leading to a contradiction to immediately retrieve an explanation. However, explaining redundant constraints is more challenging [37]. We reason on two main questions:

http://publikationsserver.tu-braunschweig.de/get/64218

CNF: $A \wedge (\neg A \vee B) \wedge (\neg B \vee A) \wedge (\neg B \vee C) \wedge (\neg C \vee B) \wedge (\neg D \vee B) \wedge (\neg C \vee D)$				
ID	Conclusion	Reason	Antecedents	Stack
#1	D=0	premise		
#2	B=1	premise		$(\neg B \vee A), (\neg B \vee C), (\neg C \vee D)$
#3	C=0	$(\neg C \vee D)$	#1	$(\neg B \vee A), (\neg B \vee C)$**!**
Violated clause: $(\neg B \vee C)$				
Explanation: *Feature D is false-optional, because: C is mandatory child of B, $C \Rightarrow D$ is Constraint.*				

Table 3.4: Explaining a false-optional feature.

1. *Shall the CNF include the redundant constraint?*
   A cross-tree constraint is only detected as redundant, if and only if the relationship among its features is already modeled in some other way in the feature model. This non-redundant relationship can be used to explain the redundant constraint. Therefore, the generated CNF from the feature model without the redundant constraint is needed.

2. *What premises are needed to explain redundant constraints?*
   The truth values for features from the redundant constraint must result in a non-satisfiable redundant constraint. This enables BCP to find a contradiction, because information from the redundant constraint is still comprised in the CNF. However, multiple assignments can lead to a non-satisfiable redundant constraint and result in different explanations. Individual explanations may be incomplete. Therefore, we need to consider all possible assignments and join all explanations in order to receive a complete explanation.

We present a feature model containing redundancy in Figure 3.6. The constraint $B \wedge C$ is redundant, because B and C are both core features and will always appear together in every configuration.

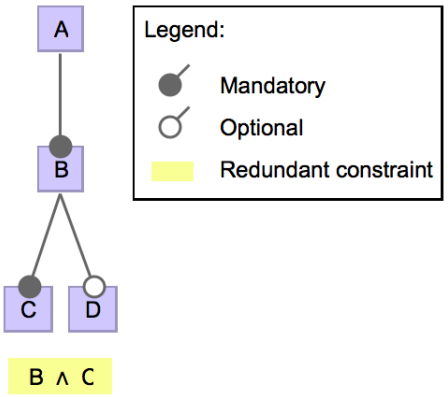

Figure 3.6: Feature model with a redundant constraint $B \wedge C$.

http://publikationsserver.tu-braunschweig.de/get/64218

As illustrated in Table 3.5, three different value assumptions lead to a non-satisfiable redundant cross-tree constraint.

B	C	$B \wedge C$	
0	0	0	1.
0	1	0	2.
1	0	0	3.
1	1	1	

Table 3.5: Truth table of the redundant constraint $B \wedge C$.

Table 3.6 demonstrates the BCP process for all three value assumptions. For the first iteration (#1.1 - #1.4) variable B and C are both bound to *false*. BCP concludes variable D and A to be *false*. The algorithm detects the violated clause (A). Then, BCP restarts the propagation with the second truth value assumption (#2.1 - #2.2). Variable B is set to *false* and C is set to *true*. A direct violation in clause $(\neg C \vee B)$ appears, since all literals are *false*. For the third iteration (#3.1 - #3.2), variable B is set to *true* and C is set to *false*. A direct violation in clause $(\neg B \vee C)$ appears and the algorithm terminates. The resulting explanation comprises all unique reasons and violated clauses from all three propagations: $(A), (\neg A \vee B), (\neg C \vee B), (\neg B \vee C)$

CNF: $(A) \wedge (\neg A \vee B) \wedge (\neg B \vee A) \wedge (\neg B \vee C) \wedge (\neg C \vee B) \wedge (\neg D \vee B)$				
ID	Conclusion	Reason	Antecedents	Stack
#1.1	B=0	premise		
#1.2	C=0	premise		$(\neg A \vee B), (\neg D \vee B)$
#1.3	D=0	$(\neg D \vee B)$	#1.1	$(\neg A \vee B)$
#1.4	A=0	$(\neg A \vee B)$	#1.1	
Violated clause: (A)				
ID	Conclusion	Reason	Antecedents	Stack
#2.1	B=0	premise		
#2.2	C=1	premise		
Violated clause: $(\neg C \vee B)$				
ID	Conclusion	Reason	Antecedents	Stack
#3.1	B=1	premise		
#3.2	C=0	premise		
Violated clause: $(\neg B \vee C)$				
Explanation: *Constraint is redundant, because: A is root (violated clause), B is mandatory child of A (#1.4), C is mandatory child of B (violated clauses).*				

Table 3.6: Explaining a redundant cross-tree constraint.

Although the clause $(\neg D \vee B)$ exists in the BCP process, it is skipped. This is due to the fact that explanations are generated backwards, i.e., depending on the antecedents, BCP traverses the reasons for conclusions backwards to the premises. The conclusion

http://publikationsserver.tu-braunschweig.de/get/64218

for variable A references variable C while ignoring all conclusions in between, i.e., the conclusion for variable D.

3.3.5 Implicit Constraints

Last but not least, we use BCP to explain implicit constraints. To summarize, the slicing algorithm by Krieter et al. [39] extracts a new, so called *sliced*, feature model with less features while preserving dependencies between features (cf. Section 2.3.1). Inputs to the slicing algorithm involve a complete feature model, which shall be used to extract a sliced model, and a set of features to remove. The sliced feature model may contain additional cross-tree constraints, which we refer to as implicit constraints. The detection of implicit constraints requires a comparison between cross-tree constraints of the complete feature model and the sliced model. If a constraint does not appear in the complete feature model, it is implicit.

We put the following thoughts into explaining an implicit constraint: Feature model slicing constructs an implicit constraint by combining two specific clauses. One clause contains a positive form of the literal to remove, while the other contains a negativ form of it. The resulting implicit constraint (resolvent) represents a transitive dependency between the involved clauses. As we describe in Section 2.2.4, transitivity is one of the main causes of redundancy. Therefore, we treat implicit constraints as redundant ones to generate an explanation with BCP. We pass the following input parameters to BCP:

- A complete feature model in CNF. We do not pass the sliced feature model, because an explanation can only arise from the complete feature model due to multiple involved submodels.
- A truth value assignments for features from the implicit constraint leading to the constraint becoming non-satisfiable.

Consider the interrelated feature models in Figure 2.12 resulting in an implicit constraint $C \Rightarrow D$. In order to apply the slicing algorithm, we need to pass a complete feature model. Tools like VELVET [1] support the composition of interrelated feature models. A composition of feature models results in a new abstract root containing the prior roots as child features. In Figure 3.7, we depict the complete feature model.

In Table 3.7, we present the BCP process in order to explain the implicit constraint $C \Rightarrow D$. BCP works in its old manner. First, we pass the feature model in CNF and premises to BCP. Then, BCP collects unit-open clauses and pushes them on stack. BCP concludes G to be *true* and updates all respective literals in the CNF with the truth value. This results in a violated clause ($\neg G \vee D$). Based on the violated clause and the stored reason, BCP generates an explanation.

By varying input parameters, BCP can explain all defects previously described (cf. Section 2.2). Furthermore, the generic explanation algorithm provides links to improve explanations. We reason on improvements in the next section.

[1] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/multiple/modeling.php

http://publikationsserver.tu-braunschweig.de/get/64218

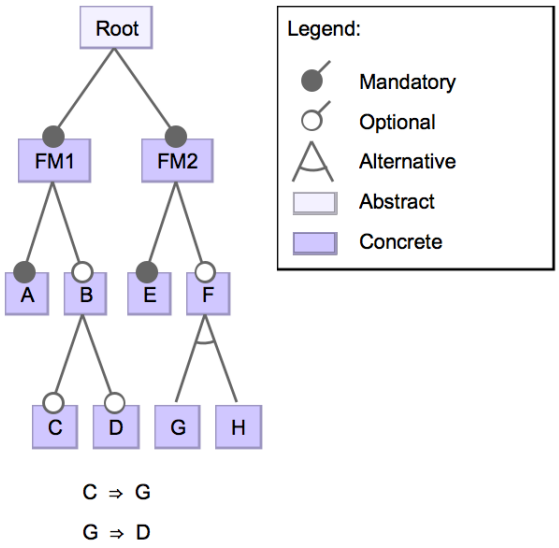

Figure 3.7: A complete feature model containing the feature models in Figure 2.12.

3.4 Improvements of the Basic Algorithm

In order to explain a defect, the BCP algorithm only presents relevant parts to the user bundled in an explanation. However, a defect might have *multiple* explanations, which differ in their size. Finding all possible explanations for a certain defect enables different improvements that we present in the following.

3.4.1 Preferring Shortest Explanations

Since BCP is *order-sensitive*, the algorithm does not always find an explanation with a minimal length. A short explanation offers the advantage of an improved readability for the developer, considering a feature model with thousands of features and corresponding large explanations for defects.
Decoupling the algorithm from its *order-sensitivity* results in examining the clauses of the CNF in every possible order to find a minimal length. A first approach lies in the permutation of all CNF clauses. Running the algorithm on every possible clause order ensures finding a minimal length. However, this approach is not feasible in terms of efficiency [37].

We propose a heuristic that takes advantage of the stack maintaining unit-open clauses. The basic BCP algorithm generates an explanation as soon as a violation occurs, while the stack might still contain unit-open clauses. Restarting the propagation process with those clauses, BCP finds further explanations. However, this approach does not guarantee to find a minimal explanation length.
Consider a simple feature model in Figure 3.8. It contains a cross-tree constraint $E \Rightarrow B$. The constraint is redundant for two reasons: First, it is superfluous due to a transitive dependency between feature E and B. Second, B is a core feature and its implication is unnecessary.

http://publikationsserver.tu-braunschweig.de/get/64218

CNF: $Root \wedge (\neg FM1 \vee Root) \wedge (\neg FM2 \vee Root) \wedge (\neg FM1 \vee A) \wedge (\neg A \vee FM1) \wedge (\neg B \vee FM1) \wedge (\neg C \vee B) \wedge (\neg C \vee B) \wedge (\neg D \vee B) \wedge (\neg FM2 \vee E) \wedge (\neg E \vee FM2) \wedge (\neg F \vee FM2) \wedge (\neg F \vee G \vee H) \wedge (\neg G \vee F) \wedge (\neg H \vee F) \wedge (\neg G \vee \neg H) \wedge (\neg C \vee G) \wedge (\neg G \vee D)$

ID	Conclusion	Reason	Antecedents	Stack
#1	D=0	premise		
#2	C=1	premise		$(\neg G \vee D), (\neg C \vee B), (\neg C \vee G)$
#3	G=1	$(\neg C \vee G)$	#2	$(\neg G \vee D)$**!**, $(\neg C \vee B), (\neg G \vee F), (\neg G \vee \neg H)$

Violated clause: $(\neg G \vee D)$

Explanation: *Constraint $C \Rightarrow D$ is transitive, because: $G \Rightarrow D$ is Constraint (violated clause), $C \Rightarrow G$ is Constraint (#3).*

Table 3.7: Explaining an implicit constraint.

In Table 3.8, we show the BCP process to generate all explanations for the the redundant cross-tree constraint. The first explanation comprises a relation-chain between feature E and B. When BCP terminates, it still contains two unit-open clauses in the stack: $(\neg E \vee D), (\neg A \vee B)$.

CNF: $A \wedge (\neg A \vee B) \wedge (\neg B \vee A) \wedge (\neg H \vee A) \wedge (\neg B \vee C) \wedge (\neg C \vee B) \wedge (\neg C \vee D) \wedge (\neg D \vee C) \wedge (\neg D \vee E) \wedge (\neg E \vee D) \wedge (\neg G \vee D)$

ID	Conclusion	Reason	Antecedent	Stack
#1	E=1	premise		
#2	B=0	premise		$(\neg E \vee D), (\neg A \vee B), (\neg C \vee B)$
#3	C=0	$(\neg C \vee B)$	#1,2	$(\neg E \vee D), (\neg A \vee B), (\neg D \vee C)$
#4	D=0	$(\neg D \vee C)$	#3	$(\neg E \vee D), (\neg A \vee B)$

Violated clause: $(\neg E \vee D)$

Explanation: *Constraint is redundant, because: E is mandatory child of D (violated clause), D is mandatory child of C (#4), C is mandatory child of B (#3).*

Table 3.8: Generating the first explanation for a redundant constraint.

In Table 3.9, we restart the algorithm with the remaining clause $\neg A \vee B$ on stack. BCP removes the clause from the stack and infers *false* for variable A. This immediately results in a violation of clause (A). Consequently, we have found a shorter explanation stating that B is a core feature, because it origins from the root. Executing the algorithm with the last remaining clause $(\neg E \vee D)$ results in the same explanation demonstrated in Table 3.8.

To summarize, both explanations comprise only relevant information leading to the redundant cross-tree constraint. We have shown that BCP can find several explanations

http://publikationsserver.tu-braunschweig.de/get/64218

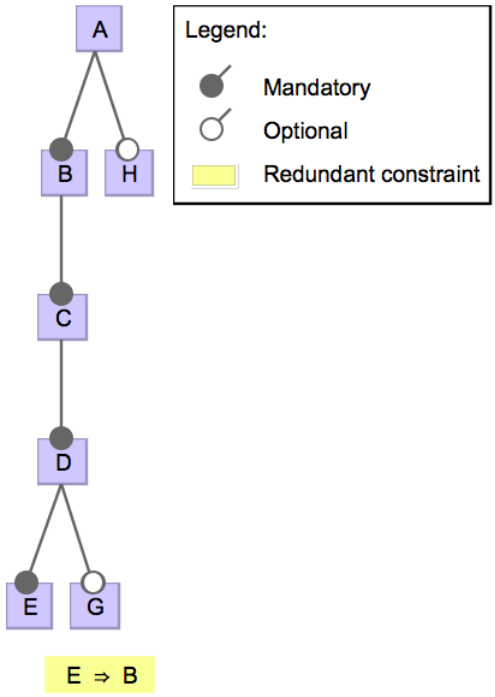

Figure 3.8: Feature model with redundant (transitive) constraint $E \Rightarrow B$.

and even different reasons for redundancy allowing us to prefer the shortest explanation found. We refer to an explanation with an improved length as the *shortest* explanation.

3.4.2 Emphasizing Significant Parts of Explanations

As presented previously, BCP can generate several explanations for different defects. Besides finding and preferring a shortest explanation, we can process this information further. Since every explanation consists of explanation parts either representing a child-parent relationship or a cross-tree constraint, identical explanation parts might occur in multiple explanations. Such common explanation parts are more likely to be the cause of a defect. Hence, editing those parts increases the probability to repair the defect. For instance, changing an explanation part that is not available in all explanations cannot fix the anomaly, since at least one explanation containing a different cause remains for the defect. In Figure 3.9, we present a feature model with three cross-tree constraints along with one redundant constraint. The constraints result in a false-optional feature C. The improved BCP algorithm generates three explanations for the false-optional feature:

http://publikationsserver.tu-braunschweig.de/get/64218

CNF: $A \wedge (\neg A \vee B) \wedge (\neg B \vee A) \wedge (\neg H \vee A) \wedge (\neg B \vee C) \wedge (\neg C \vee B) \wedge (\neg C \vee D) \wedge (\neg D \vee C) \wedge (\neg D \vee E) \wedge (\neg E \vee D) \wedge (\neg G \vee D)$				
ID	Conclusion	Reason	Antecedent	Stack
#1	E=1	premise		
#2	B=0	premise		$(\neg E \vee D), (\neg A \vee B)$
#3	A=0	$(\neg A \vee B)$	#1, #2	$(\neg E \vee D)$
Violated clause: A				
Explanation: *Constraint is redundant, because: A is root (violated clause), B is mandatory child of A (#3).*				

Table 3.9: Generating a shorter explanation for a redundant constraint.

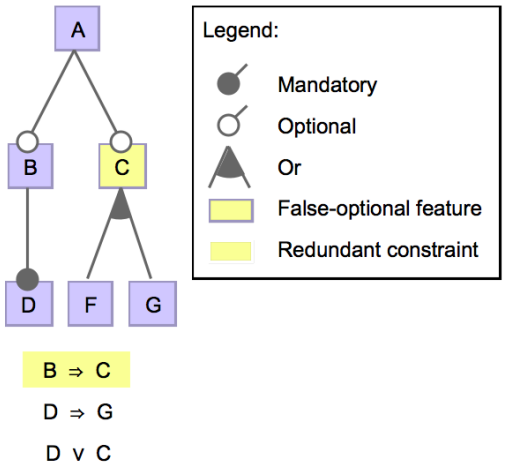

Figure 3.9: Feature model with a false-optional feature C.

1. *G is or child of C*, $D \Rightarrow G$ *is Constraint and* $D \vee C$ *is Constraint.*
2. $D \vee C$ *is Constraint, D is a mandatory child of Band BC is Constraint.*
3. $D \vee C$ *is Constraint,* $D \Rightarrow G$ *is Constraint and G is or child of C.*

After analyzing all three explanations, we conclude that the explanation part "$D \vee C$ *is a Constraint*" is available in every explanation and therefore occurs most often. Removing this constraint repairs the defect. Such information can be visually highlighted within an explanation in order to point out an explanation part, which is more likely to represent the faulty relationship. In Figure 3.10, we present a teaser how the emphasis of core parts in explanations works. By hovering the cursor of feature C, a tool tip appears containing an explanation. Behind every explanation part, numbers indicate how often an explanation part is present in all generated explanations. This is also indicated by a color-intensity, which ranges form black to red. Black explanation parts are only present in one explanation at all, while red parts occur in all explanations.

http://publikationsserver.tu-braunschweig.de/get/64218

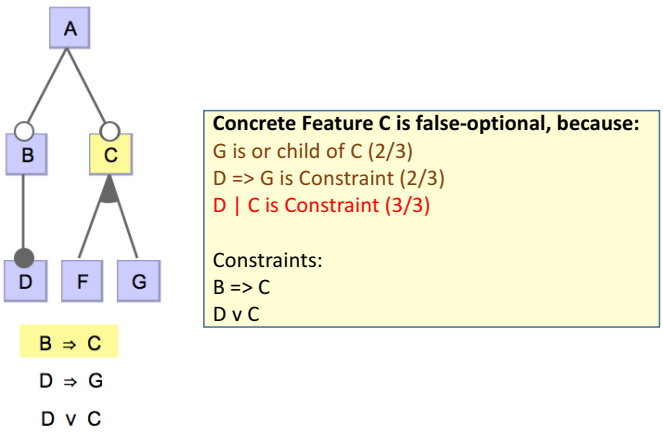

Figure 3.10: Emphasizing significant explanation parts.

3.5 Summary

In this chapter, we presented a generic explanation algorithm in order to explain defects in feature models (**RG1.1**). Therefore, we introduced an LTMS and its internal inference engine BCP (cf. Section 3.2). Only by varying two input parameters for BCP, i.e., a CNF representing a defect feature model and premises (i.e., initial value assumptions for variables of the CNF), the algorithm is able to generate explanations. In Section 3.3, we provided for every previously described defect a minimal use case to show the procedure of BCP along with the resulting explanations.

Additionally to the explanation of defects in a single feature model, i.e., void feature models, dead and false-optional features and redundant cross-tree constraints, we proposed an approach to detect and explain implicit constraints in interrelated feature models (**RG2.1**, **RG2.2**). We applied BCP to explain implicit constraints in the same matter as explaining redundant cross-tree constraints.

Furthermore, we suggested two improvements for explanations (cf. Section 3.4) (**RG1.3**). The first improvement comprised a heuristic in order to find and prefer shortest explanations. The second improvement enhanced the support to repair a defect by highlighting parts in explanations, which most likely caused it.

http://publikationsserver.tu-braunschweig.de/get/64218

http://publikationsserver.tu-braunschweig.de/get/64218

4. Implementation

In the previous chapter, we described the basics of BCP. Additionally, we explained how to apply BCP in order to explain different defects and suggested improvements of the explanation algorithm.
To provide an open-source implementation and, hence, realize **RG1.2** and **RG2.3**, we now focus on the implementation of the explanation algorithm in FeatureIDE. An implementation is further needed to evaluate the scalability of BCP to reach **RG1.4**. In Section 4.1, we describe the general workflow of explanation generation and the architecture. Additionally, we give implementation details for BCP and provide information on how to express explanations in an user-friendly manner. In Section 4.2, we demonstrate the workflow for explaining implicit constraints and the respective architecture. Finally, we provide information about the availability of source code.

4.1 Explaining Defects

In this section, we provide details of the prototypical implementation of BCP in order to explain defects a single feature model, i.e., void feature models, dead and false-optional features as well as redundant cross-tree constraints.

4.1.1 Workflow of the Generation of Explanations

We integrated our explanation approach into the general workflow of the automated analysis in FeatureIDE. The workflow consists of multiple steps as illustrated in Figure 4.1. First, feature modeling takes place (1). After every modification, an analyzer checks the consistency of the feature model (2). During this process, the analyzer might detect a defect (2.1). In this case, BCP is called to explain the defect (2.2). Finally, the system updates the visualization of the feature model and highlights a defect by coloring the faulty feature or constraint. A respective tool tip contains an explanation for the defect (3).

http://publikationsserver.tu-braunschweig.de/get/64218

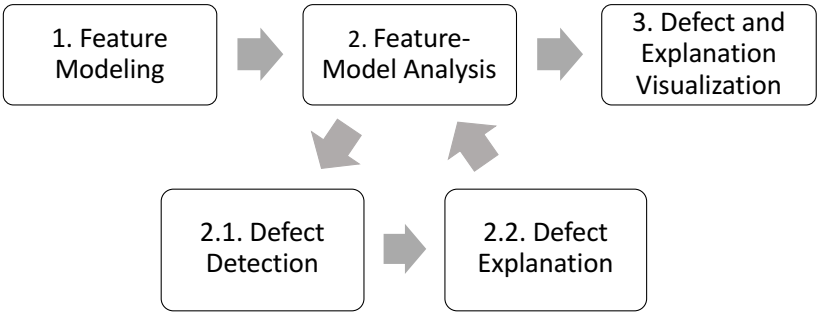

Figure 4.1: General workflow of explanation generation in FeatureIDE.

Figure 4.2 illustrates a detailed view on the workflow between defect detection and explanations: FeatureIDE transforms a feature model into a propositional formula and performs an automatic analysis on it. A SAT solver checks the formula for defects (cf. Section 2.3.1). If defects are detected, the analysis algorithm invokes the BCP algorithm on every defect. BCP receives as input the logical representation of the defect feature model and premises (i.e., initial value assumptions for variables depending on the defect). After executing the BCP algorithm, it returns an explanation for the respective defect.

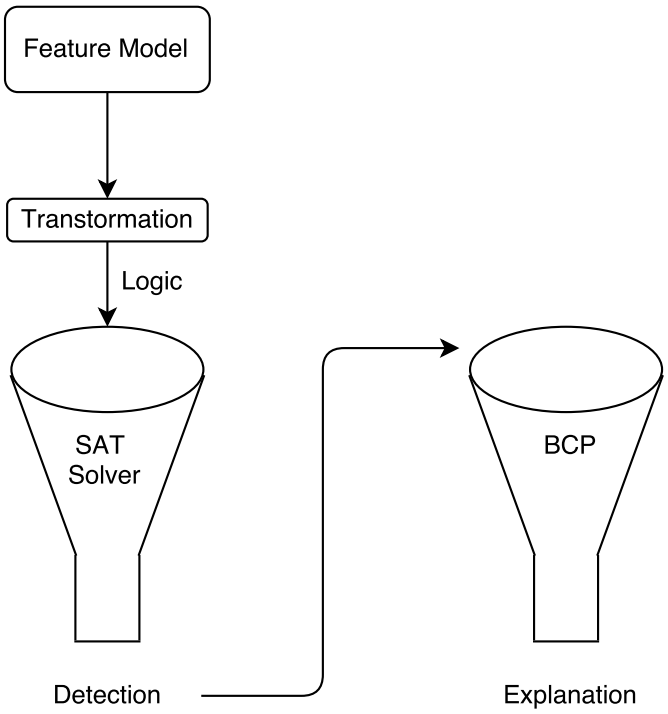

Figure 4.2: Workflow between defect detection and its explanation.

http://publikationsserver.tu-braunschweig.de/get/64218

4.1.2 Architecture of Explanations

For the software design of explanations, we chose a three-layer architecture as presented in Figure 4.3. The highest layer is responsible for defect detection. The component *Feature-Model Analyzer* performs an automated analysis of the feature model. The analyzer is invoked on every modification in a feature model. The middle layer is responsible for preparing the BCP input and error-handling in case of a missing explanation. There is a clear separation between the components *Redundant Constraint*, *Dead Feature* and *False-Optional Feature*. *Redundant Constraint* deals with finding premises for features, which lead to a non-satisfiable redundant constraint. It invokes BCP for the logical formula representing a feature model and premises. Similarly, *Dead Feature* and *False-Optional Feature* invoke the BCP process to explain the respective defect. The lowest layer consists of the component *LTMS*, which uses BCP to generate explanations.

Classes of the middle and lower layer have been added to the FeatureIDE implementation and the *Feature-Model Analyzer* was extended.

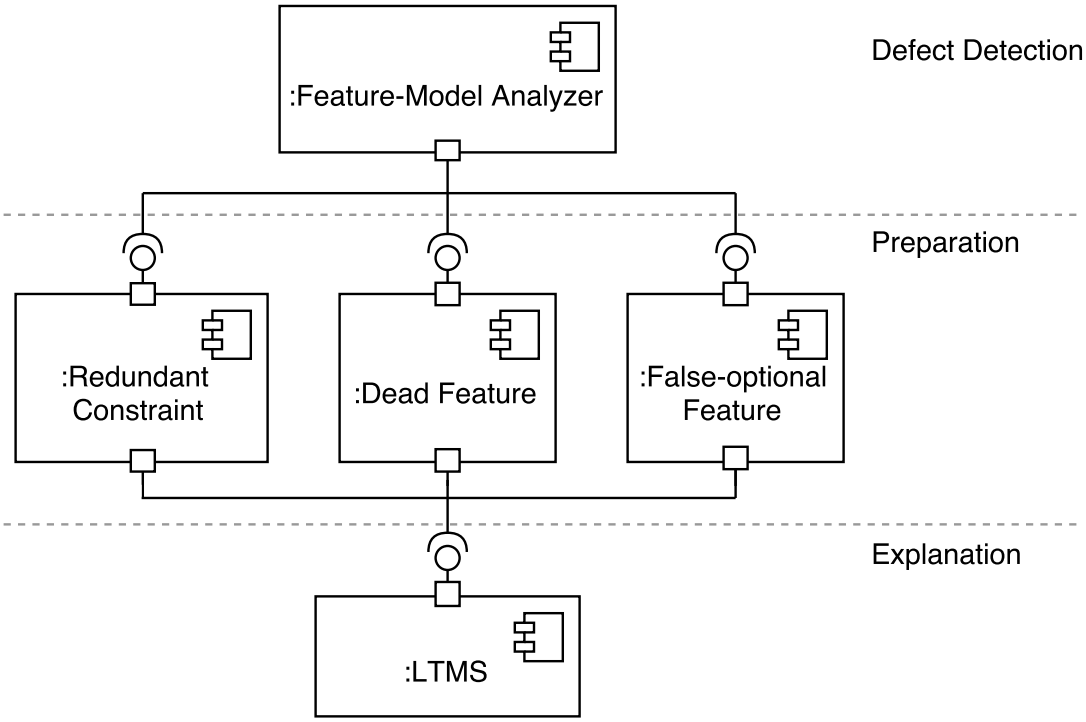

Figure 4.3: Component diagram of explanations in FeatureIDE.

Figure 4.4 depicts relations between classes of the middle and lowest layer. For reasons of clarity, we ignore return types and arguments of methods at this point. Enum types represent modes regarding the defect to explain. The hash map *valueMap* contains literals as keys and instances of the *Bookkeeping* class as values. Within the values of hash map, we store a truth value and reason for a literal. Additionally, *Bookkeeping*

http://publikationsserver.tu-braunschweig.de/get/64218

maintains the literals antecedents and a boolean flag whether the literal's truth value has been a premise. *ExplanationMode* is an enumeration of *LTMS*.

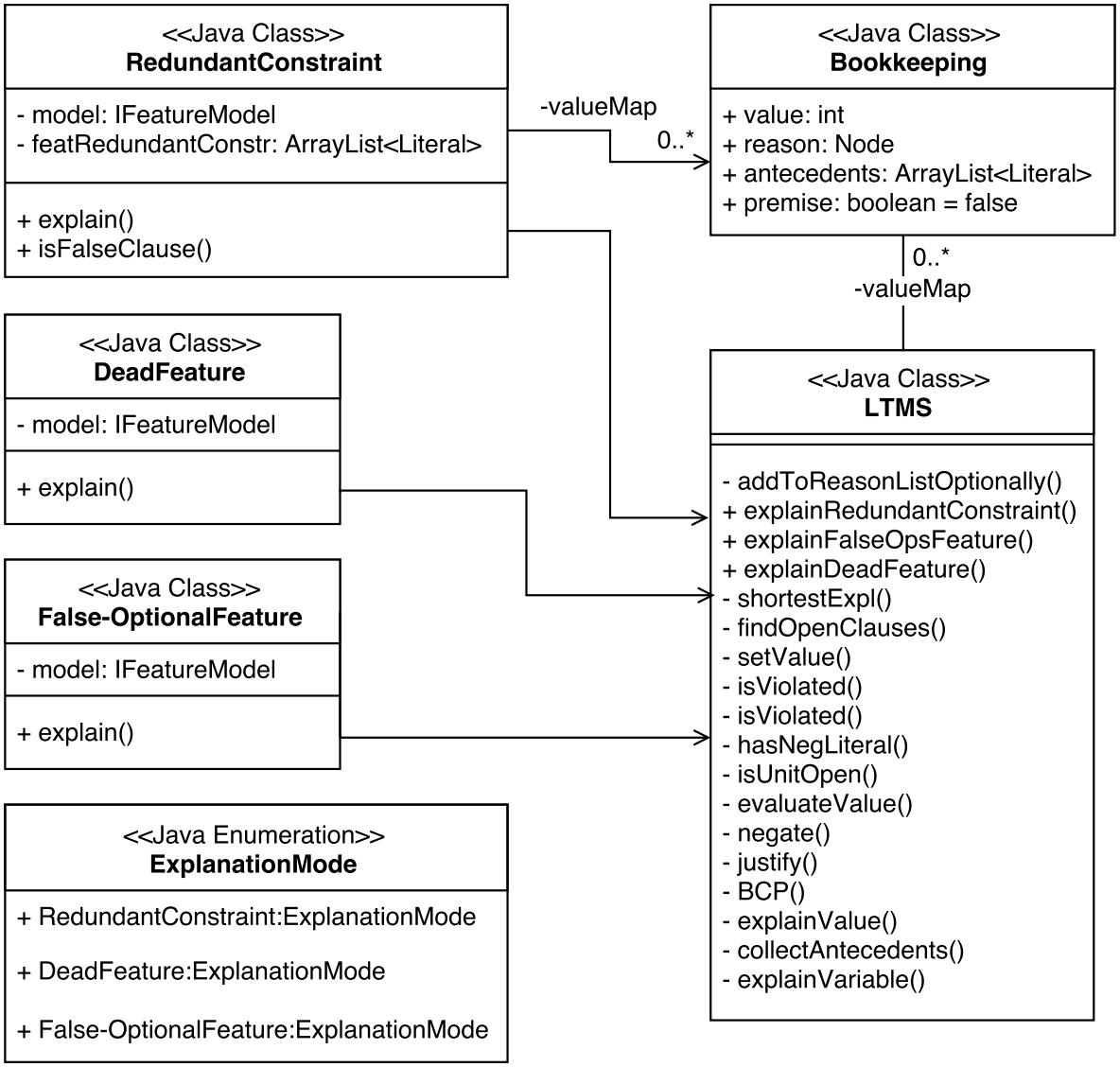

Figure 4.4: Class diagram of explanations in FeatureIDE.

4.1.3 Boolean Constraint Propagation

Figure 4.5 depicts a detailed workflow of the BCP process. If the stack contains unit-open clauses, BCP removes a unit-open clause and determines the unbound literal (1). Then, BCP justifies the literal by maintaining its reason and antecedents (2). Next, the algorithm concludes and sets the literal's truth value. Additionally, it pushes new formed unit-open clauses on stack (3). If no violation occurs and the stack still contains

http://publikationsserver.tu-braunschweig.de/get/64218

unit-open clauses, BPC restarts the propagation process. As soon as BCP detects a violated clause, it generates an explanation (4).

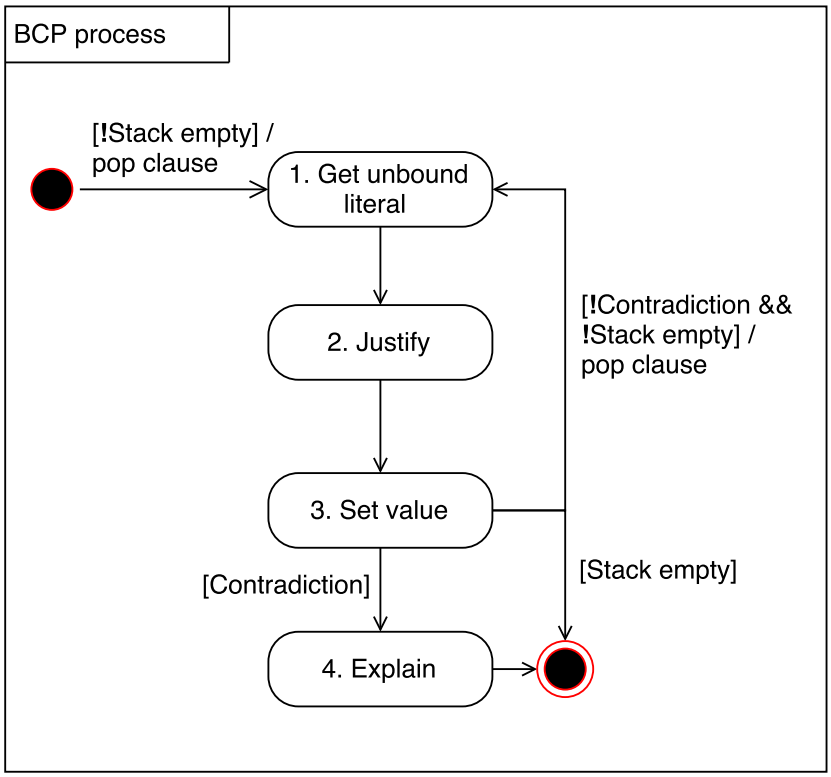

Figure 4.5: State machine of the BCP process.

Figure 4.6 demonstrates the workflow of the third step *Set Value* from Figure 4.5. After deriving a truth value for the unbound literal in an unit-open clause, BCP processes all n clauses of the CNF searching for a new unit-open clause. The explanation algorithm only checks a clause to be unit-open, if the clause contains the newly bound literal evaluated to *false*. This is a necessary criterium, because a unit-open clause consists of one unbound literal while the remaining literals are *false* (cf. Section 3.2.2). The method ends if either a violation occurred or every clause in a CNF is processed without a contradiction.

Figure 4.7 illustrates the fourth step *Explain* from Figure 4.5. The explanation starts with a violated clause A. BCP uses the *valueMap* to retrieve all antecedents of A recursively, i.e., *B, E, D*. For these literals, a respective reason is added to the explanation. Premises are excluded from explanations due to minimizing their length. The resulting explanation in Figure 4.7 consists of pure clauses whereas final explanations are expressed in an user-friendly manner. In Section 4.1.4, we reason on this improvement.

http://publikationsserver.tu-braunschweig.de/get/64218

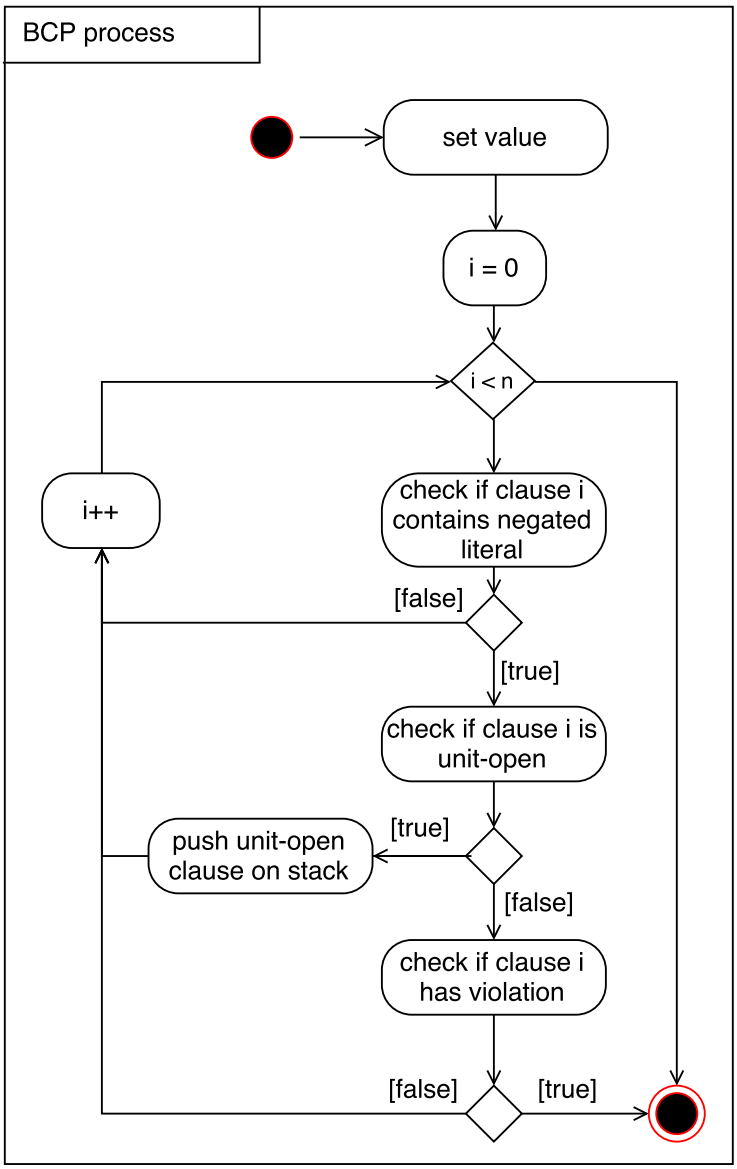

Figure 4.6: Activity diagram of the BCP process.

4.1.4 User-Friendly Explanations

In order to help users to understand defects, explanations must be easy to comprehend. As demonstrated in Figure 4.7, BCP stores pure CNF clauses as *reasons* for a defect. However, their relation to a feature model might not be obvious. Hence, presenting pure logic clauses to the user can decrease the understandability of explanations. Therefore, we need to trace the relations between the feature model and its CNF clauses.

In a feature model, every feature comprises structural information: A feature belongs to the tree topology and may additionally occur in cross-tree constraints. Regarding the tree topology, a feature can take up different roles, i.e., *child* or *parent* and be either

http://publikationsserver.tu-braunschweig.de/get/64218

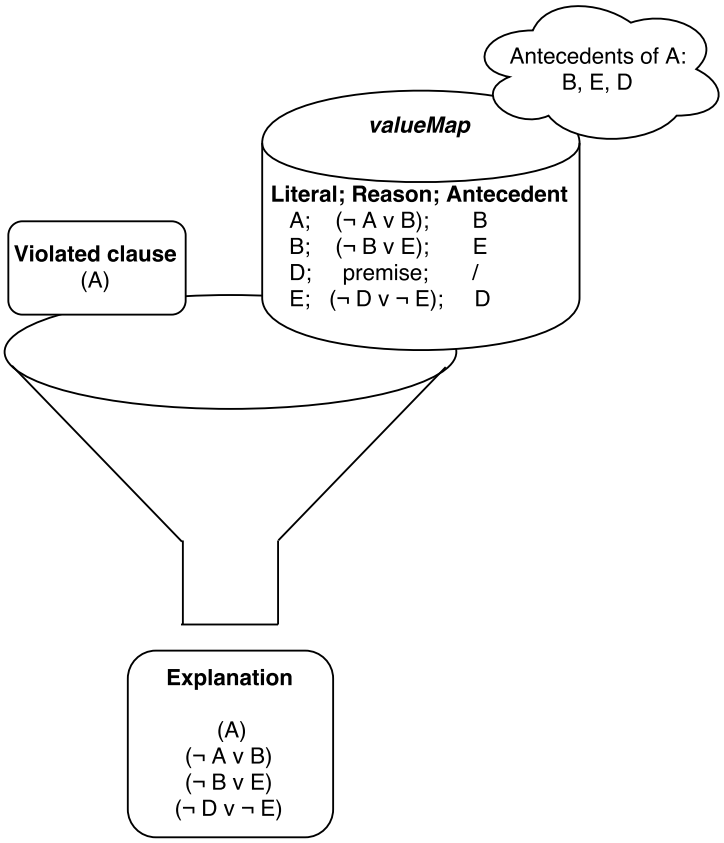

Figure 4.7: Exemplary construction of explanations with pure clauses.

mandatory or *optional* as property. Additionally, it can be contained in *alternative*-groups or *or*-groups. In order to generate user-friendly explanations, reasons can be presented using this information. The tracing of a clause to the feature model comes with some difficulties [37]:

- Transforming the feature model into a CNF results in a one-to-many relationship between a feature and literals, which represent a feature in the formula. Therefore, every literal has to carry its structural information whether it is child or parent in a tree topology or is contained inside a cross-tree constraint.

- This annotation of a literal has to be as efficient as possible, since FeatureIDE operates with literals in many different processes.

The annotation of every literal with structural information takes place during the creation of the propositional formula. FeatureIDE itself already provides means to retrieve the structural information of a feature, which is reused in this process [37]. In the following, we provide implementation details on the annotation of a literal.

http://publikationsserver.tu-braunschweig.de/get/64218

We extended literal objects with an additional numeric attribute *origin*. The attribute *origin* contains relevant information for the generation of explanations [37].
In Figure 4.8, we present a class diagram of the *Literal* class in FeatureIDE. Tracing information is encoded in the literal's attribute *origin* using the enumeration *FeatureAttribute*. A literal has *origin Up*, if it origins from the tree topology and represents a child feature within a clause. In this case, a relationship to the parent feature must be explained. A literal has *origin Down*, if it is created from the tree topology and represents a parent feature. A literal has *origin Root*, if it represents the root feature of the feature model. A literal has *origin Constraint*, if it origins from a cross-tree constraint. In all other cases (usually in error cases), a literal has *origin Undef*.

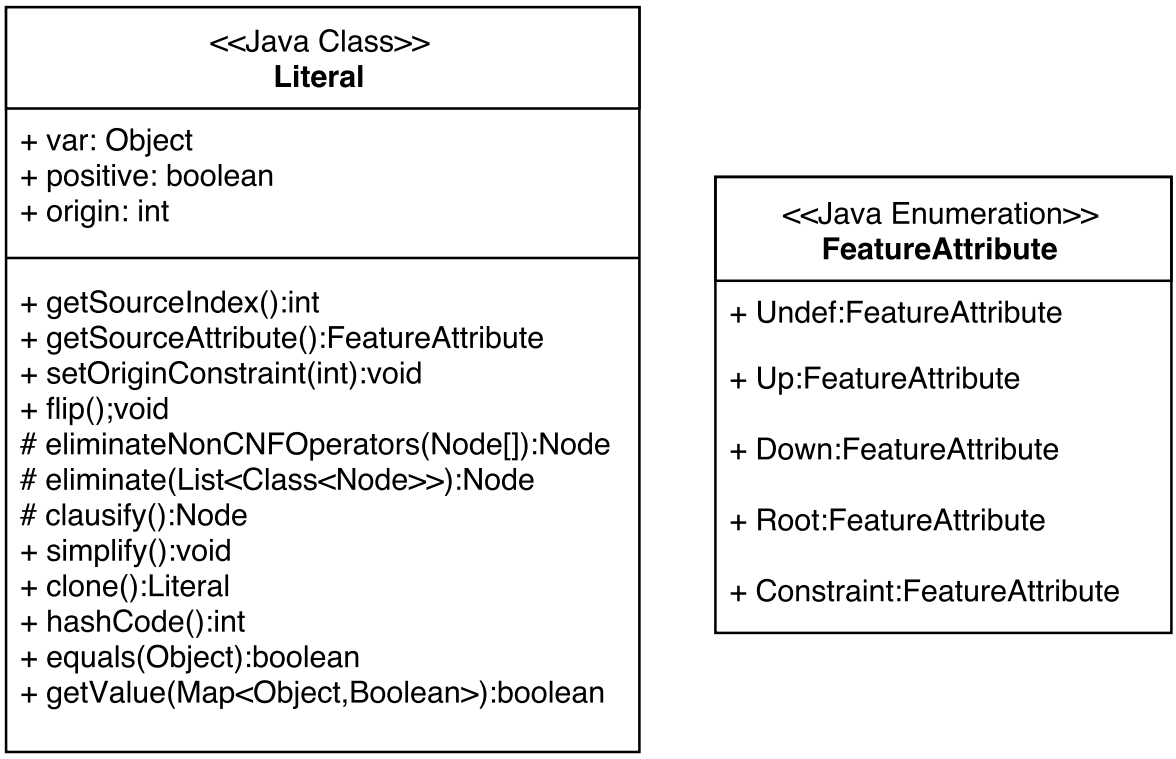

Figure 4.8: Class diagram of class *Literal*

The annotation of a literal takes place during the creation of a propositional formula. A listing in Figure 4.9 presents the encoding of *origin* using the *FeatureAttribute* value (*ordinal*) and optionally a constraint-index. In the first constructor, *origin* encodes information that the literal object belongs to the feature tree topology.

Example: Encoding a root-feature: *origin* = -1 * 5 + 3 = -2

In the second constructor, which calls the method *setOriginConstraint()*, *origin* encodes information that the literal object belongs to a cross-tree constraint and its constraint-index.

Example: Encoding a constraint-index of 0: *origin* = 0 * 5 + 4 = 4

http://publikationsserver.tu-braunschweig.de/get/64218

```
/* Annotate every literal with structural information*/
public enum FeatureAttribute {
        Undef, Up, Down, Root, Constraint
};
/**
 * Encodes a literal from the tree topology.
 * FeatureAttribute must not have the value Constraint.
 * @param var The variable
 * @param FeatureAttribute The Enumeration element
 */
public Literal(Object var, FeatureAttribute a) {
        this(var);
        if (a == FeatureAttribute.Constraint) {
                throw new InvalidParameterException
                ("Parameter Constraint is not allowed");
        }
        this.origin = -1 * FeatureAttribute.values().length
        + a.ordinal();
}
/**
 * Encodes a literal from a constraint.
 * @param var The variable
 * @param constraintIndex The index of a constraint
 */
public Literal(Object var, int constraintIndex) {
        this(var);
        setOriginConstraint(constraintIndex);
}
/**
 * Encodes a constraint-index.
 * @param constrIndex The index of a constraint
 */
public void setOriginConstraint(int constrIndex) {
        this.origin = constrIndex
        * FeatureAttribute.values().length
        + FeatureAttribute.Constraint.ordinal();
}
```

Figure 4.9: Encoding a literal object with an additional numeric attribute *origin* to store its structural information.

A listing in Figure 4.10 demonstrates getter-methods, which decode *origin*. In method *getSourceIndex()*, a constraint-index is retrieved from *origin*.

http://publikationsserver.tu-braunschweig.de/get/64218

Example: Decoding a constraint-index of 4: *origin* = 4 / 5 = 0
Returns a constraint with index 0.

In the method *getSourceAttribute()*, a *FeatureAttribute* is retrieved representing the structural information of a literal.

Example: Decoding a root feature with *origin* of -2: -2 % 5 + 5 = 3
Returns a *FeatureAttribute* with value 3 (*Root*).

BCP decodes *origin* for every passed literal and generates explanations with respect to the enumeration element and a property of the respective feature, e.g., *D is mandatory child of B* or $B \implies C$ *is Constraint*. Notice that using the enumeration element *Down* may be ambiguously due to multiple children and result in superfluous explanation parts. Therefore, we currently restrict our use to the Enumeration element *Up* and *Root*, if dealing with a literal from the tree topology to ensure unambiguousness.

Listing 4.1: Decoding *origin*

```
/**
 * Decodes a constraint-index.
 * @return origin The index of a constraint
 */
public int getSourceIndex() {
        if (getSourceAttribute() !=
        FeatureAttribute.Constraint) {
                throw new InternalError
                   ("origin is not Constraint");
        }
        return origin / FeatureAttribute.values().length;
}

/**
 * Decodes a FeatureAttribute.
 * @return FeatureAttribute The Enumeration element
 */
public FeatureAttribute getSourceAttribute() {
        int index = origin % FeatureAttribute.values().length;
        if (index < 0) {
                index += FeatureAttribute.values().length;
        }
        return FeatureAttribute.values()[index];
}
```

Figure 4.10: Decoding an additional numeric attribute *origin* of a literal object to retrieve its structural information.

http://publikationsserver.tu-braunschweig.de/get/64218

4.2 Implicit Constraints

In this section, we provide details of a prototypical implementation to derive, explain and visualize implicit constraints in interrelated feature models. Therefore, we demonstrate the workflow and architecture in FeatureIDE.

4.2.1 Workflow

In Figure 4.11, we demonstrate an user view on the calculation and visualization of implicit constraints for a submodel. The user selects a feature FM1 of the complete feature model (presented on the left side) and chooses an action from a context menu to calculate hidden dependencies between the submodel with the root FM1 and the remaining features of the complete feature model. A new dialog appears containing the sliced model (illustrated on the right side). We emphasize an implicit constraint with a red border.

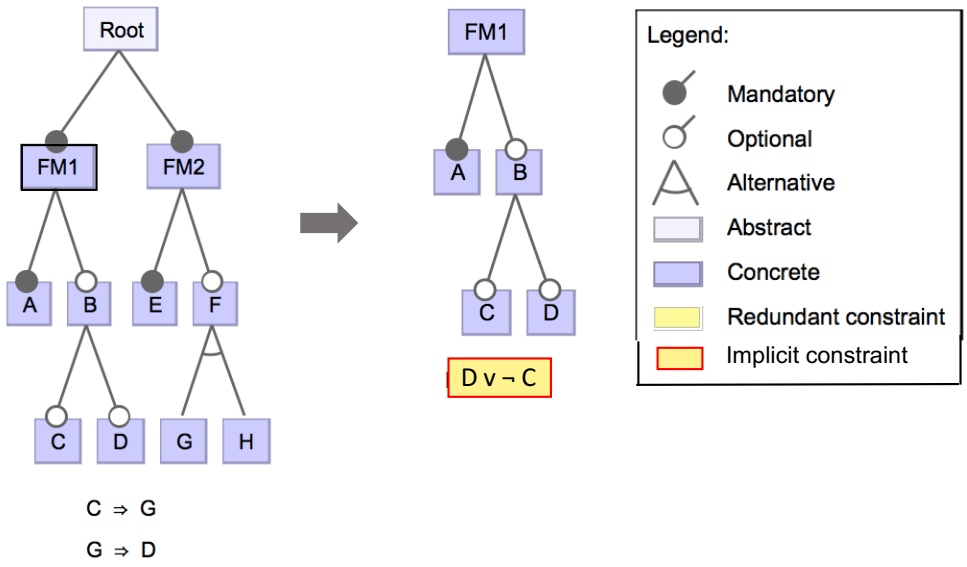

Figure 4.11: User view on calculating and visualizing implicit constraints.

In Figure 4.12, we demonstrate a detailed workflow of processing an user request to calculate hidden dependencies. First, an automated analysis of the sliced feature model takes place. The analysis comprises the detection of defects in a single feature model, i.e., void feature model, dead and false-optional features as well as redundant constraints, its explanation and visualization. Second, an automated detection, explanation and visualization of implicit constraints takes place.

http://publikationsserver.tu-braunschweig.de/get/64218

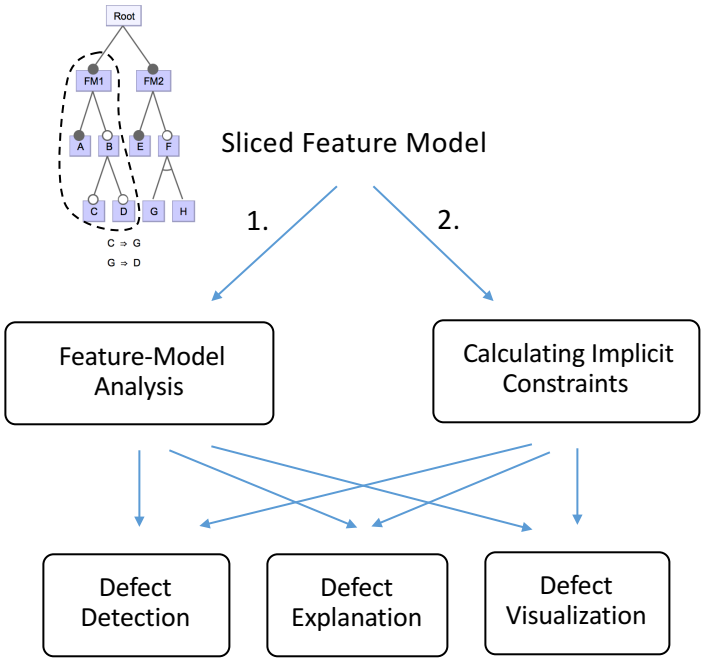

Figure 4.12: General workflow of handling implicit constraints in FeatureIDE.

4.2.2 Architecture

We demonstrate a dynamic view on calculating and displaying implicit constraints in Figure 4.13. After an user chooses to calculate hidden dependencies, the system invokes the action *CalculateDependencyAction*. The action checks if the selection of a new root is valid, i.e., it is not empty and not the root of the complete feature model. On success, it calls the operation *CalculateDependencyOperation*. The operation performs feature model slicing. If the chosen root feature is a core feature, it represents the new root in the sliced feature model. In all other cases, we need to introduce an artificial root of the sliced model to preserves the consistency. This artificial root contains the selected feature as an optional child feature. Next, the operation instantiates a new wizard *SubtreeDependencyWizard*. The wizard creates a *SubtreeDependencyPage* object displaying the sliced feature model and automatically running the model analysis. Then, the operation opens a new dialog containing the wizard page.

In Figure 4.14, we depict a static view on packages involved in calculating implicit constraints. Packages of the plugin *de.ovgu.featureide.fm.ui* provide an user interface and react to actions initiated by an user. The packages perform an action to calculate hidden dependencies and, hence, provide a sliced feature model with emphasized implicit constraints. For analysis purposes and explanation capabilities, the system makes use of FeatureIDE's plugin *de.ovgu.featureide.fm.core* and the referenced packages.

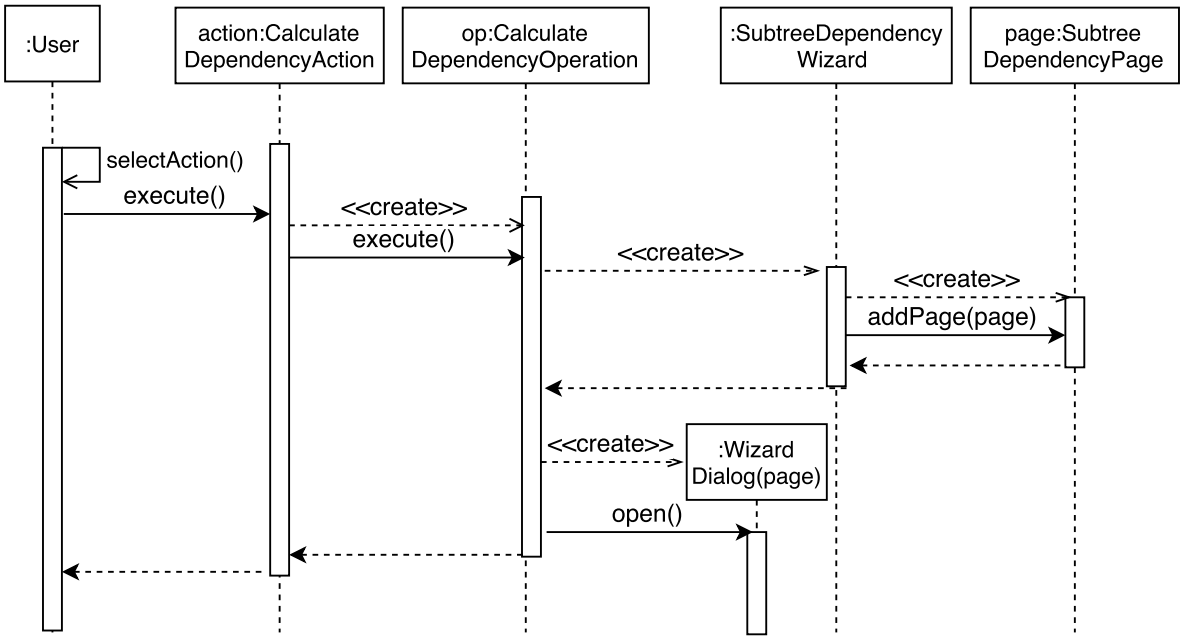

Figure 4.13: Sequence diagram for calculating and displaying implicit constraints.

4.3 Summary

In this chapter, we presented insights of our open-source implementation of the explanation algorithm (**RG1.2**, **RG2.3**). As soon as the feature model analysis detects a defect, it starts an explanation process. A tool tip contains the explanation for a defect feature or cross-tree constraint. A three-layer-architecture for explanations is used: The highest layer covers defect detection, the middle layer comprises the preparation of BCP input and the lowest layer contains the BCP algorithm.

In order to create user-friendly explanations, we annotated every literal with a numerical attribute during CNF creation. The attribute encodes structural information of the literal: It either belongs to the feature-tree (representing a child or a parent feature) or to a certain cross-tree constraint. Additionally, an explanation comprises a feature's property, i.e., whether it is *mandatory* or *optional* and if it is contained in an *alternative-group* or in a *or-group*.

Furthermore, we explained the explanation of implicit constraints on implementation level. The user initiates an action to calculate hidden dependencies in a submodel by selecting a feature from the complete feature model, which becomes the root of the sliced feature model. On success, an automated analysis is performed on the submodel and, consequently, a detection, explanation and visualization of implicit constraints.

The source code is available on GitHub[1]. Explanations are created within FeatureIDE's plugin *de.ovgu.featureide.fm.core*. The package *de.ovgu.featureide.fm.core.explanations*

[1]https://github.com/FeatureIDE/FeatureIDE/tree/explanations/

http://publikationsserver.tu-braunschweig.de/get/64218

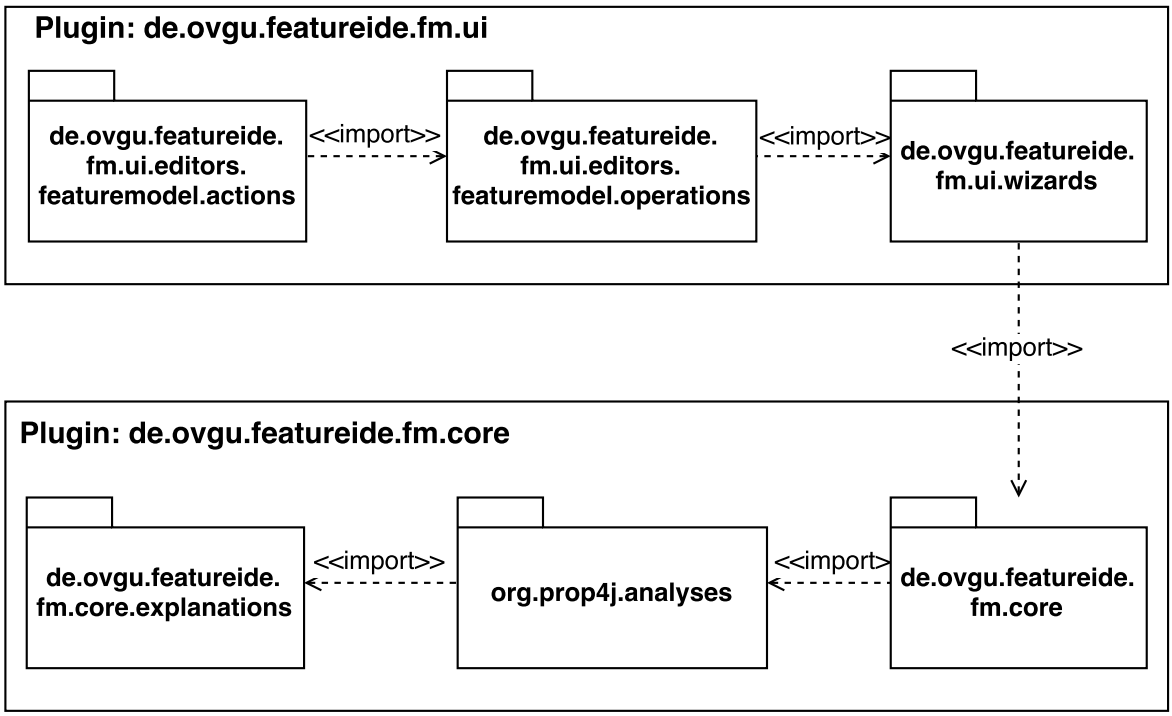

Figure 4.14: Package diagram for calculating implicit constraints.

comprises the source code for explaining defects in single feature models. Within plugin *de.ovgu.featureide.fm.ui*, the system reacts to an user's request to calculate hidden dependencies and visualize those.
A prototypical implementation is part of the major FeatureIDE 3.1.0 release.

http://publikationsserver.tu-braunschweig.de/get/64218

5. Evaluation

In the last two chapters, we described the BCP algorithm, its adaption to explain defects in feature models and provided insights on the implementation. To provide a proof of concept and inspect the algorithm, we need to evaluate the explanation algorithm for all described kinds of defects (cf. Section 2.2) from a qualitative and quantitative viewpoint. Additionally, differently sized feature models serve to evaluating the scalability of the generic algorithm to realize **RG1.4**.
Therefore, we first present evaluation goals in Section 5.1. In Section 5.2, we introduce two use cases representing real-world examples for the evaluation [21]. Next, we perform a qualitative analysis in Section 5.3 covering the correctness of BCP and inspecting the understandability of the resulting explanations. Furthermore, we provide multiple test feature models for every defect. In Section 5.4, we perform a quantitative analysis and concentrate on measurable data of explanations for differently sized feature models, e.g., performance impact of the generation of explanations and their length. We reason on the validity of the results in Section 5.5. Eventually, we summarize all findings in Section 5.6. Parts of the evaluation are available in [37].

5.1 Evaluation Goals

To evaluate the explanation algorithm and the resulting explanations from a qualitative and quantitative viewpoint, we need to inspect different characteristics of the explanation approach. To asses the quality of explanations and, hence, achieve **RG1.3**, we analyze the understandability and correctness of the BCP algorithm. We consider an explanation as understandable, if the editing or removal of the explanation parts fixes the defect. Additionally, we inspect the usability of emphasizing specific parts in an explanation. Last but not least, we also analyze the correctness of the detection and explanation of implicit constraints thus realizing **RG2.1** and **RG2.2**. Consequently, we aim to answer the following *research questions* (RQ):

RQ1 *Do explanations contain necessary information to understand the defect?*

http://publikationsserver.tu-braunschweig.de/get/64218

RQ2 *Does the explanation algorithm produce the correct output?*

To assess quantitative results of the explanation algorithm, we focus on its performance impact on the system's state prior to the generation of explanations. Using differently sized models, we test the scalability of the approach to realize **RG1.4**.

RQ3 *What is the performance impact of the algorithm for generating a first explanation, searching for the shortest one, and highlighting significant parts of the explanation?*

A significant factor for the usability of explanations is its length. We inspect the average explanation length for all defects in differently sized feature models. Based on that, we determine to what extent an explanation is able to isolate a defect. Additionally, we pay special attention to the shortening of explanations in order to asses **RG1.3**.

RQ4 *What is the average length of the shortest explanation for all kind of defects?*

RQ5 *To what extent does an explanation isolate the defect in the feature model?*

RQ6 *How often is the first explanation already the shortest one?*

RQ7 *What is the average size difference between the first and the shortest explanation?*

Unlike explaining defects in single feature models (i.e., a void feature model, dead and false-optional features and redundant cross-tree constraints), the generation of explanations for implicit constraints additionally interacts with the user and executes the slicing operation on a feature model. Hence, we focus on the performance impact to explain implicit constraints in particular. To evaluate the scalability and, consequently, realize **RG1.4**, we use a large real-world feature model. By inspecting implicit constraints and the corresponding explanations, we retrieve further information concerning the structure of such constraints and are able to reason on the percentage share of features from the observed submodel (local features) and features from adjacent submodels in an explanation.

RQ8 *What is the calculation time of explaining implicit constraints in general and of the slicing operation in particular?*

RQ9 *What is the percentage of local features in an implicit constraint?*

RQ10 *How frequently do different kinds of implicit constraints occur in a real-world feature model?*

Before answering this questions, we present two real-world case studies, which are part of both qualitative and quantitative evaluation.

http://publikationsserver.tu-braunschweig.de/get/64218

5.2 Case Studies

We use two variant-rich real-world feature models among further evaluated models. Both case studies have been chosen from Feldmann et al. and illustrate an interdisciplinary product line in the machine manufacturing domain [21]. Both studies comprise interrelated feature models based on the *customer's* and the *developer's point of view*. The customer's point of view allows for modeling product configurations. The developer's point of view allows for modeling various disciplines which contribute to the product, e.g. mechanics, electronics, and software engineering. Relations between discipline-specific components connect the various feature models. *Velvet* composes interrelated feature models into a complete feature model, which allows to perform an automated analysis in FeatureIDE [60].

Pick-And-Place Unit

The first case study involves a "Pick-And-Place Unit" (PPU), a real-world automation system with 52 features [32]. It originates from the Institute for Automation and Information System of the Technical University Munich. The PPU comprises two handling systems using a vacuum gripper. The first system contains a cylinder while the second system makes use of a changeover arm. Hence, the PPU allows for different configuration options, e.g., a cylinder or a changeover arm as handling system, the size of processed work pieces, positioning capabilities and environment conditions.

The PPU comprises a feature model from the customer's point of view and multiple discipline-specific feature models from the developer's point of view [21]. The disciplines mechanics, electronics and software are involved in the development process of the PPU and provide one view of the system. Depending on the discipline, variability differs. The mechanical subsystem only provides *Lifting/Lowering* as a variable part. In electrical engineering and software engineering, many different variants exist, e.g. an optional emergency stop button or additional functionalities such as self-healing capabilities of the PPU. Feldmann et al. provide a mapping matrix between features of the customer's point of view and the developer's point of view which represents dependencies between the different feature models [21].

Sorting Line

The second case study is a Sorting Line with four variants and 39 features [21]. The Sorting Line consists of 5 individual stations, i.e. Distribution, Inspection, Handling, Separating, Sorting. It aims at sorting work pieces into up to three chutes dependent on their color or material. A single feature *distribute work pieces* is mandatory since work pieces are initially required. Further optional features exist, such as *count work pieces* or *separate work pieces*.

Identically to the PPU, the Sorting Line comprises a feature model from the customer's point of view and multiple feature models from the developer's point of view. Modeling mechanics includes all individual stations and assigned *chutes*. In electrical engineering,

http://publikationsserver.tu-braunschweig.de/get/64218

switches are modeled as components for *pneumatics* in order to push work pieces into corresponding chutes. Furthermore, electrical engineering comprise a *motor of conveyor belt*, *sensors*, which serve for the detection of the color or material and a *light barrier*, which detects work pieces. In software engineering, the features *counting work pieces* and *sorting* are defined.

We used both the PPU and the Sorting Line as real-world feature models for a qualitative and quantitative evaluation for explanations. Additionally, we extended the PPU feature model in order to contain implicit constraints.

5.3 Qualitative Analysis

We perform a qualitative analysis to assess the understandability of resulting explanations and the correctness of BCP. This is a necessary step to inspect the quality of explanations (**RG1.3**) and provide a proof of concept of the generic explanation algorithm (**RG1.1**). To perform a qualitative analysis, we created a database containing multiple test feature models per defect. We aimed to cover as many different causes for a specific defect as possible to answer **RQ1** and **RQ2**. For every feature model input, we analyzed the BCP output concerning its correctness and the understandability of a resulting explanation. In Chapter A, we provide small sized test models per defect together with its explanation in FeatureIDE. In the following, we present a subset of the test models which serve as typical examples for explanations. Feature models displayed in the referenced figures represent the input of the system. A respective yellow tool tip containing an explanation forms the output of the system. All test models are contained on a supplied CD-Rom and provided online[1].

In Figure 5.1, we present a feature model with a dead feature E, because it is mutually exclusive to core feature C. An explanation for the defect includes that C is a mandatory child of root A (i.e., a core feature) and is mutually exclusive to feature E. Removing or editing only the relationships expressed by emphasized explanation parts fixes the defect, e.g., making feature C an optional child feature of A.

In Figure 5.2, we present a feature model with a false-optional feature C. The optional feature C becomes false-optional, because it is parent of an alternative child D, which is implied by core feature B. Consequently, feature D and its parent feature C also become core features. This leads to feature C not being optional anymore. The algorithm generates one explanation containing the faulty relationships. A removal or an edit of this connections could fix the defect. For example, either by removing the implication or changing feature C to mandatory.

[1] https://www.isf.cs.tu-bs.de/data/TestFeatureModels.zip

http://publikationsserver.tu-braunschweig.de/get/64218

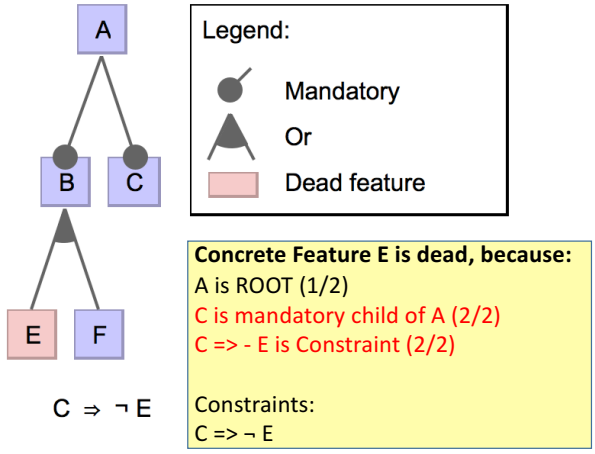

Figure 5.1: Example of an explanation for a dead feature.

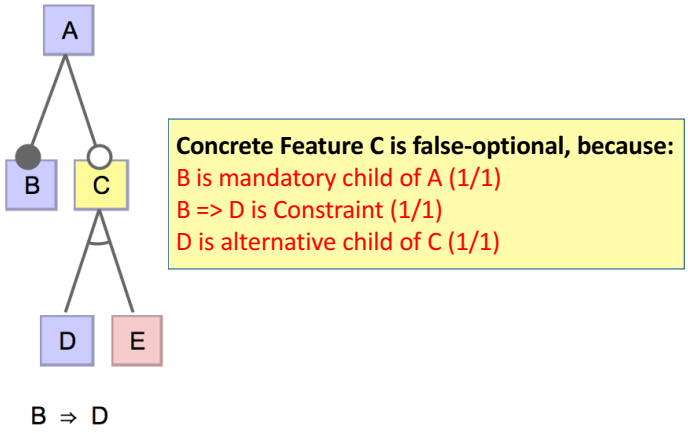

Figure 5.2: Example of an explanation for a dead feature.

Figure 5.3 shows a feature model with the redundant cross-tree constraint $\neg(C \wedge E)$. The constraint is redundant, because feature B is parent of feature C and mutually exclusive to feature E as well. Therefore, the exclusion of feature E by feature C is superfluous. The explanation reveals this connections. Removing the most emphasized explanation part -(B & E), which occurred in all generated explanations for the defect, fixes it.

We conclude that the presented explanations including all provided test models contain the necessary information to fix the defect by performing corrective actions on the relationships involved in an explanation, satisfying **RQ1**. For every test model, BCP performes the process of finding unit-open clauses and detecting a contradiction successfully (cf. Section 3.3). Consequently, we conclude that BCP generates the correct output answering **RQ2**.

http://publikationsserver.tu-braunschweig.de/get/64218

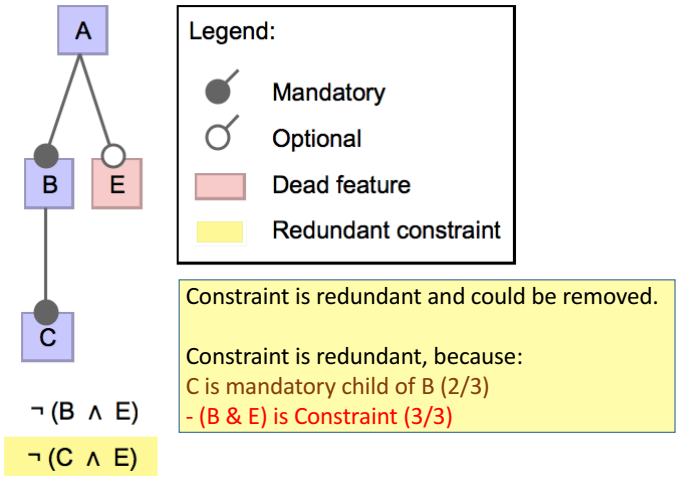

Figure 5.3: Example of an explanation for a redundant cross-tree constraint.

5.4 Quantitative Analysis

To answer the remaining questions concerning a quantitative evaluation, we focus on performance measurements and inspect the length of shortest explanations to answer **RQ3** and **RQ4**. Additionally, we investigate how often a shortest explanation is found immediately and how much shorter it is compared to the first explanation (**RQ6**, **RQ7**). Regarding the length of an explanation, we can further reason on the ability of an explanation to isolate a defect (**RQ5**). To answer **RQ8** - **RQ10**, we measure the performance for explaining implicit constraints and inspect their logical structure to determine the most common one. Additionally, we calculate the number of involved submodels and local features in an implicit constraint to analyze how many submodels participate in a hidden dependency and whether such dependencies are mostly caused by local features or by features from other submodels.

5.4.1 Results

Table 5.1 depicts evaluated feature models and the contained number of redundant constraints, dead and false-optional features. The first two real-world feature models *PPU* and *Sorting Line* origin from the case study (cf. Section 5.2). Furthermore, we use generated feature models consisting of 200, 500, 1,000 and 2,000 features with a growing number of constraints and defects for performance measurements. The generated models are freely available at the FeatureIDE website and have been chosen randomly for different model sizes. An essential requirement for generated models comprised the coverage of all defects per model. A feature model from the automotive industry represents the biggest feature model with 2,513 features and 2,833 constraints and is our third real-world example. It is provided as an example within FeatureIDE named *Automotive1*. We refer to this model as *automotive*.

The computation time has been measured using an Intel(R) Core(TM) i7-4800MQ CPU with 2.7 GHz and 16-GB RAM.

http://publikationsserver.tu-braunschweig.de/get/64218

Model	# Features	# C	# RC	# Dead features	# FO Features
Sorting Line*	39	11	2	0	0
PPU*	52	15	7	0	0
200-model	200	20	8	106	13
500-model	500	50	14	262	56
1000-model	1,000	100	44	628	138
2000-model	2,000	200	87	1,236	254
Automotive*	2,513	2,833	563	192	12

Table 5.1: Overview of evaluated feature models. C = Constraints, RC = Redundant cross-tree constraints, FO = false-optional.

Table 5.2 shows the computation times for all performance measurements. First, each feature model was analyzed in FeatureIDE without the generation of explanations (2nd column). Next, we measured the computation time for defect detection including the tracing process of a literal object to determine the cost of tracing to feature model elements (3rd column). Performance for generating explanations is divided into three individual steps to separately assess its performance impact on the system. In the first step, we focus on the generation of a first explanation for all defects in a feature model (4th column). In the second step, the algorithm tries to find shortest explanations (5th column). In the third step, explanation parts are colored based on their occurrence (6th column). Colored explanations require the generation of a first and a shortest explanation. All measurements have been repeated ten times for all models to reduce computation bias. As presented in Table 5.1, the models contained different kinds of defects. We present the average performance time in Table 5.2.

Model	No Expl.(s)	Tracing(s)	1. Expl.(s)	Sh. Expl.(s)	Col. Expl.(s)
Sorting Line*	0.03	0.03	0.03	0.05	0.06
PPU*	0.02	0.02	0.05	0.05	0.05
200-model	0.37	0.40	0.87	1.21	1.28
500-model	5.08	5.53	9.67	11.42	11.65
1000-model	43.42	46.41	88.77	116.77	116.96
2000-model	352.61	372.89	567.27	831.71	832.11
Automotive*	6,453.30	7,421.96	16,473.90	16,540.66	16,546.44

Table 5.2: Performance measurements for the generation of of a first explanation, a shorter one and a colored one. $Expl.$ = Explanation, $Sh.$ = Shortest, $Col.$ = Colored.

In Table 5.3, we summarize the performance impact for the generation of explanations. For every generation step of explanations, we compute the time factor compared to computation time which only included the detection of defects. The tracing process increases the computation time by factor 1.1 on average. To generate the first explanation, the computation time takes approximately twice as long. To find the shortest

*Real-world feature models

http://publikationsserver.tu-braunschweig.de/get/64218

explanation, performance approximately decreases by factor 2.5. The final coloration step of explanation takes on average factor 2.6. We discuss the results in Section 5.4.2.

Model	Tracing(f)	1. Expl.(f)	Shortest Expl.(f)	Col. Expl.(f)
Sorting Line*	1 (min)	1 (min)	1.7 (min)	2 (min)
PPU*	1 (min)	2.5 (max)	2.5	2.5
200-model	1.1	2.4	3.3 (max)	3.4 (max)
500-model	1.1	1.9	2.3	2.3
1000-model	1.1	2	2.7	2.7
2000-model	1.1	1.6	2.4	2.4
Automotive*	1.2 (max)	2.5 (max)	2.6	2.6
-	1.1 (avg)	2.3 (avg)	2.5 (avg)	2.6 (avg)

Table 5.3: Summary on the performance impact for the generation of explanations. *Expl.* = *Explanation*, *Sh.* = Shortest, *f* = factor, *Col.* = Colored.

To make assumptions about the average length of explanations and investigate how beneficial to search for a shortest one, we analyze the statical distribution of all explanation lengths. The length of an explanation is measured in the number of its *parts*. A *part* is a building block of an explanation representing one single reason (cf. Section 3.2) being expressed in user-friendly language and placed in one separate line, e.g., *C is mandatory child of B*.

Figure 5.4 illustrates the explanation length for defects in the feature models *Sorting Line* and *PPU*. In the left model, the box plot reveals that explanation length lies constantly at 5 parts. In the right model, 50 % of the data is concentrated between 4 and 9 parts for an explanation, having a minimal explanation length consisting of 2 parts and a maximum explanation length with 10 parts.

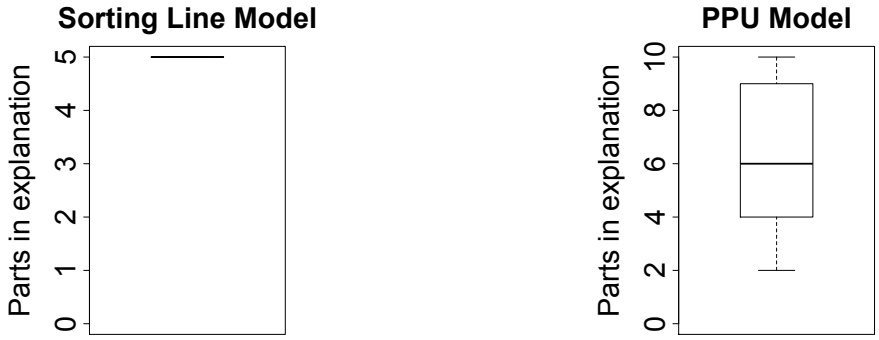

Figure 5.4: Explanation length for *Sorting Line* and *PPU*

In Figure 5.5, a box plot represents explanation length for defects in the *200-model* and the *500-model*. In the left model, 50% of explanations consists of 4 to 7 parts.

*Real-world feature models

http://publikationsserver.tu-braunschweig.de/get/64218

The shortest explanation found contains 2 parts whereas the longest explanation is at 11 parts. Additionally, outliers exists revealing an explanation length of maximum 13 parts. in the right model, 50% of explanations consists of up to 7 parts. A minimum explanation consists of 2 and the longest explanation of 33 parts.

Figure 5.5: Explanation length for the *200-model* and *500-model*

Figure 5.6 illustrates length of explanation's of the *1000-model* and the *2000-model*. Half of the explanations in the left model consists of approximately 8 to 12 parts. A minimum explanation is of length 2. Ignoring outliers, a maximum length is at 21 parts. In this model, multiple outliers exist between explanation length of 20 parts and 40 parts. The longest explanation consists of 61 parts. The right model reveals that 50% of all explanations have a length between 8 and 27 parts. A shortest explanation consists of 3 parts. Without considering outliers, a the longest explanation has 50 parts. Multiple outliers exist. Considering the outliers, the longest explanation consists of 65 parts.

Figure 5.6: Explanation length for the *1000-model* and *2000-model*

In Figure 5.7, a box plot represents explanation length for the *automotive model*. Half of explanations consists of approximately 4 to 25 parts. Several outliers occur beginning

http://publikationsserver.tu-braunschweig.de/get/64218

from 55 explanation parts. The minimum explanation length consists of 1 part, while the maximum explanation length is at 95 parts.

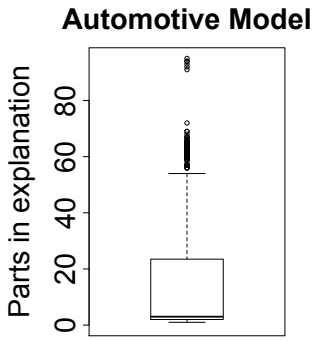

Figure 5.7: Explanation length for the *automotive model*

To draw conclusions about fault localization in general and about the number of the feature model's relationships which are involved in a defect in particular, we need to reason on the maximum possible explanation length (number of its parts) which can be generated from a model. For this purpose, we define the "size of model" as the number of features plus the number of cross-tree constraints. Since one single explanation part can contain either one single cross-tree constraint or a relationship from a child feature to its parent feature, an explanation (without redundant parts of course) cannot be longer than the "size of model". Therefore, the "size of model" represents also the maximal possible explanation length. In Figure 5.8, we present the average explanation length for all evaluated models compared to the size of a model in percentage. The figure shows that small size models like Sorting Line or PPU contain up to approximately 10% of faulty relationships, whereas fault localization for large-scale models determines less than 1% of the overall size. We reason on this findings in Section 5.4.2.

Next, we analyze statistics on the shortening of explanations for redundant cross-tree constraints. Table 5.4 presents the number of all explanations per model, the number of all first explanations which are not the shortest ones and a relative average shortening effect in percentage. To summarize the statistics, approximately 1 out of 6 first explanations could be shortened. Regarding the shortening effect, explanations are approximately 25% - 50% shorter compared to the first explanation.

In the following, we inspect implicit constraints in interrelated feature models using the automotive feature model which acts as the complete feature model. Table 5.5 summarizes the occurrence of hidden dependencies at depth 1 and 2 of the automotive feature model. The feature model consists of 6 submodels and comprises 12 implicit constraints at depth 1. At depth 2, 26 submodels and 186 implicit constraints are present. Implicit constraints at depth 1 consist of 15 explanation parts on average, while the average length of explanations at depth 2 consists of 16 parts.

*Real-world feature models

http://publikationsserver.tu-braunschweig.de/get/64218

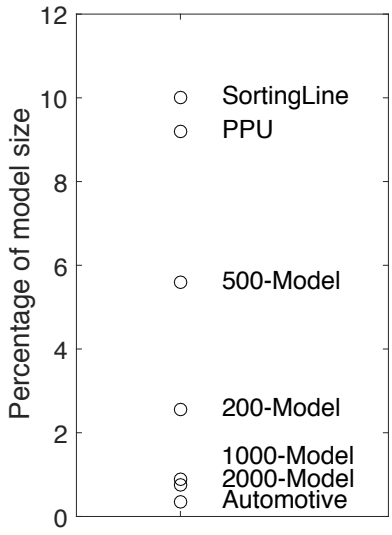

Figure 5.8: Percentage of an average explanation length compared to the size of a model.

Model	# Expl.	# First Expl. ≠ Shortest Expl.	Shorter (%)
Sorting Line*	2	n.a.	n.a.
PPU*	7	1	44.4
200-model	8	2	50
500-model	14	2	25.1
1000-model	44	11	39.7
2000-model	87	21	48.4
Automotive*	563	56	29.8

Table 5.4: Statistics on the average shortening effect of explanations for redundant cross-tree constraints. *Expl.* = Explanation.

In Table 5.6, we measured for every submodel the time for pure slicing and the overall time to explain and visualize implicit constraints. Pure slicing process takes 0.48 seconds on average, while the explanation and visualization of implicit constraints requires 102 seconds on average.
For a single implicit constraint at depth 1 and two implicit constraints at depth 2, we could not explain the implicit constraints. The error was caused due to a conditionally dead feature (i.e., a feature becoming dead due to certain conditions, for instance, other dead features.) leading to the implicit constraint at depth 1. The error propagated to the two implicit constraints at depth 2. Currently, we are not able to explain all cases of defects with BCP which are caused by conditionally dead features.

http://publikationsserver.tu-braunschweig.de/get/64218

Depth	# Submodels	# Implicit Constraints	# Explanation parts
1	6	12	15.1
2	25	186	16.46

Table 5.5: Number of Implicit constraints per depth and their average length.

Depth	Submodel (Parent)	#Features	# I.C.	Pure Slicing(s)	Overall(s)
1	1	105	1	0.48	0.81
1	2	171	-	0.4	0.46
1	3	54	-	0.41	0.52
1	4	112	-	0.40	0.51
1	5	2,065	11	1.54	542.6
1	6	5	-	0.42	0.43
2	7 (1)	8	-	0.48	0.5
2	8 (1)	72	-	0.44	0.47
2	9 (1)	4	-	0.46	0.48
2	10 (1)	3	2	0.47	1.05
2	11 (1)	17	-	0.43	0.45
2	12 (2)	167	-	0.43	0.48
2	13 (2)	3	-	0.44	0.45
2	14 (3)	21	2	0.48	2.08
2	15 (3)	18	-	0.44	0.46
2	16 (3)	3	-	0.44	0.46
2	17 (3)	3	-	0.43	0.44
2	18 (3)	5	-	0.42	0.44
2	19 (3)	3	-	0.4	0.41
2	20 (4)	3	-	0.45	0.46
2	21 (4)	88	-	0.43	0.45
2	22 (4)	16	-	0.42	0.43
2	23 (4)	4	-	0.44	0.45
2	24 (5)	684	123	0.5	170.83
2	25 (5)	16	-	0.43	0.44
2	26 (5)	948	39	0.57	75.7
2	27 (5)	231	20	0.4	21.24
2	28 (5)	185	3	0.44	2.51
2	29 (6)	2	-	0.44	0.45
2	30 (6)	1	-	0.43	0.44
2	31 (6)	1	-	0.45	0.46

Table 5.6: Calculation time of the slicing operation in paricular and the overall time including a detection, explanation and visualization of implicit constraints per model. $I.C.$ = implicit constraints.

http://publikationsserver.tu-braunschweig.de/get/64218

Figure 5.9 represents the calculation time of submodels at depth 2 containing implicit constraints. The calculation time comprises a model analysis to detect defects in single feature models (i.e., dead features, false-optional features and redundant cross-tree constraints) followed by the slicing operation and the detection, explanation and visualization of implicit constraints. We observe, that computation time increases with the number of implicit constraints. Notice that on average the number of implicit constraints increases with the number of features per model. A linear regression shows that calculation time increases approximately by 1.4 seconds per additional implicit constraint. The data from Table 5.6 reveals that pure slicing and the number of features has a negligible influence on the overall time.

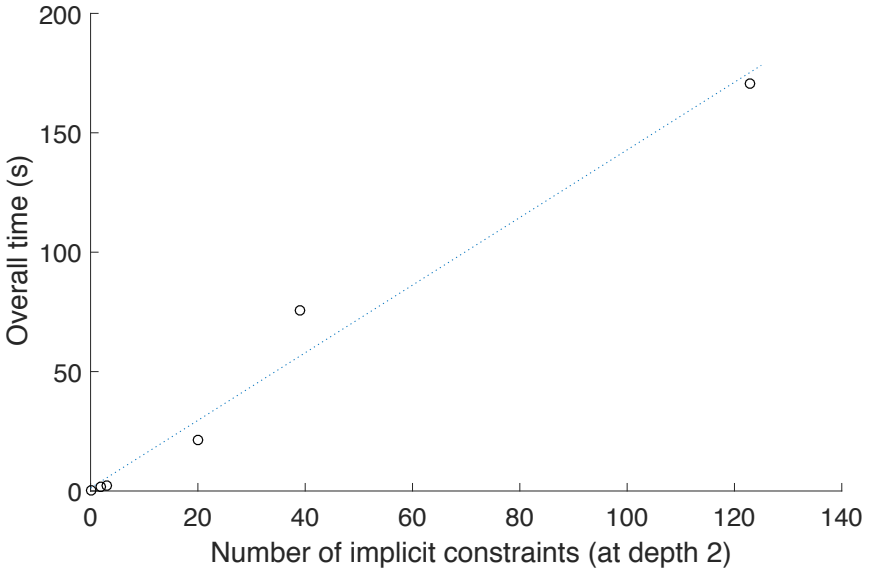

Figure 5.9: Calculation time for detecting, explaining and visualizing implicit constraints per model including a preceding automated analysis.

Table 5.7 depicts the number of adjacent submodels involved in a hidden dependency and the average percentage of *local* features in an explanation. A *local* feature is contained in the considered submodel. Hidden dependencies at depth 1 involve up to three adjacent submodels out of five. Hidden dependencies at depth 2 involve up to four adjacent submodels out of 24. The percentage of local features fluctuates between 24% and 56.1% and comprises on average 37.35%.

Table 5.8 presents a classification of implicit constraints in an implication, exclusion, negation or another kind of logical expressions, e.g., conjunction. Additionally, every expression is further depicted as CNF pattern. 11 out of 12 Implicit constraints at depth 1 represent an implication of the form $\neg A \vee B$. At depth 2, implicit constraints form in 18 out of 186 cases an implication. In 115 cases, an other logical expression occurs in the form of $\neg A \vee \neg B \vee C$.

http://publikationsserver.tu-braunschweig.de/get/64218

Depth	Submodel	# I.C.	# Involved/All Adjacent Submodels	Local Features (%)
1	5	11	3/5	40
2	14 (3)	2	1/24	25
2	24 (5)	123	2/24	29.7
2	26 (5)	39	1/24	24
2	27 (5)	20	1/24	56.1
2	28 (5)	3	4/24	51.3

Table 5.7: Statistics on the maximum number of adjacent submodels involved in a hidden dependency per submodel and the average number of local features in implicit constraints.

Logical Expression	CNF Pattern	Depth 1	Depth 2	Overall (%)
Negation	$\neg A$	1	16	8.6
Implication	$\neg A \vee B$	11	18	14.6
Exclusion	$\neg A \vee \neg B$	-	31	15.7
Other	$A \vee B \vee C$	-	1	0.5
	$\neg A \vee \neg B \vee C$	-	115	58.1
	$\neg A \vee B \vee C$	-	5	2.5

Table 5.8: Classification of implicit constraints into logical expressions and representation as CNF patterns.

5.4.2 Interpretation

In this subsection, we analyze the collected data and describe results and observations. Based on that, we evaluate the data and formulate insights concerning the performance impact of the generation of explanations, their length and findings with respect to implicit constraints.

Calculation time

1. **Generating explanations increases computation time significantly.** Table 5.2 shows that the generation of the first and the shortest explanation is a significant factor for the performance results of an automated analysis in FeatureIDE. The computation time is roughly doubled by the generation of the first explanation, while finding the shortest explanation takes additionally about 30% time. In detail, computing a colored explanation exceeds the calculation time which only included the detection of defects by factor 2.6 on average. Nevertheless, this is a reasonable expense to retrieve information enabling a precise and fast default correction compared to the notoriously difficult debugging of large-scale feature models. Hence, we conclude that the performance impact of BCP is acceptable answering **RQ3**. Still, we can reduce the tradeoff between the performance impact of explanation and its information yield by choosing to generate

http://publikationsserver.tu-braunschweig.de/get/64218

only the first explanation, which in most cases already returns the shortest one. Table 5.2 additionally reveals only a slightly increased computation time for tracing by 6.75 % in average, while the coloring process is almost instantly finished. Hence, the tracing to feature model elements and coloring process of explanation parts is of little importance for the performance impact of the generation of explanations.

2. **Generating explanations is scalable.**
From the results depicted in Table 5.2, we also observe that the relative performance impact stays roughly constant for models of different sizes. Consequently, we conclude a scalability of the BCP algorithm.

Explanation length

1. **Explanation length increases slightly compared to feature model size**.
Given the rather small product line of the PPU, we observe that each explanation has at most nine parts (cf. Figure 5.4). Although the 200-Model is about four times larger (cf. Figure 5.5), we only see a slight increase in the explanation length. This trend continues for the larger models as well (cf. Figure 5.6). Even for the automotive feature model with 2,513 features, 75% of the explanations contain merely up to 25 parts (cf. Figure 5.7) To answer **RQ4**, the average length of an explanation comprises roughly 11 parts which we consider as still comprehensible for developers.

2. **Isolation of defects gets better for larger model sizes.**
Related to the finding above, we observe that an increasing model size results in a better isolation of a defect by analyzing Figure 5.8. To answer **RQ5**, we determine that an explanation is able to greatly isolate a defect up to mostly 0.34% of the automotive model size.

3. **Exceptionally long explanations can occur in large feature models.**
Compared to small and middle sized models, large models can cause unusually long explanations. For the automotive model with 2,513 features and 2,833 constraints, the longest explanation consists of 95 parts which is approximately 2% of the model size (cf. Figure 5.7). Outliers in Figure 5.6 and Figure 5.7 represent such explanations. We consider explanations of this length as not comprehensible anymore and suggest respective future work (cf. Chapter 8).

4. **In most cases, BCP finds the shortest explanation immediately.**
Table 5.4 shows that in most cases, the first found explanation is already the shortest one answering **RQ6**. To further improve the performance impact of the generation of explanations, we may decide to only generate first explanations.

5. **Shortest explanations are significantly smaller.**
Table 5.4 shows that shortest explanations are 25% - 50% smaller than first explanations to answer **RQ7**. Although BCP already finds the shortest explanation

http://publikationsserver.tu-braunschweig.de/get/64218

immediately, a shortest explanation is significantly smaller which justifies the effort when dealing with middle or large-scale feature models.

Implicit Constraints

1. **The number of implicit constraints increases with the number of adjacent submodels.**
   In Table 5.5, we observe the occurrence of implicit constraints in a real-world automative feature model. At depth 1 of a feature model, the amount of implicit constraints is 11, whereas at depth 2 it significantly increases to 186.

2. **Computation time increases with the number of implicit constraints.**
   Based on the performance measurements in Table 5.6, we determine 0.48 seconds as the average pure slicing time of the automotive feature model. Regarding the computation time of submodels at depth 2 in Figure 5.9, computation time increases with the number of implicit constraints. In particular, calculation time increases approximately by 1.4 seconds per additional implicit constraint answering **RQ8**. We believe that this is a reasonable expense for explaining hidden dependencies, especially in a large-scale model.

3. **Explanation length stays constant at different depths.**
   As presented in In Table 5.5, the explanation length increases from 15 to 16 parts. Therefore, we assume that explanation length does not depend on source depths of submodels.

4. **Less than 40% of all features in an explanation are local features.**
   Table 5.7 depicts that an explanation comprises from 24% to 56.1% local features answering **RQ9**. On average, explanations comprise 37.35% local features leading us to the conclusion that a hidden dependency is mostly caused by relations between features from other submodels. This assumption is further supported by the fact that up to 4 adjacent submodels of the evaluated submodel can be involved in a hidden dependency as presented in Table 5.7.

5. **Implicit constraints occur most often as $\neg A \vee \neg B \vee C$.**
   To answer **RQ10**, Table 5.8 shows that an implicit constraint occurs most often as $\neg A \vee \neg B \vee C$ ($\equiv A \wedge B \Rightarrow C$). The second leading CNF pattern comprises an exclusion between two features, while an implication of the form $A \Rightarrow B$ represents the third leading pattern.

5.5 Threats to Validity

In this section, we reason about the validity of the presented results. We categorize the validity of our results according to Wohlin et al. into *conclusion, internal, construct* and *external validity* and identify respective threats [68].

http://publikationsserver.tu-braunschweig.de/get/64218

Conclusion validity statistically ensures the rationality of our conclusions about relationships in our data. While conclusion validity only concentrates on whether there exists a relationship, *internal validity* evaluates a causal relationship between the program and the results and that no unknown or not measured factor was involved in the outcome. *Construct validity* assesses whether we designed the experiment correctly to retrieve the desired data. *External validity* is related to generalizing the results, e.g., conclusions from our results hold for other feature models.

Conclusion validity

Due to the lack of large real-world feature models which are still scalable for analysis operations, results for implicit constraints are based on one single large real-world feature model. To validate the derived insights concerning implicit constraints, e.g., number of occurrences, logic structure and the explanation length of implicit constraints, we need to evaluate further large real-world feature models containing implicit constraints.

Internal validity

For the internal validity, we identified two threats:

1. The resulting explanation can depend on the sequence of clauses in a CNF (cf. Section 3.2.2). If we would change the sequence and restart the explanation algorithm, we may receive another content as explanation. To tackle this threat, we developed a heuristic which is used to search for multiple explanations per defect to prefer the shortest one. Hence, we are able to find other explanations. We also put thought into considering every possible clause order, i.e., a permutation of all CNF clauses. However, the performance cost is not viable.

2. The usage of artificially generated feature models might have a distorting effect on hypotheses about the length of explanations. Nonetheless, we chose some generated models due to the availability and reasonable number of different defects. This allowed us to collect data for differently sized models.

Construct validity

For the construct validity, we identified a threat caused by a *mono-operation bias*[68]. In other words, our results concerning implicit constraints may under-represent the respective findings due to a single evaluated feature model. This is due to the lack of scalable large real-world feature models, as mentioned above. Nevertheless, we carefully analyzed the model at different depths which led to a reasonable variety and amount of data.

http://publikationsserver.tu-braunschweig.de/get/64218

External validity

For the external validity, we identify following threats concerning the generalization of our results.

1. The usage of generated feature models for evaluation limits the generalization of our results to other real-world feature models. We argue that we are still able to get a proper overview of the generated explanation length by covering a great number of different defects and additionally evaluating multiple real-world feature models.

2. To ensure an external validity even more, we need to evaluate more real-world feature models from different domains.

5.6 Summary

In this chapter, we performed a qualitative and quantitative analysis of the resulting explanations. We used feature models differing in kind and size including real-world feature models. Three models contained 1,000 features and above showing the scalability of the generic algorithm and, thus, realizing **RG1.4**.

For the qualitative analysis, we provided multiple test feature models per defect along with its explanation in Chapter A and reasoned on some exemplary explanations. Based on the test models, we analyzed and confirmed the correctness of the explanation algorithm. We concluded that an explanation reveals the faulty connections which lead to a defect. Additionally, we inspected the highlighting of specific parts. We argued that changing or removing only those parts already fixes the defect. For the quantitative analysis, we further analyzed that a first generated explanation is most often already the shortest one, while an average explanation length varies between 3 and 18 parts per model. Nevertheless, shorter explanations are up to 50% smaller which is especially helpful if dealing with large feature models. Hence, we concluded that improvements of the basic explanation algorithm help to increase the quality of explanations satisfying **RG1.3**. We reasoned that for implicit constraints, the average time to derive and explain implicit constraints is still acceptable and grows with the number of implicit constraints. Additionally, we detected that up to 5 out of 25 submodels are involved in a hidden dependency and that $\neg A \vee \neg B \vee C$ is the most common pattern of an implicit constraint. It must be noted that the generic explanation approach is subject to continuous monitoring in FeatureIDE and evaluation is still going on.

http://publikationsserver.tu-braunschweig.de/get/64218

6. Related Work

In this chapter, we present related work to clearly differentiate the developed explanation approach. First, we describe work in the area of the automated analysis of feature models involving defect detection. Next, we concentrate on former and other explanation approaches and compare them to our work. Finally, we describe the state-of-the-art for detecting hidden dependencies including further slicing approaches.

Automated analysis and defect detection

The automated analysis of feature models is an active area of research which has led to many analysis operations, algorithms and tools to support an automated analysis process of a feature model [8, 10]. Meinicke et al. provide an overview on analysis tools for SPLs [47]. FeatureIDE, pure::variants, SPLOT and FaMa are some of the tools to support feature modeling[1]. According to Benavides et al. [10], automated analysis comprises up to 30 different analysis operations including defect detection, checking the validity of products, explanations, refactoring and optimization. To produce error-free feature models, many approaches including the previously mentioned tools provide a feature model error analysis. Von der Massen and Lichter give a definition and an overview of various feature model errors while Benavides et al. present an overview of automated analysis operations to detect the described defects in feature models [10, 67]. Hemakular introduces an approach to statically detect contradictions in feature models [30]. The approach comprises model checking in combination with BCP to propagate consequences of a feature selection and report contradictions. Model checking is a powerful analysis tool which is able to traverse states of a state machine and checks whether particular states are reachable. Here, model checking finds user selections which lead to an *error state*. An error state results from a contradiction which is detected by BCP during constraint propagation. Nevertheless, model checking performs poorly for large feature models and no explanation support is given. Further significant research for

[1] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/fosd-tools/

http://publikationsserver.tu-braunschweig.de/get/64218

error detection was performed by Trinidad et al. [66]. The authors express a feature model in terms of a CSP and focus on the detection of dead features, false-optional features and void feature models (cf. Section 2.3.2). Trinidad et al. make use of the *Theory of Diagnosis* which includes a mapping of the feature model to a corresponding *diagnosis problem* based on concepts proposed by Reiter [54].
In this thesis, we propose an explanation algorithm which requires an arbitrary defect detection to generate a respective explanation. Additionally, it can be used independently of any solver.

Defect explanation

Concerning the explanation process, Batory did significant research on reusing an LTMS for explaining a user's configuration of an SPL variant (cf. Section 2.3.2) [7]. Batory introduced a tool called *guidsl* that provides an open-source implementation of the LTMS. Our explanation algorithm is most closely related to his findings. In contrast, *guidsl* can explain only dead features while no explanation support is provided for the remaining defects. We improved Batory's approach by explaining all kind of defects in a more user-friendly manner, searching for shorter explanations and highlighting relevant parts. Additionally, we applied LTMS to already support a developer during the modeling phase.

Using the *Theory of Diagnosis* by Reiter [54], Trinidad retrieves a minimal set of faulty constraints to explain a defect (cf. Section 2.3.2) [66]. The implementation is available in FAMA [12, 65]. We identify similar yet different elements compared to our approach: We provide support to fix a defect at the early stage of feature modeling by highlighting significant explanation parts. FAMA also provides support for error-repairing, but at the later stage of product configuration by presenting a set of necessary feature selections and deselections to fix an invalid product configuration. Furthermore, FAMA is able to explain all kinds of defects except for redundant constraints. Further differences comprise FAMA using only small feature models for evaluation and expressing explanations in a greatly abbreviated way (cf. Section 2.3.2) which may be at the expense of expressing an explanation in a user-friendly manner.

A well-known diagnosis algorithm is *QuickXplain*, which made a major contribution concerning the efficiency of explaining a CSP [34]. *QuickXplain* uses of a divide-and-conquer strategy to determine a minimal faulty set of constraints in a CSP. Lesta et al. adapt *QuickXplain* to explain defects in *attributed* feature models, i.e., a feature containing integer-valued attributes [40]. For reasons of efficiency, the authors refrain from searching for a minimal explanation. In contrast, we focus on providing support on defect explanation for *pure* feature models, i.e., features without attributes. Further on, the approach explains all defects except for redundant constraints and relies on a constraint solver whereas our approach is independent of any solver. Last but not least, the source code of the presented approach is not open-source.

Felfernig et al. demonstrates another explanation approach based on the *Theory of Diagnosis* using the *FastDiag* algorithm [25]. Similarly to *QuickXplain*, *FastDiag* is a

http://publikationsserver.tu-braunschweig.de/get/64218

divide-and-conquer diagnosis algorithm. It receives a set of diagnosable constraints of the feature model and divides the constraints iteratively into subsets, which results in determining a smallest possible subset of inconsistent constraints. A resulting explanation consists of the faulty constraint set, which has to be adapted or deleted in order to resolve the defect [26]. Hence, the algorithm guarantees to find a minimal possible explanation in contrast to our approach. Apart from previously described approaches, *FastDiag* is not restricted to a CSP. Similarly to our work, *FastDiag* explains all types of defects and is independent from any solver. In constrast, evaluation is limited to feature models offered by the S.P.L.O.T. repository containing 72 and 172 features. Additional cross-tree constraints have been randomly inserted to induce inconsistencies. No information is provided on the number or kind of defects in the feature model. An evaluation of the performance impact for large-scale feature models is missing to show the scalability of *FastDiag*.

The ontological rule-based approach by Ricón et al. generates explanations for defects in feature models in natural language (cf. Section 2.3.2) [55]. The approach provides explanations using a similar syntax as we do and rivals our explanation length. Nonetheless, only dead and false-optional features can be explained.

Another approach which does not concentrate on explaining defects but active configurations is proposed by Kramer et al. [38]. The authors introduce an explanation approach for active configurations within Dynamic Software Product Lines (DSPL). DSPLs are reconfigurable at runtime enabling a dynamic variability for an SPL. To generate explanations, every feature holds its own description and every relationship is mapped to a respective expression, e.g., Optional - "*which is optional*", Or - "*which can be*". As shown in Figure 6.1, explanation fragments belong to features and relationships. Their concatenation forms a complete explanation why configuration *stream* is active, i.e., *"The ContentStore stores historical content purchases in all configurations, of which can be retrieved in all configurations, either downloaded or streamed, in this case, streamed"*.

Osman et al. propose an approach for multiple analysis operations on feature models (feature model validation, detection of dead features and inconsistencies and explanations) [23]. In contrast, explanations are provided during the configuration process of an SPL if the user selects invalid feature combinations whereas we assist on defect explanation during feature modeling. Similarly, Osman et al. provide explanations which guide users in case of an invalid feature selection, i.e., explanations comprise suggestions to select or deselect certain features. We aim to guide a user by highlighting relevant explanation parts.

Finally, related work exists which focuses on finding guaranteed minimal explanations. This is currently not possible with our approach and part of future work. Liffiton et al. focus on finding all minimal unsatisfiable subset of clauses (i.e., *minimal core*) given an unsatisfiable boolean formula [42]. The authors refer to a minimal core as a *minimal unsatisfiable subset* (MUS). In relation to defects in feature models, an MUS would form the basis of a minimal explanation consisting of pure clauses. In contrast, a

http://publikationsserver.tu-braunschweig.de/get/64218

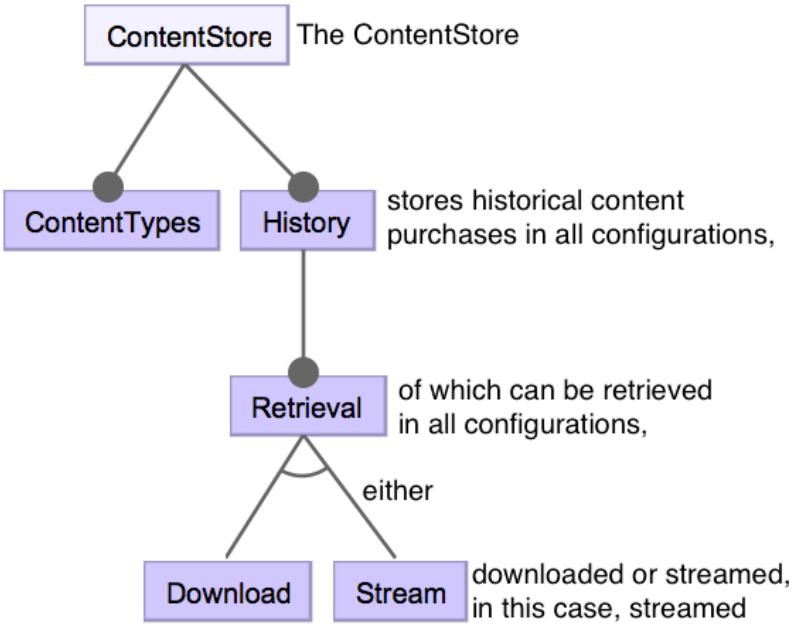

Figure 6.1: Explaining fragments for the active configuration *stream* [38].

minimal correction subset (MCS) is a minimal set of clauses whose removal leads to the propositional formula becoming satisfiable. Liffiton et al. propose *CAMUS* (**C**ompute **A**ll **M**inimal **U**nsatisfiable **S**ubsets), a two-phase approach to determine all minimal cores. First, *CAMUS* finds *all* MCSs. Based on that, it derives MUSs. According to the authors, algorithms of *CAMUS* are easily to adapt in order to work with different constraint solvers or to reach different goals. An evaluation of *CAMUS* shows its scalability and reveals that it performs significantly better than existing algorithms for each of the two phases. Nevertheless, the first phase is very costly in terms of resources and therefore not advisable when only one minimal explanation is required.

An up-to-date approach to determine one minimal explanation in terms of clauses is presented by Guthmann et al. [29]. The authors present the tool *HSMTMUC* which is implemented on top of a *satisfiability modulo theories* (SMT) solver for finding one minimal unsatisfiable subset of clauses in a propositional formula, i.e., a *minimal unsatisfiable core* (MUC) [51]. As the approach described above, an MUC would represent a raw explanation consisting of pure clauses (if applied to explain a defect within a feature model). SMT solvers determine the satisfiability for first-order formulas comprising some logical theory, e.g., integer arithmetic, and are also capable of finding an unsatisfiable core [6]. Guthmann et al. evaluate different SMT solvers, i.e., *Z3* [50] and *MATHSAT* [14], to first extract an unsatisfiable core which is, however, usually non-minimal and then proceed to minimize it by invoking *HSMTMUC*. The authors conclude that best runtime and smallest average core size is retrievable by first running *MATHSAT* and then invoking *HSMTMUC*.

http://publikationsserver.tu-braunschweig.de/get/64218

Detection of hidden dependencies

While the detection of defects is well researched, only few approaches exist in literature with respect to dependencies between interrelated feature models. One of those approaches is introduced by Mendonça et al. [48]. The authors argue that the configuration of products is complicated due to different involved parties. They propose an approach called *collaborative product configuration* to coordinate the configuration of products. To support the validation of configurations, Mendonça et al. provide efficient dependency analysis algorithms which detect *interdependent features*, i.e., the selection of one feature automatically selects an other feature. Specifically, dedicated algorithms are performed on graphs representing feature models to support the dependency analysis. Contrary to our approach, hidden dependencies are detected on configuration level, whereas we perform a dependency detection on modeling level. Additionally, no explanations of the detected dependencies are given.

Another approach to detect hidden dependencies between features is introduced by Ghanam et al. [28]. The authors present a linking of feature models to code artifacts using *executable acceptance tests* (ETA). EATs are executable specifications of requirements which automatically test the behavior of a system for a certain input. Ghanam et al. extend features containing implementation artifacts with multiple EATs. Consequently, the selection of a feature implies the execution of all EATs. Additionally, EATs inherit dependencies imposed on the linked features, i.e., EATs of alternative features are alternative as well. The authors argue that EATs help to reveal hidden feature interactions comprising *excludes* and *requires* dependencies. For instance, EATs fail when two independent features are simultaneously selected. This may either reveal an issue of the implementation or an *excludes* dependency between two features. In contrast to our work, no explanation is provided. Consequently, the developer has to inspect the respective implementation part or feature model to detect why an EAT failed. Although the presented approach is able to detect hidden dependencies, only dependencies representing simple constraints can be detected whereas we are able to derive complex constraints as well (cf. Section 2.1.1).

Lettner et al. introduce an approach to support the modeling of large-scale feature models with multiple dedicated, interrelated feature models for different purposes, scopes and granularities (cf. Section 2.1.2) [41]. The authors face a lack of explicit knowledge about feature dependencies and the challenge to detect such dependencies between features of different modeling spaces, i.e., solution, problem and configuration space. They refer to Berger et al. who found that modelers from industry even try to avoid cross-tree constraints and only use parent-child relationships [13]. Contrary, we do not operate in different spaces, but we claim that revealing and understanding hidden dependencies is also a current challenge in interrelated feature models.

Jonata et al. concentrate on describing an SPL from different perspectives, i.e., a feature model and an architectural model [33]. The architectural model is a component model which comprises implementation details of the SPL. During the configuration process of a feature model, inconsistencies between both models can arise, e.g., a feature model

http://publikationsserver.tu-braunschweig.de/get/64218

enables a configuration which is not possible due to restrictions of the architectural model. To tackle this problem, the authors propose to integrate both models and to detect hidden dependencies in the feature model. Therefore, both models are expressed in propositional logic and combined. By applying *existential quantification*, which is a possible approach for feature-model slicing, Jonata et al. compute an induced feature model containing implicit constraints. The original feature model is then enhanced with such implicit constraints to support the application engineer during configuration. Similar to our work, feature model slicing is performed on a CNF and implicit constraints are detected. Contrary, Jonata et al. use *existential quantification* to slice a feature model, whereas we take advantage of *logical resolution*, which is based on *existential quantification* but improves it by keeping the resulting formula in CNF.

Next, we address related work concerning feature model slicing, i.e., the removal of features from a model while not changing existing dependencies. Thüm et al. first introduces the idea of feature model slicing when reasoning about edits between two feature models, particularly the removal of abstract features in a feature model [62]. An edit to a feature model results in a new feature model and can either be classified into a *specialization* (i.e., products are removed from an SPL), a *generalization* (i.e., no new products are added to an SPL) or a *refactoring* (i.e., the number of products of the SPL stays constant). To determine an edit between two feature models containing abstract features, Thüm et al. remove every occurrence of an abstract feature until a resulting propositional formula representing a feature model only consists of concrete features. Support for the classification of edits is integrated in FeatureIDE.

On this basis, Acher et al. propose a feature model slicing technique but focuses on the decomposition of large-scale feature models, i.e., splitting large models into a set of interrelated submodels [2, 3]. The slicing technique is implemented into **FAMILIAR**, a domain specific language of manipulating feature models [1].

Based on the work by Thüm et al. [62], Krieter et al. implemented a more efficient slicing approach in FeatureIDE which uses *logical resolution* (cf. Section 2.3.1) [39]. We make use of this approach to derive a submodel from a complete one. In a large case study, Schröter et al. take advantage of the slicing algorithm to derive *feature-model interfaces* which contain a subset of arbitrary features from the complete feature model to hide information for modelers and stakeholders [59]. Furthermore, the authors enable a composition of feature-model interfaces. To reduce the amount of re-computations for a composed model and ease its maintainability, Schröter et al. reuse analyses results of single interfaces.

Another concept to hide information in feature models are *feature model views*, introduced by Clark et al. [15]. A view hides undesired features of a complete feature model to reduce the complexity of a large-scale feature model. Contrary to feature model slicing, it is often used to support the configuration process a feature model, e.g. multiple views cover all configuration questions and propagate feature selections to other views to ensure their consistency [44, 58].

http://publikationsserver.tu-braunschweig.de/get/64218

7. Conclusion

Variability has become subject to many software systems and is at the centre of the development paradigm of software product lines. Applying the idea of product lines in the software domain has led to an efficient production of variant-rich software systems. Nevertheless, modeling a software product line can lead to different modeling errors. Not only the detection, but also the user-friendly explanation of such defects potentially allows for an improved and faster error handling.

In this thesis, we proposed a generic explanation approach based on predicate logic, which is most closely related with previous work by Batory [7]. We applied it to explain different kinds of defects in single and interrelated feature models in a user-friendly and structural manner (cf. Section 2.2). The generic explanation approach is based on a *logic truth maintenance system* (LTMS) which is a classic AI program (cf. Section 3.2.1). It derives assumptions about truth values of variables in a propositional formula and remembers the reasons for its belief. A most suitable algorithm for implementing an LTMS is considered to be the *boolean constraint propagation* (BCP) (cf. Section 3.2.2). In this thesis, we reused BCP to explain defects in feature models by only varying two input parameters: the faulty feature model represented by a CNF formula and initial premises, i.e., truth values for a subset of literals of the CNF. The essence of BCP is to find unit-open clauses in a CNF (i.e., clauses whose literals are all bound to *false* except for one unbound literal) and infer the truth value for the unbound literal to ensure the satisfiability of the clause. This process iteratively continues until BCP encounters a contradiction. A contradiction arises from a violated clause (i.e., a clause whose literals are all bound to *false*). A resulting explanation consists of the violated clause and all maintained reasons (i.e., former unit-open clauses and assumptions). We refer to a single reason in an explanation as an *explanation part* being expressed in user-friendly manner, such as *B is mandatory child of A*. Additionally, we identified and implemented two main improvements of the basic BCP algorithm. A generated explanation by BCP depends on the order of the CNF clauses. Consequently, the algorithm does not always find an explanation with a minimal length. To find a shorter explanation

http://publikationsserver.tu-braunschweig.de/get/64218

compared to the first one, we proposed a heuristic which takes advantage of the stack maintaining unit-open clauses: After a contradiction occurred and BCP generated the first explanation, we restarted the explanation process for the remaining unit-open clauses on stack and preferred the shortest explanation found (cf. Section 3.4.1). The second improvement comprised an emphasis of relevant explanation parts which most likely caused the defect. Error-prone explanation parts would occur in all or many explanations for a defect. We emphasized such parts depending on their occurrence using a coloration from red (occurred in all explanations) to black (occurred in one out of several explanations). Applying the improved BCP algorithm, we generated explanations for void feature models, dead and false-optional features as well as for redundant cross-tree constraints. Furthermore, we provided explanations for implicit constraints representing hidden dependencies in interrelated feature models. To derive implicit constraints, an existing slicing algorithm was applied on a complete feature model. The complete feature model was either created by combining interrelated feature models into one large feature model or it was a monolithic feature model comprising interrelated submodels which, for instance, represented various disciplines of a product. The slicing operation returned a submodel enhanced with implicit constraints. To explain implicit constraints, we treated such constraints identically to redundant ones, since implicit constraints were derived from relationships of the complete feature model

We integrated the explanation approach into the existing structure and workflow of FeatureIDE (cf. Chapter 4). The current implementation is available in the major FeatureIDE 3.1.0 release. To express explanations in a user-friendly manner, we added structural information to every literal during the creation of a CNF. For reasons of efficiency, the structural information is encoded by an additional numeric attribute of a literal object. Structural information of a literal comprises whether it belongs to the tree topology, acts as parent or child feature in a CNF clause or if it is part of a cross-tree constraint.

Finally, we performed a qualitative and quantitative evaluation of the resulting explanations (Chapter 5). The qualitative evaluation included a manual analysis of the correctness of the BCP algorithm and whether explanations contained the necessary information to understand the defect and, consequently, fix it. For the qualitative analysis, we used multiple test feature models per defect, overall 33 models. We confirmed the correctness of BCP and that explanations revealed the faulty relationships between features which led to a defect.
For the quantitative evaluation, we used seven feature models of different sizes including three real-world models. The retrieved measurements comprised the performance impact for the generation of explanations, their length and statistics on the shortening of explanations. For implicit constraints in particular, we additionally inspected the number of implicit constraints in a large real-world automotive feature model together with their logical structure, the computation time of explaining implicit constraints and determined the percentage of local features in an explanation, i.e., features which are contained in the observed submodel. To summarize the main insights, computation time for model analysis significantly increased when generating explanations for

http://publikationsserver.tu-braunschweig.de/get/64218

defects. Finding a first explanation approximately doubled the computation time while searching for a shortest explanation took additional 30% time over finding the first one. Nevertheless, we concluded that the performance impact is acceptable and that advantages of explaining defects in large feature models compared to the difficulties of debugging such models clearly overweight the increased computation time. Furthermore, we showed the scalability of the explanation approach by evaluating different sizes of feature models containing large-scale feature models with 2000 features and more. For all models, the performance impact stayed roughly constant. The length of explanations varied between 3 and 18 parts on average per model. We observed only a slight increase in the number of explanation parts for significantly larger model sizes. Based on that, we concluded that the percentual benefit of the algorithm improves for larger feature models and, hence, the isolation of defects gets better for larger model sizes. Further findings concerning explanation length comprised that in most cases, BCP found the shortest explanation immediately. Nevertheless, if BCP generated a shorter explanation than the first one in a later processing step, it was approximately 25% - 50% smaller. We considered the shortening effect as an essential benefit if dealing with long explanations. Finally, we observed implicit constraints of submodels at different depths of a real-world large-scale automotive model with 2,513 features. Main findings comprised that computation time increased with the number of implicit constraints per submodel. In particular, an implicit constraint took approximately 1.4 seconds. We believed that this is a reasonable expense for explaining hidden dependencies, especially in a large-scale model. Furthermore, we analyzed explanations of implicit constraints and discovered that up to four (out of six) submodels at depth 1 and up to five (out of 25) submodels at depth 2 were involved in a hidden dependency. On average, less than 40% of the features in an explanation originated from the observed submodel. We assumed, that hidden dependencies were mostly caused by relations between features from other submodels.

To end this chapter, we refer to the research goals **RG1** and **RG2**. To realize **RG1**, we developed a generic algorithm which produces a correct output representing a comprehensible explanation for any defect within a feature model. A tool tip contains the user-friendly explanation for a defect and is displayed during the modeling phase. After achieving **RG1** which comprised the development of a generic explanation algorithm, we concentrated on realizing **RG2** which involved an explanation of implicit constraints in interrelated feature models by reusing the generic explanation approach. Therefore, we took advantage of an efficient slicing algorithm in FeatureIDE to derive a submodel from a complete model. The resulting submodel was enhanced with implicit constraints which we needed to detect first. Applying the generic explanation algorithm, we were able to explain implicit constraints in interrelated feature models with a reasonable performance impact. To view implicit constraints in a submodel, a user is able to select a feature from the complete feature model which acts as the root feature of a submodel. Sequently, a new dialog opens containing the submodel with highlighted implicit constraints and respective explanations displayed in a tool tip.

http://publikationsserver.tu-braunschweig.de/get/64218

http://publikationsserver.tu-braunschweig.de/get/64218

8. Future Work

The elaboration of a generic algorithm provides many interesting approaches for future work. Improvements comprise, for instance, the usability and understandability of explanations. Additionally, we propose future work with respect to implicit constraints.

- *Guaranteeing the shortest explanation.*
  Develop a new strategy with BCP to guarantee finding the overall shortest explanation. Nevertheless, computing an optimal explanation makes it difficult to keep the explanation approach scalable.

- *Graphically support the understandability of explanations in the feature model.*
  Generating explanations for feature models might result in quite long and unreadable explanations. Highlighting all parts of the explanation in the feature-tree and respective cross-tree constraints might significantly increase and speed up the user's understanding of explanations. Additionally, if an explanation is generated for a large feature model, irrelevant parts of the model with respect to the explanation could be concealed to enable a compact view on the model.

- *Using BCP to generate explanations on different levels of abstraction.*
  Currently, we make use of BCP to explain defects which occur during the modeling phase of a feature model. Nevertheless, BCP can also be applied to generate explanations on further levels of abstraction of an SPL, e.g., support developers during the configuration process of a variant. For instance, *guidsl* applies BCP on configuration level to provide feedback why certain features cannot be selected [7]. Another application might comprise the explanation of *dead code*, e.g., code blocks surrounded by #ifdef directives which are never included in any feature selection [61]. To detect dead code, a combined analysis consisting of the feature model and implementation of the SPL is performed. Particularly, the conjunction of a *presence condition* (i.e., a propositional formula comprising feature

http://publikationsserver.tu-braunschweig.de/get/64218

selections in which a code block is present) and the formula representing a feature model is not satisfiable [4]. Since BCP is based on predicate logic, we wonder if it might also be used to detect and explain dead code.

- *Including the user's wish how to use explanations.*
  For large feature models, explanations might become unreadable and downgrade the performance of the feature model analysis. Additionally, users might not be interested in explanations. If a user decides to use explanations, two modes could be distinguished. Either to compute the first explanation which approximately doubles the computation time of the feature model analysis or to search for a shortest explanation which approximately takes additional 30% time.

- *Improving the counting of explanations for redundant constraints.*
  For the coloration process of explanation parts, we count all generated explanations. Explanations for redundant constraints result from combining individual explanations for different assignments (premises) which lead to non-satisfiable redundant constraints. Thus, explanations can occur which contain the same clauses appearing in a different order. An improvement comprises the detection of such explanations and ignoring those in the counting of all explanations for one redundant constraint.

- *Explaining conditionally dead features.*
  Conditionally dead features become dead due to certain conditions, e.g., they always appear together with other dead features in a configuration. BCP is currently able to only explain unconditionally dead features by setting the truth value of a dead feature to *true*. Applying the same strategy to conditionally dead features can result in a missing contradiction. Consequently, no explanation is possible. A strategy is needed which determines the preconditioned dead features and which then can be considered during the BCP process.

- *Evaluating the performance impact of BCP to detect redundancy.*
  FeatureIDE detects redundancy using a SAT solver. Amongst the detection of other defects such as dead or false-optional features, the detection of redundancy is of high computational effort. BCP generates explanations for a redundant constraints based on a contradiction, which also acts as an indicator for the existence of such constraints. A substitution of the redundancy detection with BCP might accelerate the analysis of feature models. This requires a performance comparison between the two detection methods.

- *Comparing the performance impact of BCP to other explanation algorithms.*
  To asses the performance impact of BCP compared to rivaling explanation algorithms, e.g., *QuickXPlain* or *FastDiag*, a performance comparison is needed.

- *Providing tool support for repairing defects.*
  The coloring of relevant explanation parts can be further exploited, e.g., automatically remove the emphasized parts to fix the defect.

http://publikationsserver.tu-braunschweig.de/get/64218

- *Enabling edit operations in a submodel.*
  Currently, the slicing operation returns a submodel which is non-editable. It represents a consistent partial view on the complete feature model. An improvement for handling implicit constraints comprises edit operations of the submodel. Changes in the submodel need to be reflected back to the complete model to keep the two models consistent to each other and support maintenance.

- *Enabling test automation.*
  Evaluation concerning the correctness and performance impact of BCP is currently performed manually. To efficiently (re)test feature models, test automation is needed. Testing the correctness of an explanation is hard, because future improvements might lead, for instance, to other or shorter explanations. A first step towards this goal is testing whether the algorithm generates explanations for different input data at all. Another way to verify the correctness of the BCP algorithm is a formal analysis which proves that the algorithm works for any input data covering a defect.

http://publikationsserver.tu-braunschweig.de/get/64218

http://publikationsserver.tu-braunschweig.de/get/64218

A. Appendix

In this chapter, we demonstrate test feature models containing the previously described defects and respective explanations. For every test model, we provide a description of the defect cause. The explanation algorithm receives the presented feature model as input and returns the explanation contained in a yellow tool tip as output. All test models are contained on a supplied CD-Rom and available online[1].

Test models for void feature models

In Table A.1, we provide four cases of void feature models and a description of the defect cause (path on CD-Rom: */TestFeatureModels/VoidFeatureModels/*).

Reference	Description
Figure A.1	The feature model is void, because core feature B implies feature E whose parent feature C is negated (Test_Void1.xml).
Figure A.2	The feature model is void, because core feature B implies feature E which is alternative to feature D. However, feature E implies feature D which makes the model void (Test_Void2.xml).
Figure A.4	The feature model is void, because core feature B implies the mandatory feature D. The mandatory feature F of the same parent feature as D excludes the core feature (Test_Void3.xml).
Figure A.3	The feature model is void, because it consists of two core features B and C which are mutually exclusive to each other (Test_Void4.xml).

Table A.1: Overview of test feature models for a void feature model.

[1] https://www.isf.cs.tu-bs.de/data/TestFeatureModels.zip

http://publikationsserver.tu-braunschweig.de/get/64218

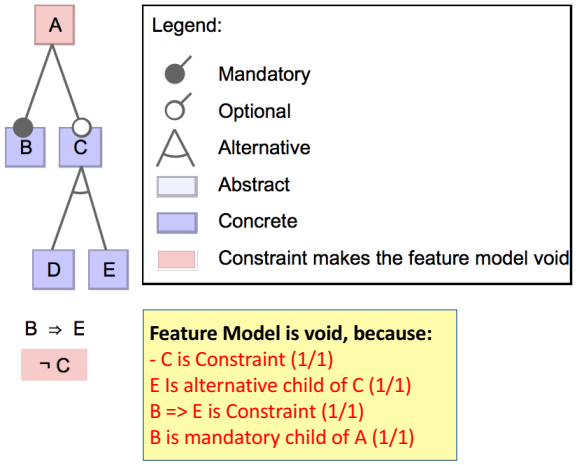

Figure A.1: Void feature model: test case 1

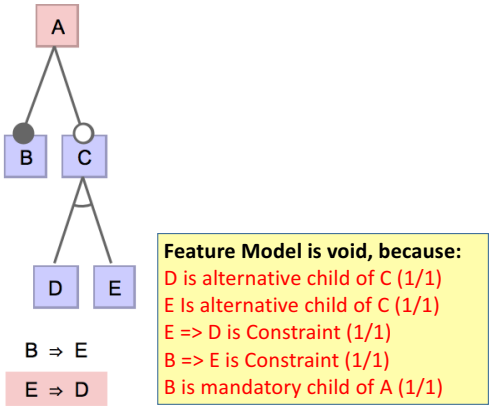

Figure A.2: Void feature model: test case 2

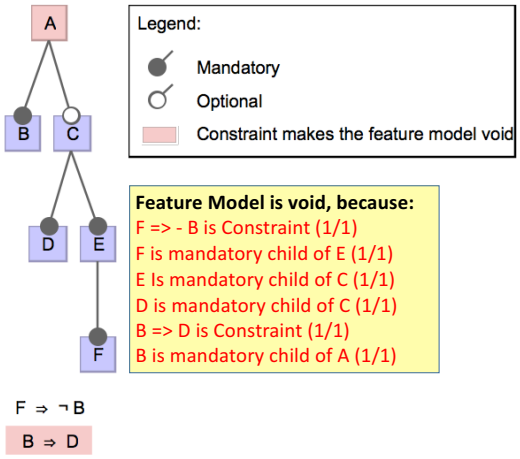

Figure A.3: Void feature model: test case 3

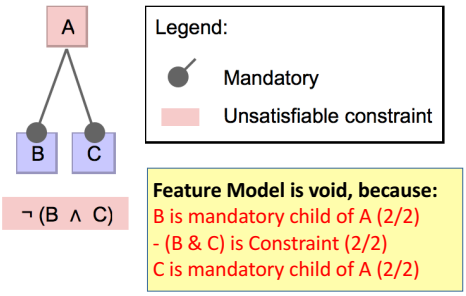

Figure A.4: Void feature model: test case 4

Test models for dead features

In Table A.2, we provide 7 feature models with 10 cases of dead feature and a description of the defect cause (path on CD-Rom: */TestFeatureModels/DeadFeature/*).

http://publikationsserver.tu-braunschweig.de/get/64218

Reference	Description
Figure A.5	Feature D is dead, because core feature B implies feature E which is alternative to D (Test_DeadF1.xml).
Figure A.6	Feature C is dead, because it is mutually exclusive to core feature B. Feature D is (conditionally) dead, because it implies the dead feature C (Test_DeadF2.xml).
Figure A.7	Feature A is dead, because it implies feature B which is exclusive to feature A (Test_DeadF3.xml).
Figure A.8	Both features C and D are dead, because they are mutually exclusive to root A (Test_DeadF4.xml).
Figure A.9	Feature D is dead, because it is excluded by root A (Test_DeadF5.xml).
Figure A.10	Feature E is dead, because it is excluded by core feature C (Test_DeadF6.xml).
Figure A.11	Feature Bluetooth is dead, because it is excluded by core feature Carbody (/TestFeatureModels/AllDefects/Test_AllDefects.xml).
Figure A.12	Feature Manual is dead, because core feature Carbody implies feature Automatic which is alternative to feature Manual (/TestFeatureModels/AllDefects/Test_AllDefects.xml).

Table A.2: Overview of test feature models for dead features.

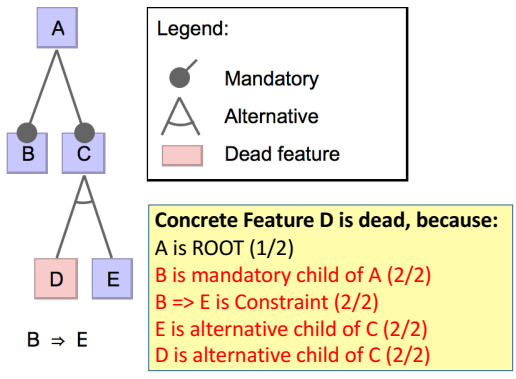

Figure A.5: Dead feature: test case 1

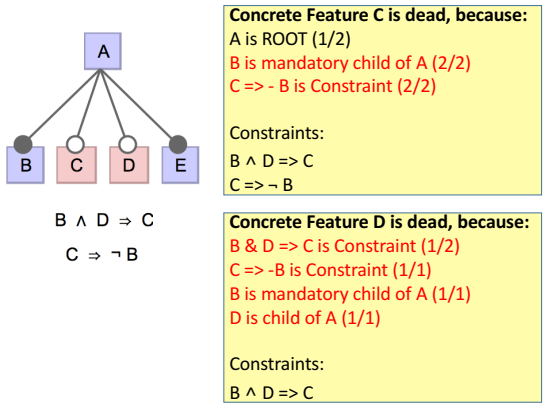

Figure A.6: Dead feature: test case 2

Test models for false-optional features

In Table A.3, we provide 6 feature models with 6 cases for a false-optional feature and a description of the defect cause (path on CD-Rom: */TestFeatureModels/FalseOptionalFeature/*).

Test models for redundant cross-tree constraints

In Table A.4, we provide 15 feature models with 24 cases for redundant constraints (path on CD-Rom: */TestFeatureModels/RedundantConstraints/*).

http://publikationsserver.tu-braunschweig.de/get/64218

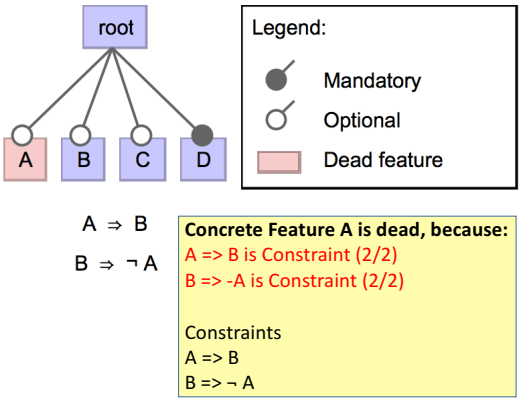

Figure A.7: Dead feature: test case 3

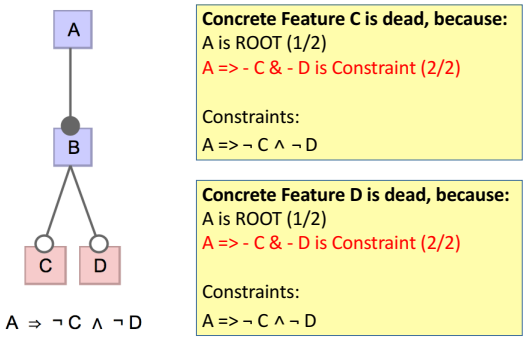

Figure A.8: Dead feature: test case 4

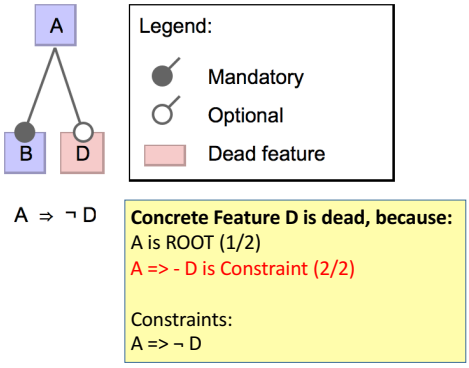

Figure A.9: Dead feature: test case 5

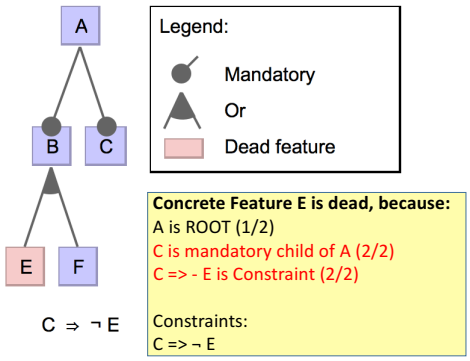

Figure A.10: Dead feature: test case 6

Reference	Description
Figure A.13	Feature B is false-optional, because it is implied by root A (Test_FO1.xml).
Figure A.14	Feature B is false-optional, because it is implied by core feature C (Test_FO2.xml).
Figure A.15	Feature C is false-optional, because core feature B implies the child feature D of C (Test_FO3.xml).
Figure A.16	Feature C is false-optional, because core feature B implies an alternative child feature E of C (Test_FO4.xml).
Figure A.17	Feature Ports is false-optional, because GPSAntenna becomes a core feature and implies feature USB which is a child feature of Ports (/TestFeatureModels/AllDefects/Test_AllDefects.xml).
Figure A.10	Feature Navigation is false-optional, because it is implied by core feature Carbody (/TestFeatureModels/AllDefects/Test_AllDefects.xml).

Table A.3: Overview of test feature models for false-optional features.

http://publikationsserver.tu-braunschweig.de/get/64218

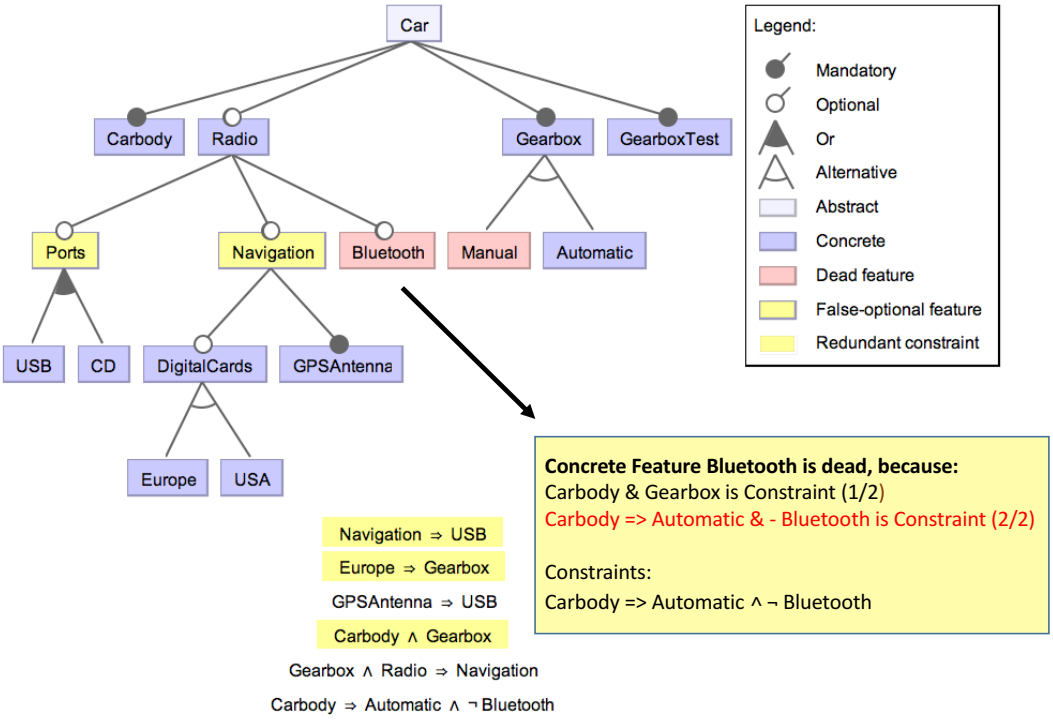

Figure A.11: Dead feature: test case 7

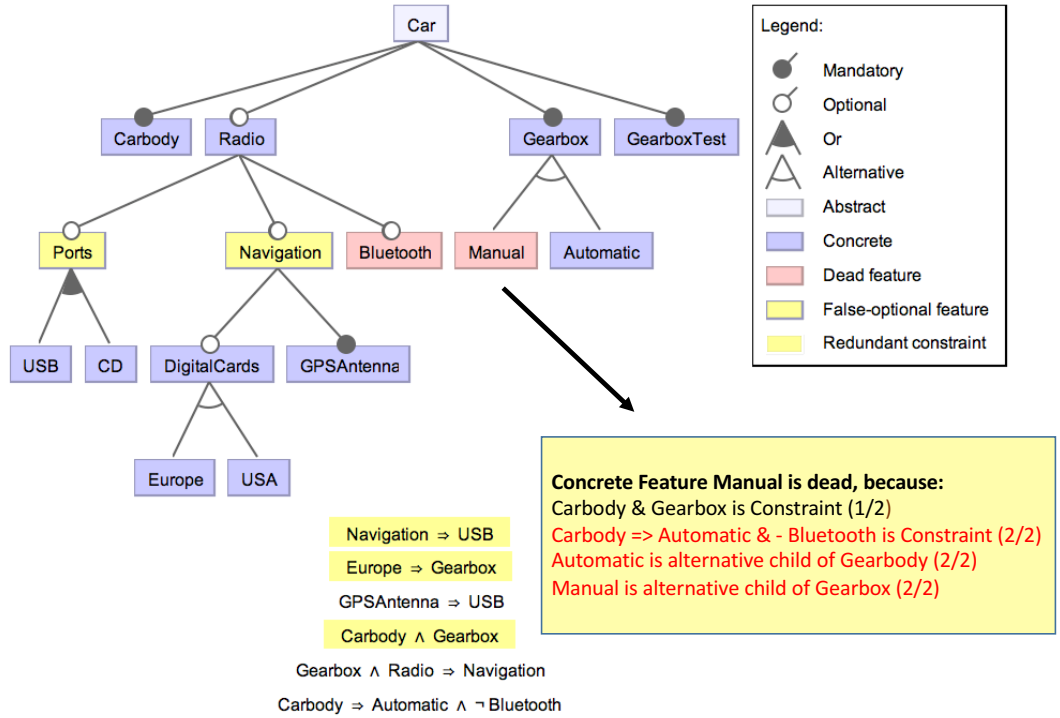

Figure A.12: Dead feature: test case 8

http://publikationsserver.tu-braunschweig.de/get/64218

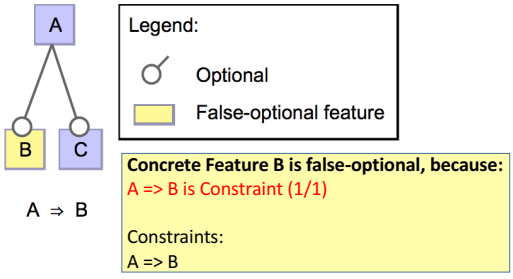

Figure A.13: False-optional feature: test case 1

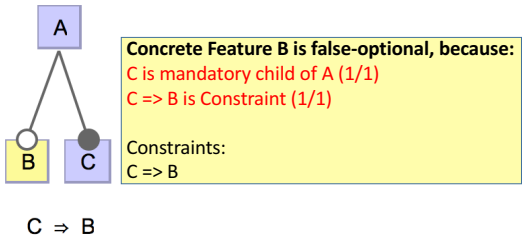

Figure A.14: False-optional feature: test case 2

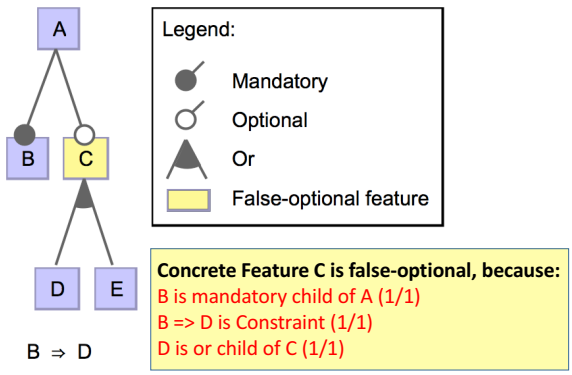

Figure A.15: False-optional feature: test case 3

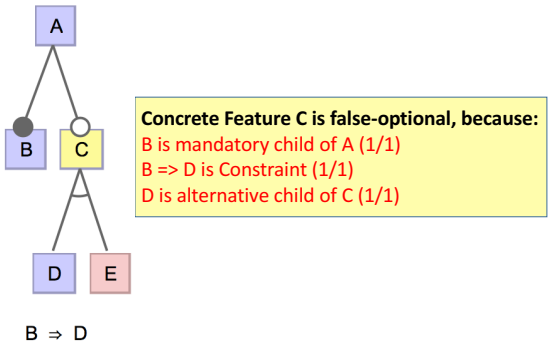

Figure A.16: False-optional feature: test case 4

http://publikationsserver.tu-braunschweig.de/get/64218

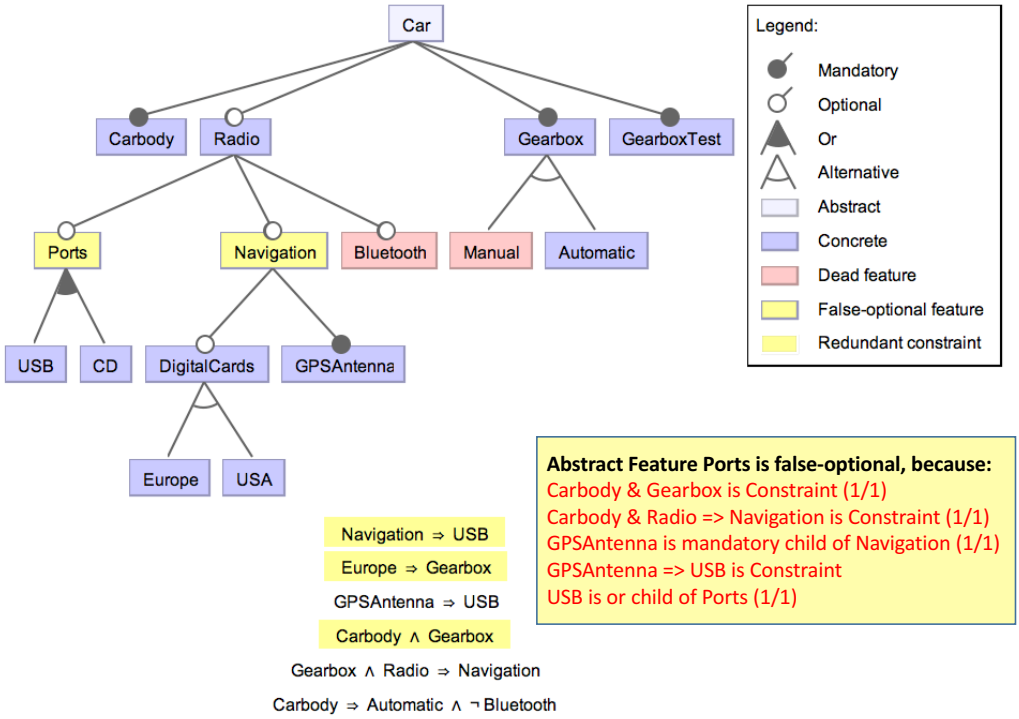

Figure A.17: False-optional feature: test case 5

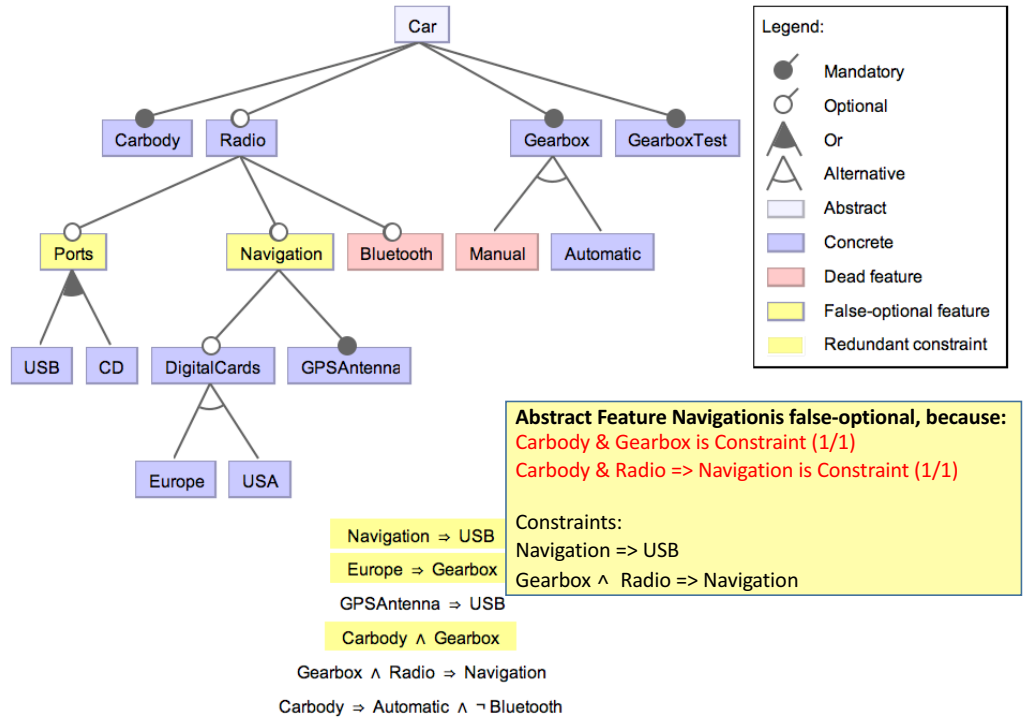

Figure A.18: False-optional feature: test case 6

http://publikationsserver.tu-braunschweig.de/get/64218

Reference	Description
Figure A.19	Constraint $\neg(B \wedge C)$ is redundant, because it represents a mutual exclusion between two alternative features B and C (Test_RC1.xml).
Figure A.20	Constraint A $\Rightarrow$ C is redundant due to transitivity (Test_RC2.xml).
Figure A.21	Constraint $\neg(C \wedge E)$ is redundant, because it represents a multiple exclusion of feature E (Test_RC3.xml).
Figure A.22	Constraint B $\Rightarrow$ C is redundant due to a multiple implication of feature E (Test_RC4.xml).
Figure A.23	Constraint D $\Rightarrow$ C is redundant, because feature D implies the core feature C (Test_RC5.xml).
Figure A.24	Constraint C $\Rightarrow D \vee E$ is redundant, because feature D and E are both alternative to each other which includes an or-relationship between both features (Test_RC6.xml).
Figure A.25	Constraint $\neg A \vee C \vee D$ is redundant, because both features C and D have the same feature parent A and are contained in an or-group (Test_RC7.xml).
Figure A.26	Constraint $B \wedge C$ is redundant, because both B and C are core features (Test_RC8.xml).
Figure A.27	Constraint $\neg(A \vee B) \wedge \neg(A \vee C)$ is redundant, because both feature B and C are child features of A and, hence, are implied by feature A (Test_RC9.xml).
Figure A.28	Constraint $\neg E \vee \neg C$ represents a multiple exclusion of feature C (Test_RC10.xml).
Figure A.29	Constraint $B \vee C \vee D$ is redundant, because all features are already contained in an or-group within the feature tree topology (Test_RC11.xml).
Figure A.30	Constraint E $\Rightarrow$ B is redundant due to transitivity (Test_RC12.xml). Here, a shorter explanation is found.
Figure A.31	Constraint Navigation $\Rightarrow$ USB is redundant, because a mandatory child GPSAntenna of a false-optional feature Navigation already implies USB. Constraint Europe $\Rightarrow$ Gearbox is redundant, because both features already appear as a conjunction in a constraint (/TestFeatureModels/AllDefects/Test_AllDefects.xml). Here, a shorter explanation has been found. A longer explanation states that both features are core features.
Figure A.32	Constraint Garbody $\wedge$ Gearbox is redundant, because both features are core features (emphasized parts) and, additionally, Carbody indirectly implies Gearbox (/TestFeatureModels/AllDefects/Test_AllDefects.xml).
	7 redundant constraints (PPU.xml)
	2 redundant constraints (SortingLine.xml)

Table A.4: Overview of test feature models for redundant cross-tree constraints.

http://publikationsserver.tu-braunschweig.de/get/64218

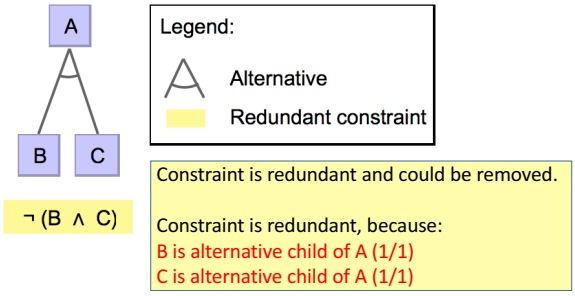

Figure A.19: Redundant cross-tree constraint: test case 1

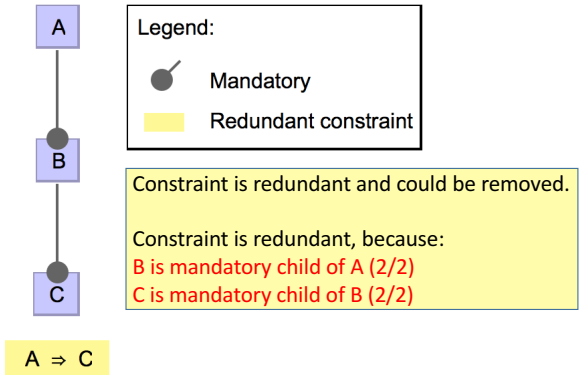

Figure A.20: Redundant cross-tree constraint: test case 2

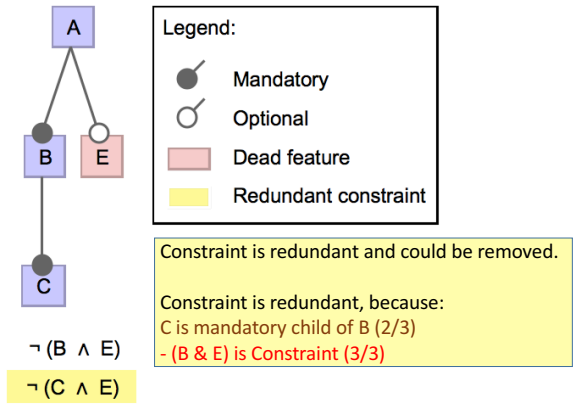

Figure A.21: Redundant cross-tree constraint: test case 3

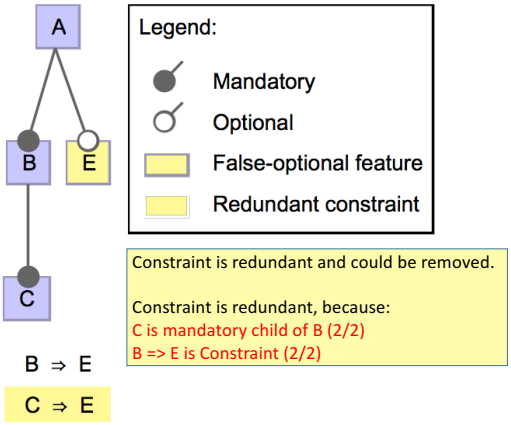

Figure A.22: Redundant cross-tree constraint: test case 4

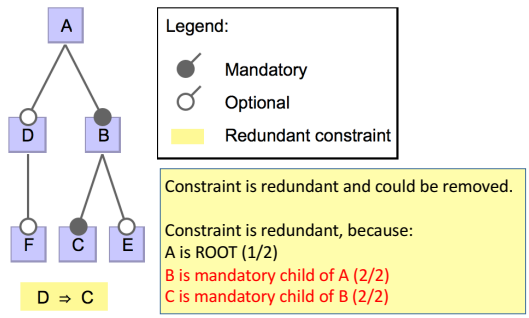

Figure A.23: Redundant cross-tree constraint: test case 5

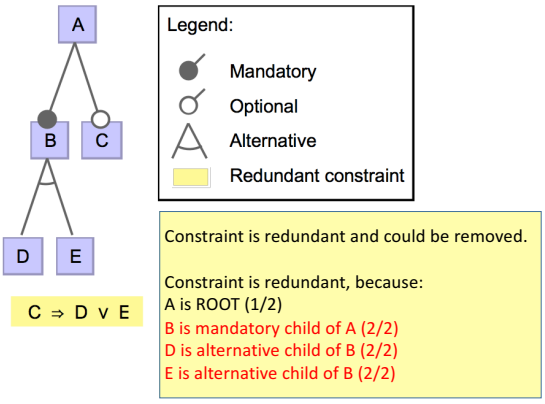

Figure A.24: Redundant cross-tree constraint: test case 6

http://publikationsserver.tu-braunschweig.de/get/64218

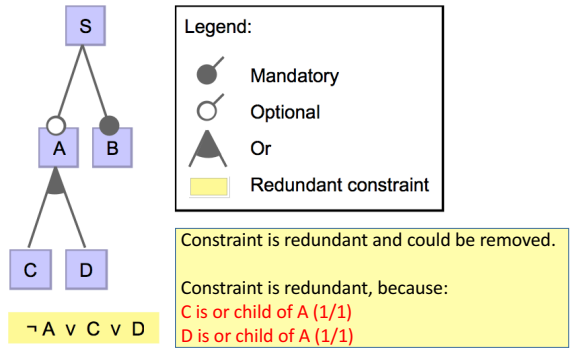

Figure A.25: Redundant cross-tree constraint: test case 7

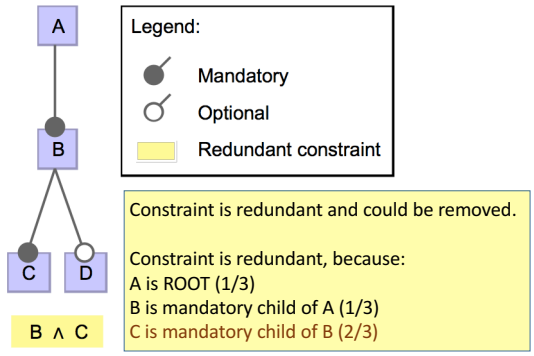

Figure A.26: Redundant cross-tree constraint: test case 8

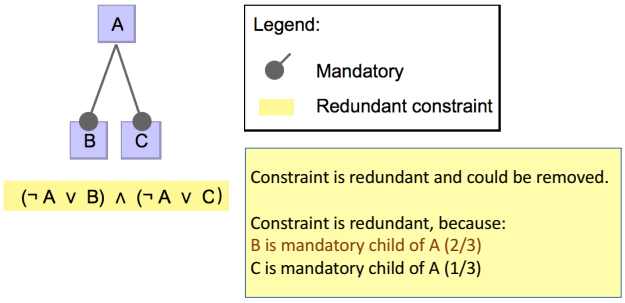

Figure A.27: Redundant cross-tree constraint: test case 9

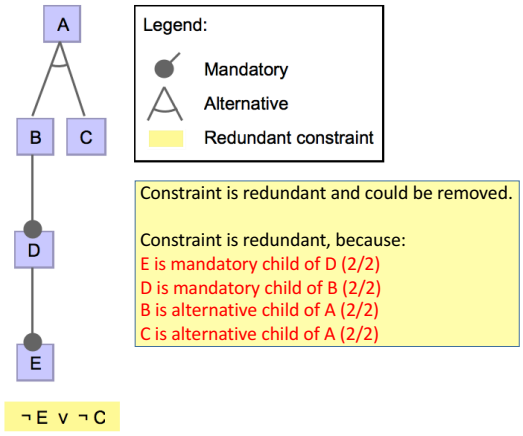

Figure A.28: Redundant cross-tree constraint: test case 10

Test models for implicit constraints

In Table A.5, we provide 5 feature models with 9 cases of implicit constraints and a description of the defect cause (path on CD-Rom: */TestFeatureModels/ImplicitConstraints/*). Model PPU has been artificially extended with cross-tree constraints leading to overall 4 implicit constraints.

http://publikationsserver.tu-braunschweig.de/get/64218

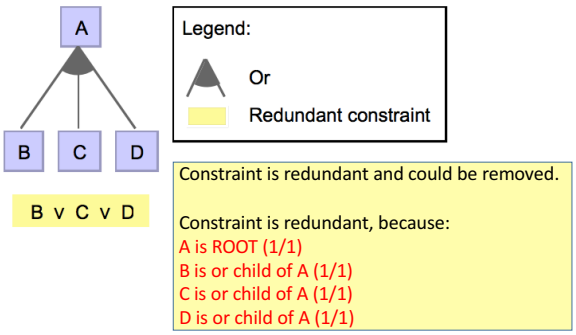

Figure A.29: Redundant cross-tree constraint: test case 11

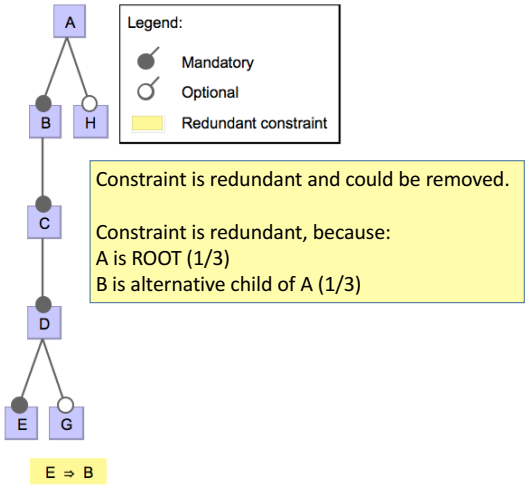

Figure A.30: Redundant cross-tree constraint: test case 12

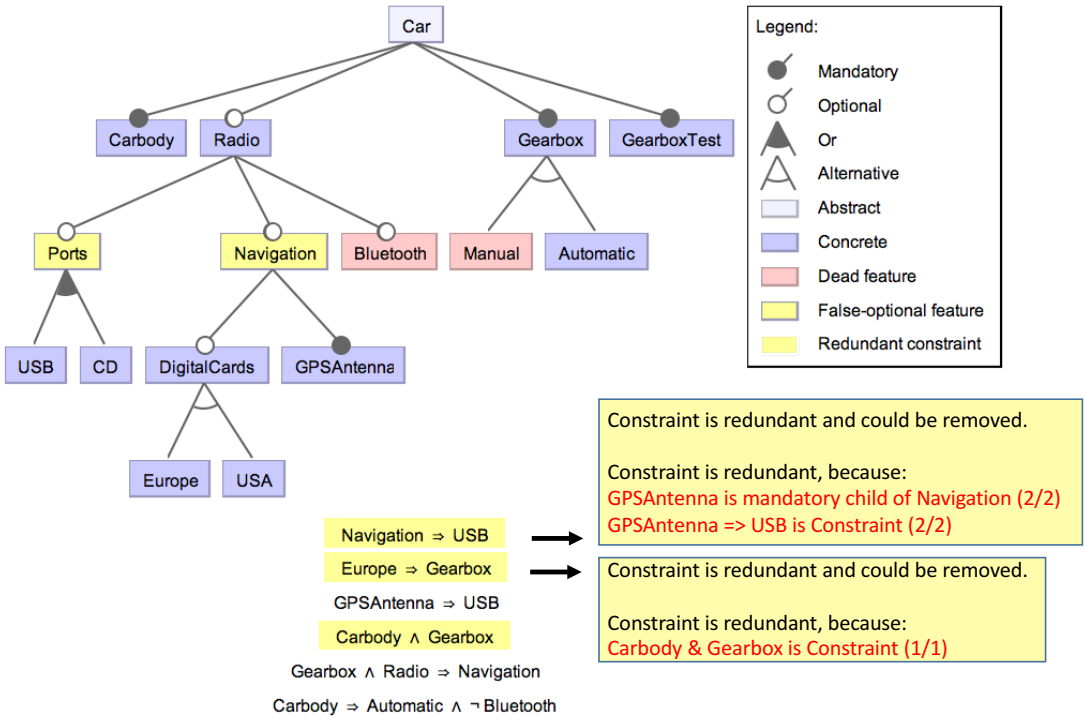

Figure A.31: Redundant cross-tree constraint: test case 13

http://publikationsserver.tu-braunschweig.de/get/64218

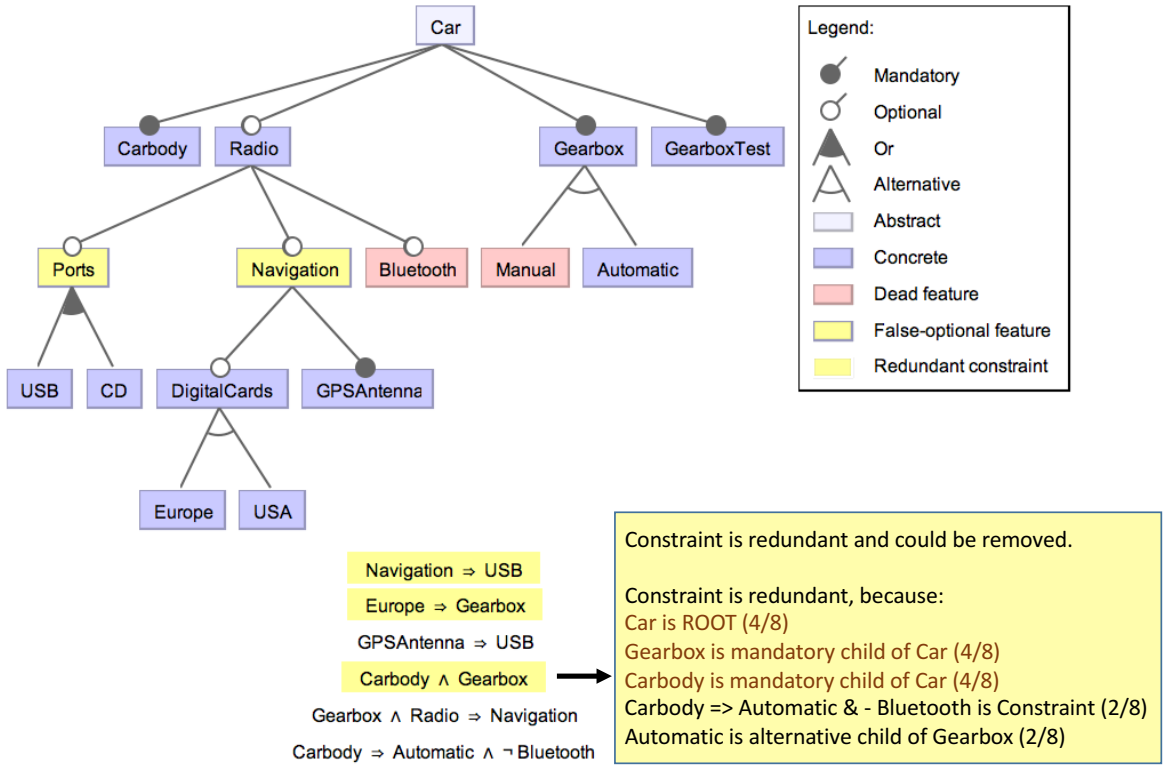

Figure A.32: Redundant cross-tree constraint: test case 14

Reference	Description
Figure A.34	C ∨ G is transitive, because feature G is implied by feature D which is already in an or-relationship to feature C (Test_IC1.xml).
Figure A.33	Constraint ¬ Bluetooth is transitive, because it is excluded by a core feature Carbody. Constraint Navigation ∨¬ Radio is transitive, because Radio already implies Navigation (/TestFeatureModels/AllDefects/Test_AllDefects.xml).
Figure A.35	Constraint D ∨¬ C is transitiv, because feature C implies feature G and feature G implies feature D, resulting in a transitive relationship between feature C and D (Test_IC2.xml).
Figure A.36	Constraint $\neg D \vee \neg C$ is transitiv, because both feature C and D imply alternative features G and H which results in feature C and D being mutually exclusive to each other(Test_IC3.xml) .
	Submodels of PPU model (# implicit constraints): PPUC(2), PPUE(1), PPUS(1) (*/PPU_Extended.xml*)

Table A.5: Overview of test feature models for implicit constraints.

http://publikationsserver.tu-braunschweig.de/get/64218

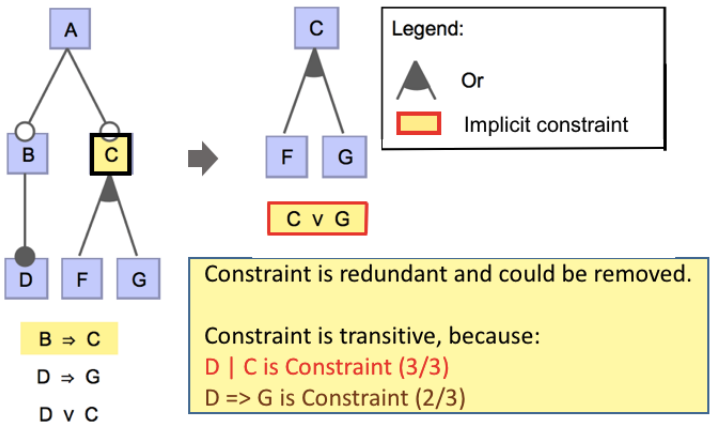

Figure A.33: Implicit constraint: test case 1

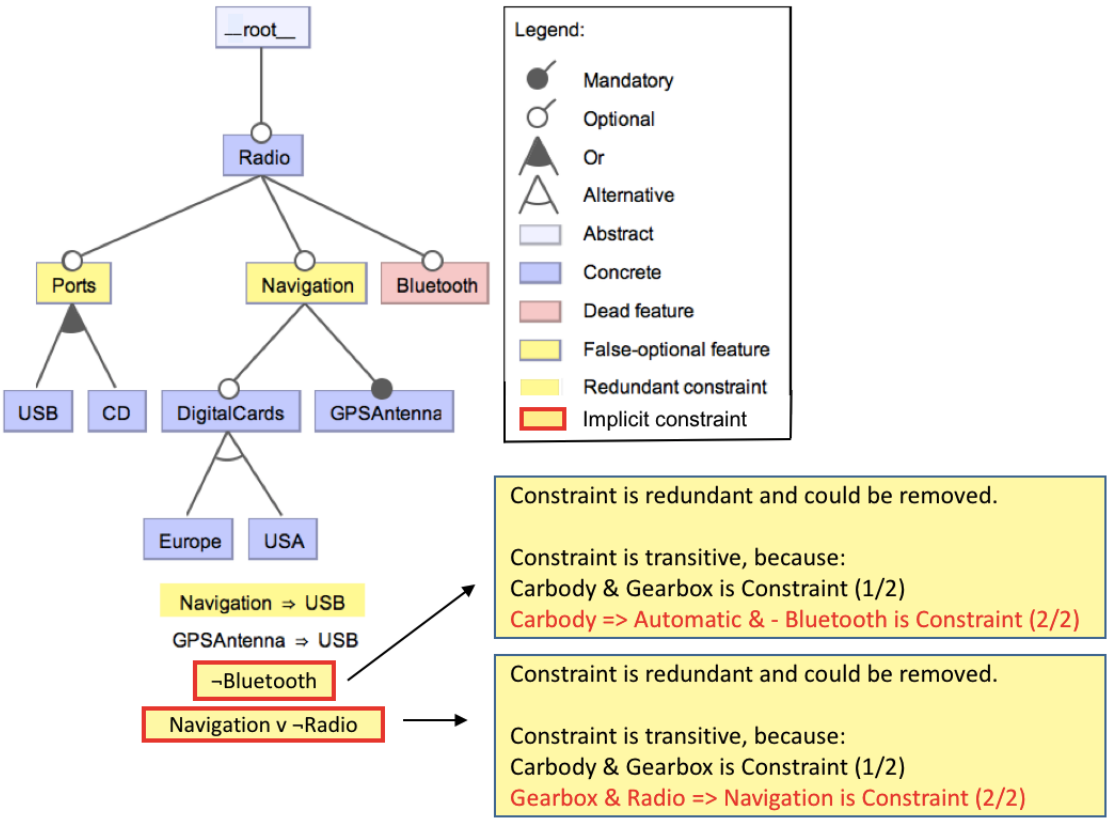

Figure A.34: Implicit constraint: test case 2

http://publikationsserver.tu-braunschweig.de/get/64218

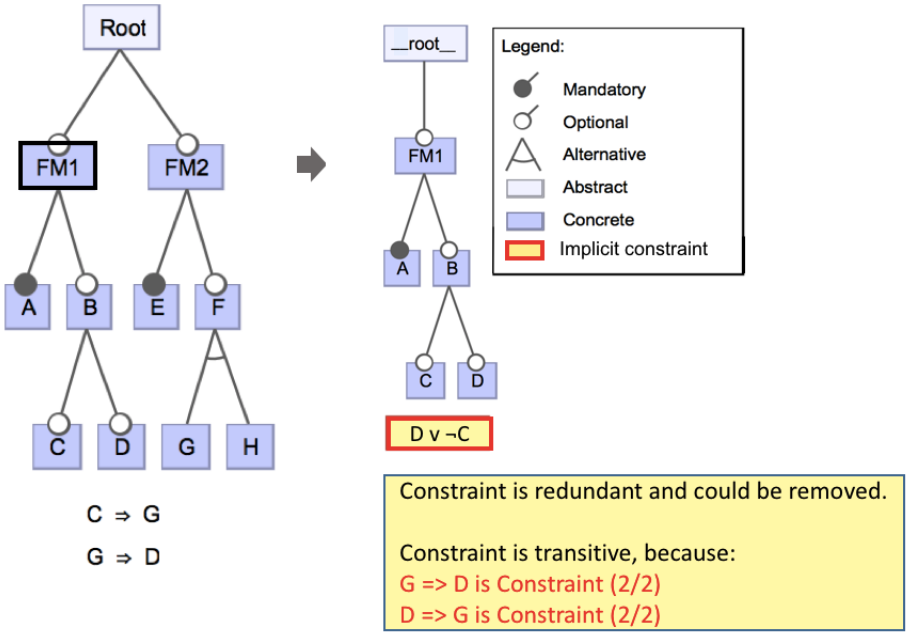

Figure A.35: Implicit constraint: test case 3

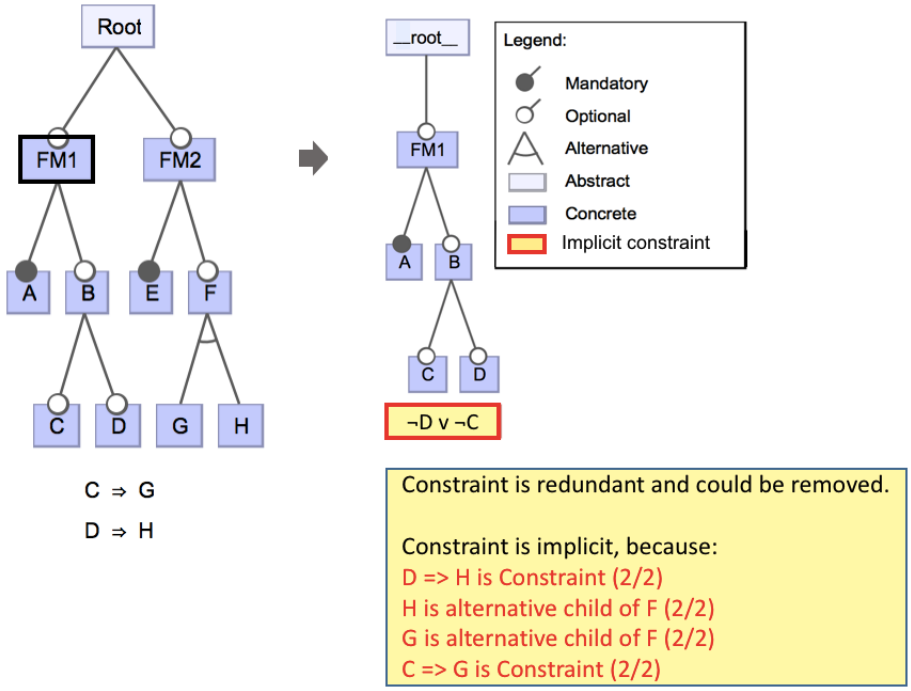

Figure A.36: Implicit constraint: test case 4

http://publikationsserver.tu-braunschweig.de/get/64218

Bibliography

[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. A Domain-Specific Language for Managing Feature Models. In *Proceedings of the Symposium on Applied Computing (SAC)*, pages 1333–1340. ACM, 2011. (cited on Page 78)

[2] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Decomposing Feature Models: Language, Environment, and Applications. In *Proceedings of the 26th International Conference on Automated Software Engineering (ASE)*, pages 600–603. IEEE/ACM, 2011. (cited on Page 19 and 78)

[3] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Slicing Feature Models. In *Proceedings of the 26th International Conference on Automated Software Engineering (ASE)*, pages 424–427. ACM/IEEE, 2011. (cited on Page 19 and 78)

[4] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer-Verlag, Berlin Heidelberg, Berlin, 2013. (cited on Page 6 and 84)

[5] Krzysztof R. Apt. *Some Remarks on Boolean Constraint Propagation*, volume 1865. Springer-Verlag, Berlin Heidelberg, 2001. (cited on Page 25)

[6] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. *Satisfiability modulo theories*, volume 185 of *Handbook of Satisfiability*, chapter 26, pages 825–885. IOS Press, February 2009. (cited on Page 76)

[7] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th International Conference on Software Product Lines (SPLC)*, volume 3714 of *Lecture Notes in Computer Sciences*, pages 7–20. Springer, 2005. (cited on Page 2, 8, 11, 19, 21, 24, 26, 28, 74, 79, and 83)

[8] Don Batory, David Benavides, and Antonio Ruiz-Cortés. Automated Analyses of Feature Models: Challenges Ahead. *Communications of the ACM (CACM)*, 2006. (cited on Page 73)

[9] David Benavides, Alexander Felfernig, José A. Galindo, and Florian Reinfrank. *Automated Analysis in Feature Modelling and Product Configuration*. Springer, Berlin, Heidelberg, 2013. (cited on Page 2)

http://publikationsserver.tu-braunschweig.de/get/64218

[10] 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, 2, 17, 18, 19, 20, and 73)

[11] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. A First Step Towards a Framework for the Automated Analysis of Feature Models. In *Managing Variability for Software Product Lines: Working With Variability Mechanisms*, pages 39–47, Washington, 2006. IEEE. (cited on Page 18 and 20)

[12] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *Proceedings of the 1st International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 129–134, Limerick, Ireland, January 2007. (cited on Page 18, 20, and 74)

[13] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wąsowski. Three cases of feature-based variability modeling in industry. In *Proceedings of the 17th International Conference of Model Driven Engineering Languages and Systems (MODELS)*, pages 302–319, 2014. (cited on Page 77)

[14] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. A simple and flexible way of computing small unsatisfiable cores in sat modulo theories. In *Theory and Applications of Satisfiability Testing–SAT*, volume 4501, pages 334–339. Springer, 2007. (cited on Page 76)

[15] Dave Clarke and José Proença. Towards a theory of views for feature models. In *Proceedings of the 1st International Workshop on Formal Methods in Software Product Line Engineering (FMSPLE)*, pages 91–98, 2010. (cited on Page 78)

[16] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA, 2001. (cited on Page 1 and 6)

[17] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposion on Theory of Computing (STOC)*, pages 151–158, 1971. (cited on Page 18)

[18] Ben Coppin. *Artificial Intelligence Illuminated*. Jones & Bartlett, 2004. (cited on Page 24)

[19] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000. (cited on Page 7)

[20] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960. (cited on Page 25)

http://publikationsserver.tu-braunschweig.de/get/64218

[21] Alexandre Dolgui, Jurek Sasiadek, Marek Zaremba, Stefan Feldmann, Christoph Legat, and Birgit Vogel-Heuser. Engineering Support in the Machine Manufacturing Domain through Interdisciplinary Product Lines: An Applicability Analysis. In *Proceedings of the 15th IFAC Symposium on Information Control Problems in Manufacturing (INCOM)*, volume 48, pages 211–218, 2015. (cited on Page 2, 4, 55, and 57)

[22] Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3):251–272, 1979. (cited on Page 24)

[23] Abdelrahman Osman Elfaki, Somnuk Phon-Amnuaisuk, and Chin Kuan Ho. Knowledge Based Method to Validate Feature Models. In *Proceedings of the 1st International Workshop on Analyses of Software Product Lines (ASPL)*, 2008. (cited on Page 75)

[24] Boi Faltings and Peter Struss. *Recent Advances in Qualitative Physics*. MIT Press, 1992. (cited on Page 27)

[25] Alexander Felfernig, David Benavides, J Galindo, and Florian Reinfrank. Towards Anomaly Explanation in Feature Models. In *Proceedings of the Workshop on Configuration, Vienna, Austria*, pages 117–124. Citeseer, 2013. (cited on Page 2 and 74)

[26] Alexander Felfernig and Monika Schubert. Fastdiag: A Diagnosis Algorithm for Inconsistent Constraint Sets. In *Proceedings of the Workshop on the Principles of Diagnosis (DX 2010), Portland, OR, USA*, pages 31–38, 2010. (cited on Page 75)

[27] Kenneth D. Forbus and Johan De Kleer. *Building Problem Solvers*. MIT Press, 1993. (cited on Page 24 and 25)

[28] Yaser Ghanam and Frank Maurer. Linking feature models to code artifacts using executable acceptance tests. In *Proceedings of the 14th International Software Product Line Conference (SPLC)*, pages 211–225. Springer-Verlag, Berlin Heidelberg, 2010. (cited on Page 77)

[29] Ofer Guthmann, Ofer Strichmann, and Anna Trostanetski. Minimal unsatisfiable core extraction for SMT. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2016. (cited on Page 76)

[30] Adithya Hemakumar. Finding Contradictions in Feature Models. In *Proceedings of the 1st International Workshop on Analyses of Software Product Lines (ASPL)*, pages 183–190, 2008. (cited on Page 28 and 73)

[31] Gerald Holl, Paul Grünbacher, and Rick Rabiser. A Systematic Review and an Expert Survey on Capabilities Supporting Multi Product Lines. In *Proceedings of the International Conference on Information and System Technology (IST)*, volume 54, pages 828–852, Newton, MA, USA, 2012. Butterworth-Heinemann. (cited on Page 9)

http://publikationsserver.tu-braunschweig.de/get/64218

[32] Institute of Automation and Information Systems. The Pick and Place Unit Demonstrator for Evolution in Industrial Plant Automation. http://www.ppu-demonstrator.org. (cited on Page 57)

[33] Mikoláš Janota and Goetz Botterweck. Formal approach to integrating feature and architecture models. In *Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 31–45. Springer-Verlag, Berlin Heidelberg, 2008. (cited on Page 77)

[34] Ulrich Junker. Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th National Conference on Artifical intelligence*, pages 167–172, 2004. (cited on Page 74)

[35] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SE Institute, 1990. (cited on Page 7)

[36] Johan De Kleer. An assumption-based truth maintenance system. *Artificial Intelligence*, 28:127–162, 1986. (cited on Page 24)

[37] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. Explaining Anomalies in Feature Models. In *Proceedings of the 15th International Conference on Generative Programming: Concepts & Experiences (GPCE) (to appear)*, 2016. (cited on Page 18, 23, 27, 31, 35, 47, 48, and 55)

[38] Dean Kramer, Christian Severin Sauer, and Thomas Roth-Berghofer. Towards Explanation Generation using Feature Models in Software Product Lines. In *Proceedings of the 9th International Workshop on Knowledge Engineering and Software Engineering (KESE9)*, 2013. (cited on Page xiii, 75, and 76)

[39] Sebastian Krieter, Reimar Schröter, Thomas Thüm, and Gunter Saake. An Efficient Algorithm for Feature-Model Slicing. Technical report, School of Computer Science, University of Magdeburg, 2016. (cited on Page 11, 17, 19, 34, and 78)

[40] Uwe Lesta, Ina Schaefer, and Tim Winkelmann. Detecting and Explaining Conflicts in Attributed Feature Models. In *Proceedings of the 6th Workshop on Formal Methods and Analysis in SPL Engineering (FMSPLE)*, pages 31–43, London, UK, 2015. (cited on Page 2 and 74)

[41] Daniela Lettner, Klaus Eder, Paul Grünbacher, and Herbert Prähofer. Feature Modeling of Two Large-Scale Industrial Software Systems: Experiences and Lessons Learned. In *Proceedings of the 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 386–395. ACM/IEEE, 2015. (cited on Page 9, 17, and 77)

[42] Mark H. Liffiton and Karem A. Sakallah. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. *Journal of Automated Reasoning*, 40:1–33, 2008. (cited on Page 75)

http://publikationsserver.tu-braunschweig.de/get/64218

[43] Mike Mannion. Using First-Order Logic for Product Line Model Validation. In *Proceedings of the 2nd International Conference on Software Product Lines (SPLC)*, pages 176–187, London, UK, 2002. Springer-Verlag. (cited on Page 11)

[44] Mike Mannion, Juha Savolainen, and Timo Asikainen. Viewpoint-Oriented Variability Modeling. In *Proceedings of the 33th International Computer Software and Applications Conference (COMPSAC)*, pages 67–72, Seattle, 2009. IEEE Computer Society. (cited on Page 78)

[45] Filip Maric. Formalization and Implementation of Modern SAT Solvers. *Journal of Automated Reasoning*, 43:81–119, 2009. (cited on Page 18)

[46] Joao P. Martins and Stuart C. Shapiro. *Reasoning in Multiple Belief Spaces*. PhD thesis, State University of New York at Buffalo, 1983. (cited on Page 24)

[47] Jens Meinicke, Thomas Thüm, Reimar Schöter, Fabian Benduhn, and Gunter Saake. An Overview on Analysis Tools for Software Product Lines. In *Proceedings of the Workshop on Software Product Line Analysis Tools (SPLat)*, pages 94–101, New York, NY, USA, 2014. ACM. (cited on Page 73)

[48] Marcílio Mendonça, Donald Cowan, William Malyk, and Toacy Oliveira. Collaborative product configuration: Formalization and efficient algorithms for dependency analysis. *Journal of Software*, 3:69–82, 2008. (cited on Page 77)

[49] Tom Mens and Serge Demeyer. *Software Evolution*. Springer-Verlag, Berlin Heidelberg, 2008. (cited on Page 1)

[50] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS*, volume 4963, pages 337–340, 2008. (cited on Page 76)

[51] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Efficient mus extraction with resolution. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 197–200, 2013. (cited on Page 76)

[52] B. Joseph Pine. *Mass Customization: The New Frontier in Business Competition*. Harvard Business School Press, 1999. (cited on Page 5)

[53] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Berlin, Heidelberg, 2005. (cited on Page 6)

[54] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987. (cited on Page 20 and 74)

http://publikationsserver.tu-braunschweig.de/get/64218

[55] LF Rincón, Gloria Lucia Giraldo, Raúl Mazo, and Camille Salinesi. An Ontological Rule-Based Approach for Analyzing Dead and False Optional Features in Feature Models. *Electronic Notes in Theoretical Computer Science*, 302:111–132, 2014. (cited on Page 2, 20, and 75)

[56] Marko Rosenmüller and Norbert Siegmund. Automating the Configuration of Multi Software Product Lines. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 123–130, Germany, 2010. Universität Duisburg-Essen. (cited on Page 9)

[57] Vladislav Rutenburg. Propositional truth maintenance systems: Classification and complexity analysis. *Annals of Mathematics and Artificial Intelligence*, 10:207–231, 1994. (cited on Page 24)

[58] Julia Schroeter, Malte Lochau, and Tim Winkelmann. Multi-Perspectives on Feature Models. In *Proceedings of the 15th International Conference on Model Engineering Languages and Systems (MODELS)*, pages 252–268. Springer-Verlag, Berlin Heidelberg, 2012. (cited on Page 78)

[59] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 667–678. ACM, 2016. (cited on Page 78)

[60] Reimar Schröter, Thomas Thüm, Norbert Siegmund, and Gunter Saake. Automated Analysis of Dependent Feature Models. In *Proceedings of the 3rd International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 9:1–9:5. ACM, 2013. (cited on Page 57)

[61] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *Journal of the ACM Computing Surveys (CSUR)*, 47(1), 2014. (cited on Page 83)

[62] Thomas Thüm, Don Batory, and Christian Kästner. Reasoning about Edits to Feature Models. In *Proceedings of the 31th International Conference on Software Engineering (ICSE)*, pages 254–264, Washington, 2009. IEEE. (cited on Page 18, 19, and 78)

[63] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming (SCP)*, 79(0):70–85, 2014. (cited on Page 2 and 4)

[64] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. Abstract Features in Feature Modeling. In *Proceedings of the 15th International Conference of Software Product Line Conference (SPLC)*, pages 191–200, Washington, DC, USA, 2011. IEEE. (cited on Page 19)

http://publikationsserver.tu-braunschweig.de/get/64218

[65] Pablo Trinidad. *Automating the Analysis of Stateful Feature Models*. PhD thesis, University of Seville, 2012. (cited on Page 19 and 74)

[66] Pablo Trinidad, David Benavides, Amador Durán, Antonio Ruiz-Cortés, and Miguel Toro. Automated Error Analysis for the Agilization of Feature Modeling. *Journal of Systems and Software*, 81(6):883–896. (cited on Page 2, 20, and 74)

[67] Thomas von der Maßen and Horst Lichter. Deficiencies in Feature Models. In *Proceedings of the Workshop on Software Variability Management for Product Derivation-Towards Tool Support*, 2004. (cited on Page 1, 11, 13, 30, and 73)

[68] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. *Experimentation in Software Engineering*. Springer-Verlag, Berlin Heidelberg, 2012. (cited on Page 70 and 71)

http://publikationsserver.tu-braunschweig.de/get/64218

http://publikationsserver.tu-braunschweig.de/get/64218

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

Braunschweig, den

http://publikationsserver.tu-braunschweig.de/get/64218