TU Braunschweig

Master's Thesis

Explaining Satisfiability Queries for Software Product Lines

Author:

Timo Günther

2017-10-17

Advisors:

Prof. Dr.-Ing. Ina Schaefer

Institute of Software Engineering and Automotive Informatics

Dr.-Ing. Thomas Thüm

Institute of Software Engineering and Automotive Informatics

Günther, Timo:
Explaining Satisfiability Queries for Software Product Lines
Master's Thesis, TU Braunschweig, 2017.

Abstract

Many analyses have been proposed to ensure the correctness of the various models used throughout software product line development. However, these analyses often merely serve to detect such circumstances without providing any means for dealing with them once encountered. To aid the software product line developer in understanding the cause of defects, a new algorithm capable of explaining satisfiability queries in a software product line context is presented in this thesis. This algorithm finds explanations by using SAT solvers to extract minimal unsatisfiable subsets from the propositional formulas that express the defects. The algorithm is applied to feature model defects such as dead features and redundant constraints, automatic truth value propagations in configurations, and preprocessor annotations that are superfluous or cause dead code blocks. Using feature models and configurations from real software product lines of varying sizes, this approach is evaluated against an existing explanation approach based on Boolean constraint propagation. The results show that Boolean constraint propagation occasionally fails to find any explanation at all but is magnitudes faster than using minimal unsatisfiable subset extractors. In response, both algorithms are combined into a single one that is as fast as Boolean constraint propagation for the cases where that finds an explanation, but also finds an explanation for all the other cases.

Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

A result of today's industrial economy is the economic imperative to cater to as many customers as possible. At the same time, the diversity of the customer demands can make a one-size-fits-all solution counter-productive. In software engineering, it is possible to accommodate these demands using a software product line [PBvdL05, vdLSR07]. The idea behind software product line engineering is to take into consideration the variability of the software platform in order to enable the deliberate and structured reuse of software artifacts such as program code. This way, fewer resources are spent on developing the commonalities of the various products more than once. Hence, product line engineering reduces the costs of the development of variable software [ABKS13, PBvdL05, vdLSR07]. In the end, by making customizable software feasible, software product line engineering results in individual needs being met more adequately.

However, with software development being difficult enough as is, adding variability to the engineering process requires additional engineering techniques [BCH15, PBvdL05, vdLSR07]. Variable software platforms need to be modeled [BRN+13, CGR+12], implemented [LAL+10], and tested [ER11] differently from single-system software. This in turn entails various analyses aimed towards making software product line engineering more robust [TAK+14].

For example, the variability of software product lines is usually modeled using feature models [KCH+90, ABKS13]. However, given the sheer complexity of larger software platforms, creating and maintaining feature models is not a trivial task. To ensure the correctness of a feature model, a wide variety of analyses may be used [BSRC10, SKT+16]. To this end, the feature model is typically transformed into a propositional formula [Bat05] which can be used to reason over the feature model [Man02, Men09]. In particular, the feature model formula can be used in a satisfiability query deciding some property of the feature model such as whether it is even possible to configure the software product line in such a way that any valid product can be obtained [BSRC10, KAT16, SKT+16]. Such satisfiability queries can be used to detect all sorts of circumstances that require some form of intervention, be it automatic or

manual, not just for feature models but for any artifact used throughout the software product line engineering process [TAK+14].

Still, detecting such a circumstance only tells the developer *that* a certain circumstance takes place but not *why*. However, figuring out the cause of the circumstance is crucial to understand why an automatic intervention had to happen or which steps need to be taken to do so manually. To assist the developer in this regard, automatically finding explanations of the circumstance can save time and effort when trying to solve an issue with the model. This is particularly important for large models as their complexity can easily be overwhelming. By pointing out which elements of the model are involved in causing the circumstance, precise explanations reduce the number of elements that need to be considered and inspected for possible changes.

Unfortunately, existing approaches for finding explanations in a software product line context [KAT16, MNSY17, Bat05, Bat05, TBD+08] suffer from a number of issues such as failing to find an explanation all the time, leading to results in unintuitive representations, or not having been evaluated for larger models. Additionally, these explanations approaches are typically limited to feature model defects even though circumstances worth explaining can also arise in configurations as well as code annotated with preprocessor directives. Indeed, to the knowledge of the author, this thesis constitutes the first attempt to find explanations for code annotated with preprocessor directives automatically.

1.1 Goals

The goal of this thesis is the development of a generic explanation algorithm capable of finding explanations for any circumstance in a software product line context that can be expressed as a satisfiability query. Relevant criteria of the algorithm are runtime performance, correctness, completeness, and the quality of the explanations it finds. The quality of an explanation is measured by its length, as a short explanation is in general easier to understand than a longer one.

The completeness requirement is in response to the incompleteness of the algorithm proposed by Ananieva [Ana16, KAT16], which is based on Boolean constraint propagation and fails to find any explanation in certain cases. Additionally, that algorithm is designed with only feature model defects in mind, whereas the algorithm developed in this thesis should be applicable to any use case in software product line engineering.

To demonstrate the applicability of the algorithm devised in this thesis, it is applied to several use cases in software product line engineering. In particular, it is applied to the following three use cases. First, it is applied to the feature model defects that can already be explained using Ananieva's algorithm, which are dead features, false-optional features, redundant constraints, implicit constraints, and void feature models. Next, it is applied to configurations in order to explain why certain features have to be automatically selected or automatically unselected for a valid configuration. Finally, it is applied to code annotated with preprocessor directives, specifically to explain why a given annotation is superfluous or causes a dead code block.

1.2 Structure

The remainder of this thesis is structured as follows. In general, where applicable, each section contains subsections dealing separately with the three concrete use cases of feature model defects, configurations, and code annotated with preprocessor directives. First, Chapter 2 covers the preliminaries. Next, in Chapter 3, the explanation algorithm is developed conceptually. To this end, the underlying idea of extracting minimal unsatisfiable subsets is contrasted against the similar yet incomplete approach by Ananieva based on Boolean constraint propagation [Ana16]. The implementation as part of the software product line development tool FeatureIDE [MTS+17, TKB+14] is detailed in Chapter 4. To ensure that the requirements of the algorithm are met, the implementation is evaluated using models from real software projects in Chapter 4. This involves comparing it against Ananieva's existing algorithm based on Boolean constraint propagation. Afterwards, the commonalities and differences between the approach contributed in this thesis and other works are discussed in Chapter 6. Afterward, a conclusion of the findings of this thesis is drawn in Chapter 7. Finally, Chapter 8 lists ideas for future work based on this thesis.

2. Background

This chapter covers the theoretical foundation of this thesis. The content explained in this chapter is critical to the remainder of this thesis and must therefore be understood before continuing with the following chapters.

In particular, Section 2.1 discusses the problem of Boolean satisfiability. After its definition, automated solution approaches to it are outlined in Section 2.1.1, followed by an definition of minimal unsatisfiable subsets in Section 2.1.2. Software product lines are introduced in Section 2.2. The concrete software product line artifacts considered in this thesis are tied together through the software product line engineering process outlined in Section 2.2.1. Those artifacts are feature models (Section 2.2.2), configurations (Section 2.2.3), and code with preprocessor directives (Section 2.2.4). Finally, Boolean satisfiability and software product lines are united for software product line analysis in Section 2.3. The analyses revolve around the three artifacts just mentioned and are for feature model defects (Section 2.3.1), configuration selections (Section 2.3.1), and preprocessor directives (Section 2.3.3. An approach for generating explanations for these three follows in the next chapter.

2.1 Boolean Satisfiability

A Boolean formula is a formula over Boolean variables and Boolean formulas [End01]. When each variable is interpreted to be either true or false, a formula can be evaluated by replacing each literal (an occurrence of a variable) with the corresponding truth value and applying the contained functions. Classically, these functions can be any of the following, listed in descending order of binding strength, meaning the functions listed first bind the operators more tightly unless overwritten with parentheses:

Negation $\neg\alpha$ evaluates to true iff the Boolean formula α evaluates to false.

Conjunction $\alpha_1 \wedge \cdots \wedge \alpha_n$ evaluates to true iff each of the n Boolean formulas α_i evaluates to true.

Disjunction $\alpha_1 \vee \cdots \vee \alpha_n$ evaluates to true iff at least one of the n Boolean formulas α_i evaluates to true.

Implication $\alpha \Rightarrow \beta$ evaluates to true iff $\neg\alpha \vee \beta$ evaluates to true.

Equivalence $\alpha \Leftrightarrow \beta$ evaluates to true iff $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$ evaluates to true.

Given such a Boolean formula, the Boolean satisfiability problem (or "SAT" for short) asks whether an interpretation exists under which it evaluates to true. If so, the formula is said to be satisfiable. For instance, the formula $A \wedge B$ is satisfiable since it evaluates to true when interpreting both variables A and B as true. In contrast, $A \wedge \neg A$ is unsatisfiable since no interpretation exists under which the formula would evaluate to true.

2.1.1 SAT Solving

Even though SAT is an inherently difficult problem to solve due to being NP-complete [Coo71], a lot of research is conducted to come up with increasingly efficient approaches for tackling the problem [CESS08]. Indeed, there are even competitions dedicated to measuring and comparing state-of-the-art SAT solvers [JLBRS12].

Most SAT solvers operate on formulas in conjunctive normal form, which every Boolean formula can be transformed to [EMS07, Tse68] in a process called clausification. A formula is in conjunctive normal form if it is a conjunction of disjunctions of literals, where a literal is a variable or a negated variable [CESS08]. This conjunctive normal form makes it easier to reason over the formula. Indeed, being a conjunction of clauses, it must evaluate to false as soon as any clause evaluates to false, since every clause must evaluate to true for the conjunction to evaluate to true. Analogously, each clause evaluates to true as soon as any of the literals it contains evaluates to true.

These facts can be exploited when trying to find an interpretation under which each clause and therefore the formula evaluates to true. In Boolean constraint propagation [McA90], the first step is to identify clauses that contain exactly one variable with no truth value assignment and otherwise only literals that evaluate to false. These so-called unit-open clauses are important as they can only ever evaluate to true if the remaining literal evaluates to true. As such, the corresponding truth value is derived by interpreting the variable to be false if the literal is negated and true otherwise. This entire process can be repeated until all clauses evaluate to true, in which case a satisfying interpretation is found, or until no unit open-clause is left or deriving a truth value would cause a contradiction in another unit-open clause, in which case no satisfying interpretation is found.

However, a satisfying interpretation might exist even if Boolean constraint propagation does not manage to find it. After all, at any point where unit propagation requires picking from multiple instead of just a single unit-open clauses, it would be possible to choose a different one, which might lead to a satisfying interpretation down the line. Likewise, not all hope is lost if there are no unit-open clauses left; it would be possible to give an interpretation to any variable with no truth value assignment and see if that results in a unit-open clause. Indeed, this is called the

Davis-Putnam-Logemann-Loveland procedure [DLL62, DP60, NOT06], which keeps track of these decisions and backtracks upon failure to try out another path in the decision tree.

That is how SAT solvers used to work for a long time [CESS08]. More modern SAT solvers such as Sat4J [LBP10] typically use conflict-driven clause learning engines that also add new, internal clauses on the fly [ES04, MMZ+01]. In any case, eventually, the SAT solver provides proof of the satisfiability of the formula in the form of a so-called model, which is a set of variables which, when interpreted as true and the other ones as false, causes the formula to evaluate to true.

2.1.2 Minimal Unsatisfiable Subsets

Some SAT solvers not only provide functionality for deciding whether a Boolean formula is satisfiable, but also for analyzing why it is not. This is accomplished by extracting a minimal unsatisfiable subset from the unsatisfiable formula [BLMS12]. A minimal unsatisfiable subset is any subset of the clauses of the formula in conjunctive normal form for which any strict subset is satisfiable. There might in fact be multiple minimal unsatisfiable subsets for a given unsatisfiable formula. In contrast, the smallest one of them is called the minimum unsatisfiable subset [LMS04], which is more difficult to compute, for instance naïvely by generating all minimal unsatisfiable subsets [dlBSW03, LS08] and choosing the smallest one.

The usefulness of minimal unsatisfiable subsets lies in acting as an explanation for the unsatisfiability of a formula [GMP08a]. If a problem can be formulated as a satisfiability query, a minimal unsatisfiable subset pinpoints one of the sets of clauses that in tandem cause the unsatisfiability and therefore the problem.

There are several approaches for extracting a minimal unsatisfiable subset [BLMS12]. The approach may be destructive, which means clauses are removed until further removal would make the subset satisfiable. Conversely, constructive approaches add clauses until the subset becomes unsatisfiable. Finally, both approaches can be combined for a dichotomic approach.

In all approaches, multiple SAT solver calls are required. More importantly, all these satisfiability queries are fairly similar and might even differ by as little as only one clause at a time. Therefore, it makes sense to exploit the similarities between these satisfiability queries using incremental SAT solving [ES03, ALS13]. Instead of simply forgetting the state of the previous queries, only the now affected parts need to be solved, which drastically improves the performance and thus allows extracting minimal unsatisfiable subsets from bigger formulas.

2.2 Software Product Lines

As an example for variability, modern cars are often highly customized to meet specific customer requirements [PBvdL05, vdLSR07]. For instance, a car may come with or without special features such as GPS, Bluetooth, and an automatic as opposed to a manual gearbox. Of course, each possible car product requires a different software system to control the car.

Traditionally, each product would be developed from ground up. However, as the variability increases, so does the development effort when not acknowledging and leveraging the commonalities. Therefore, this single-system software engineering quickly reaches its limits. The example above with merely 3 independent features already results in 8 possible configurations. In a more complex example such as the Linux kernel [SSSPS07] with over 10,000 features [TLSSP11], there are far more possible configurations than atoms in the known universe. Providing a product for every possible configuration is therefore hardly feasible using single-system software engineering.

The difficulty of dealing with variability in software systems is handled using software product line engineering. A software product line is a family of software systems that are at the same time different enough from each other to be considered distinct products but also similar enough to profit from planned reuse of software artifacts [ABKS13]. A more liberal interpretation sees a software product line as any software platform [ML97] for which many differing products sharing a common code base exist [PBvdL05, TAK+14]. The core principal in either case is the variability between the products, that is the many interrelated software systems.

Identifying and exploiting this variability is the fundamental advantage of software product line engineering over single-system software engineering [ABKS13, PBvdL05]. In the ideal case, both the core system, that is the part of the software platform that is shared by all products, and each variation point are only implemented once, thereby greatly reducing the development effort.

Thus, software product line engineering is a powerful extension of software engineering that can be used for a number of advantages if the software system contains enough variability. In particular, when compared to single-system software engineering, the development time and costs can be reduced significantly while also improving the quality [ABKS13, PBvdL05, vdLSR07]. Additionally, it allows for individual needs to be met more adequately.

2.2.1 Domain Engineering and Application Engineering

Figure 2.1 shows how the software product line engineering process is realized in two orthogonal dimensions [ABKS13]. The first dimension is domain engineering versus application engineering. Domain engineering is concerned with the domain [CE00], that is the set of all products, whereas application engineering focuses on a single product [PBvdL05]. Shifting effort from application engineering to domain engineering is how software product line engineering makes it possible to see the big picture and reuse software artifacts in a planned as opposed to an opportunistic manner. The other dimension is problem space versus solution space. In problem space, the perspective of the stakeholders is assumed by thinking in terms of requirements and features, whereas the solution space is targeted at the developers paying attention at concrete software artifacts and how they can be fit together to create working products.

The first three of the four emerging tasks of software product line engineering are explained in more detail in the following subsections. In particular, domain analysis is covered in Section 2.2.2, requirement analysis in Section 2.2.3, and domain implementation in Section 2.2.4. Product derivation is not relevant to this thesis.

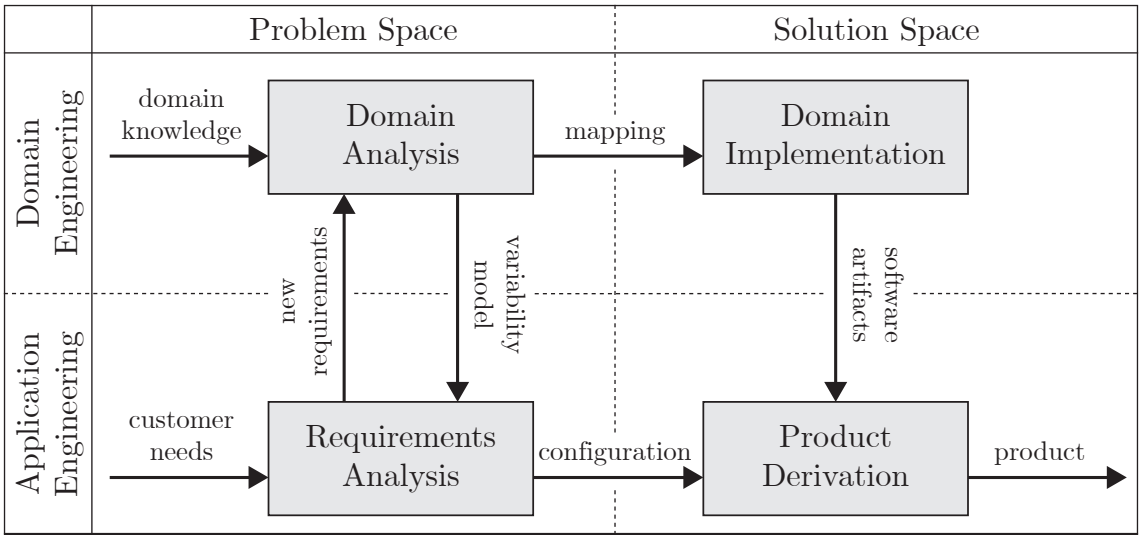


Figure 2.1: The software product line engineering process [ABKS13, adapted from p. 20].

2.2.2 Feature Models

As shown in the top left of Figure 2.1, domain engineering in the problem space is called domain analysis [ABKS13]. During domain analysis, the scope of the domain [CE00], that is which products are even considered part of the product line, is defined. The result is documented in a variability model [BRN+13, CGR+12], which may come in many different forms such as an orthogonal variability model [PBvdL05], a decision model [SRG11], or a feature model [KCH+90]. However, in the context of this thesis, only feature models are considered due to their widespread use [BRN+13, CGR+12].

As the name suggests, a feature model contains features. A feature is "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems" [KCH+90] according to its original definition in the software product line context, or, more concisely put, "an increment of program functionality" [Bat05]. The idea is to partition the software platform's variation points into cohesive units called features. Each product can then be identified by the features it contains versus the ones it does not, which is called a configuration (cf. Section 2.2.3).

However, since not every configuration might be valid, the feature model defines which products are part of the product line [Bat05, KCH+90]. The most basic way to do this is to enumerate all valid configurations [SKT+16], which is obviously unfeasible for large product lines.

Alternatively, the feature model may be expressed as a Boolean formula [Bat05, Man02]. To this end, each feature is mapped to exactly one Boolean variable. The truth value interpretation of the variable corresponds to the selection status of the feature. In other words, if the interpretation of a variable in the formula is true, the feature is considered to be part of the current configuration, and likewise it is not if it is false. While this representation is useful to enable reasoning over the variability with tried and tested tools [Men09], i.e., SAT solvers, it is still difficult to read and maintain.

As such, feature models typically come in the more readable form of feature diagrams. There are in turn varying notations for feature diagrams [BSRC10]. Throughout this thesis, the notation of FeatureIDE [TKB+14] is used for illustration purposes.

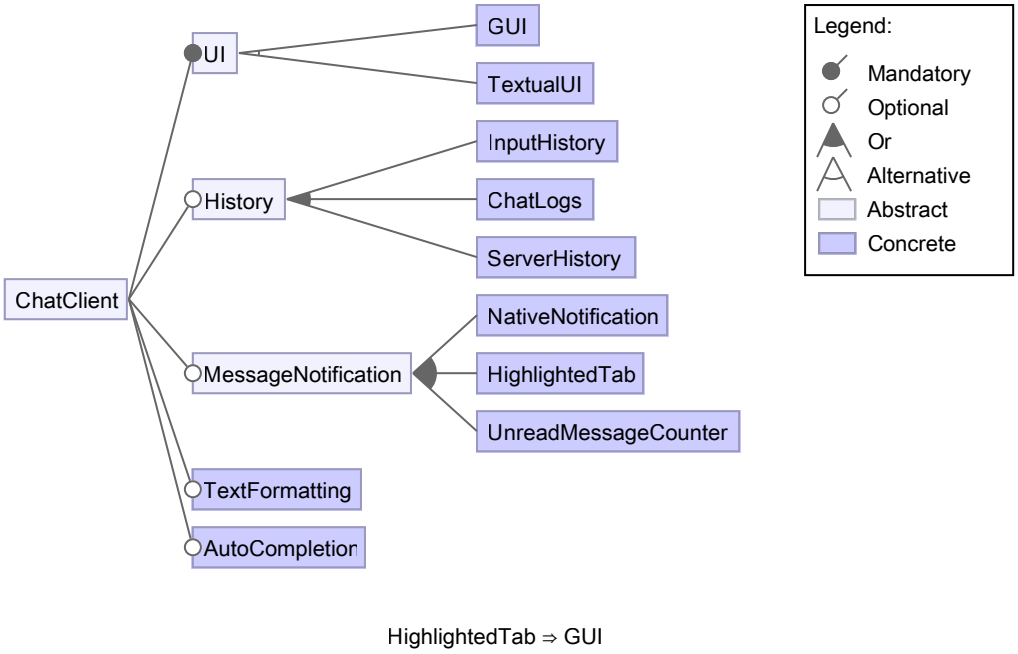


Figure 2.2: An exemplary feature diagram for a chat client.

Figure 2.2 is an example of a feature diagram for a chat client. The chat client always comes with a user interface, which may be either a graphical or textual one but not both. It provides a number of features providing access to data past its usual lifespan, specifically a history of user input, chat logs for messages sent to and received from other users, and a list of the servers that were visited in the past. Any of these three features may be used with or without any of the other history features. The same is true for the notifications that a new message was received, which may be any combination of notifications specific to the operating system, having the tab of the program blink, or displaying the number of unread messages in the program's title bar. Finally, text may or may not be formatted in other font styles and colors and user input may or may not be automatically completed. It should be noted that functionality which is not subject to variability, meaning core features such as being able to send and receive messages, does not need to be modeled.

This feature diagram showcases all additional structures that may be enforced on the features to express the relationship between them [ABKS13, BSRC10, KCH+90]:

Optional An optional child may or may not be selected if the parent is selected. However, as with all child relationships, the parent must be selected if the child is.

Mandatory A mandatory child must be selected if and only if the parent is selected.

Or In an or-group, at least one child must be selected if the parent is selected.

Alternative In an alternative group, exactly one child must be selected if the parent is selected.

In addition to being part of a child relationship or a group, a feature may be either concrete or abstract [TKES11]. A concrete feature is considered in the implementation, whereas an abstract feature is not. Finally, the feature model may contain cross-tree constraints (henceforth simply "constraints"), which are Boolean formulas (containing variables with the semantics outlined above) further restricting the set of valid configurations. Constraints are used to add restrictions that cannot be expressed using any of the previously mentioned syntax. In fact, the sole constraint in Figure 2.2 (that is that the selection of feature $HighlightedTab$ implies the selection of feature GUI) is functionally a requires-relationship, which would be written as an arrow from one feature to the other in some other notations [BSRC10].

Another advantage of the feature diagram notation is that it is easily convertible to a Boolean formula for further analysis such as whether it describes any valid configurations at all [Bat05, Man02]. The formula fm describing the feature model and therefore all valid configurations is:

$$r \wedge (\bigwedge_{s \in S} s) \wedge (\bigwedge_{c \in C} c)$$

Where r is the root feature, S the set of all structural elements, and C the set of all constraints. Each structural element $s \in S$ is transformed as following:

Optional $f \Rightarrow p$, where f is the optional child and p its parent.

Mandatory $f \Leftrightarrow p$, where f is the mandatory child and p its parent.

Or $p \Leftrightarrow \bigvee_{f \in F} f$, where F is the set of all children of the or-group and p the parent.

Alternative $(p \Leftrightarrow \bigvee_{f \in F} f) \wedge \bigwedge_{\{a,b\} \subseteq F, a \neq b} \neg a \vee \neg b$, where F is the set of all children of the alternative group and p the parent.

2.2.3 Configurations

As depicted in Figure 2.1, after capturing the variability in a feature model during domain analysis, switching from domain engineering to application engineering mandates a requirements analysis [ABKS13]. To recall, application engineering means that this task is now about a single product instead of all of them. Due to the feature model abstraction, the customer only needs to select the features of the desired product. This set of selected features is called a configuration.

An example of a configuration of the aforementioned chat client feature model from Figure 2.2 is shown in Figure 2.3. It refers to the aforementioned feature model for a chat client from Figure 2.2 and as such specifies a concrete chat client product. In this configuration, only two concrete features are selected: Firstly, the choice between a textual and a graphical user interface went in favor of the graphical one,

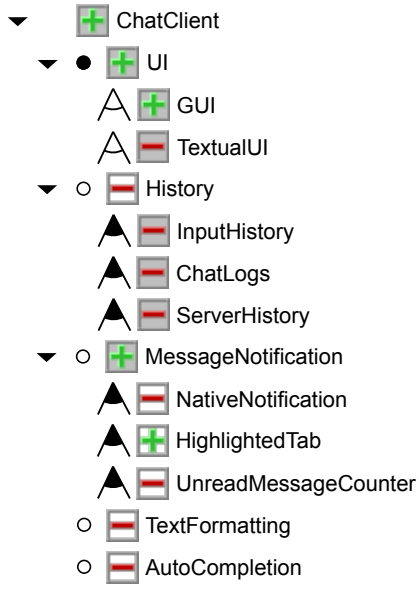


Figure 2.3: An exemplary configuration of a chat client. A green plus sign denotes the selection of a feature, whereas a red minus sign denotes its deselection. A gray background denotes that the selection or deselection of the feature is necessary for validity given the other, user-made choices (cf. Section 2.3.2).

and secondly, the notification of new messages is accomplished by highlighting the tab of the program but none of the other means. Indeed, all of the other concrete features in the feature model are unselected and therefore not part of the specified product.

The process to arrive at such a configuration is an incremental one. For each customer requirement, the feature model is consulted. If a corresponding feature is already contained in the feature model, it is selected to be part of the configuration. On the other hand, if the customer requires an unprecedented feature or a combination that is currently considered invalid, the feature model may or may not be updated to meet the newly required variability depending on whether it fits in the scope of the project or not. If afterwards all required features are included in the feature model and the combination is valid, the configuration is complete.

Then, all that is left is to build the respective product as dictated by the configuration. From that point on, it is all about the solution space. The concrete software artifacts obtained through domain implementation (explained in Section 2.2.4) are combined during product derivation (depicted in the bottom right of Figure 2.1) [ABKS13, CE00]. Product derivation might require additional implementation effort specific to that product or, in the best case, constitutes only the press of a button. In any case, it accomplishes the final goal of generating the desired product.

2.2.4 Implementation Using Preprocessors

The only step depicted in Figure 2.1 that is in solution space and relevant to this thesis is the domain implementation [ABKS13]. Its goal is the development of software artifacts (especially code) from which every valid product can be derived. The

variability can be realized through various means, be it annotative (such as preprocessors), i.e., removing all artifacts that do not belong to the product, compositional (such as aspect-oriented programming), i.e., adding together all those that do, or transformational (such as delta-oriented programming), i.e., a combination of the two.

Most commonly, the domain implementation is achieved through the annotative mechanism of preprocessors [LAL+10]. Historically intended for simple metaprogramming, they can also be used in software product line engineering. To this end, the code is annotated with preprocessor directives, which are if-statements containing variables that refer to features from the feature model. These directives evaluate to true or false depending on the configuration. If true, the annotated code block remains part of the product to be derived, otherwise it is removed. During product derivation, the preprocessed code then only contains the parts necessary for the product and can be compiled as usual.

```
1  public class ChatClient {
2      public void onMessageReceived(ChatMessage msg) {
3          printMessage(msg);
4          // #if MessageNotification
5          if (!isMessageRead(msg)) {
6              // #if NativeNotification
7  //@          displayNativeNotification(msg);
8              // #endif
9              // #if HighlightedTab
10             highlightTab(msg);
11             // #endif
12             // #if UnreadMessageCounter
13 //@          increaseUnreadMessageCounter(msg);
14             // #endif
15         }
16         // #endif
17     }
18 }
```

Listing 2.1: Java code for a chat client annotated with Antenna preprocessor directives

Listing 2.1 is an example of Java code annotated with preprocessor directives for the preprocessor Antenna, which is one of the three preprocessors currently supported by FeatureIDE [MTS+16], next to CPP and Munge. The variables refer to features from the chat client feature model from Figure 2.2 with its configuration from Figure 2.3. The code itself is an excerpt of the part of the application that reacts to incoming chat messages. First, being a core functionality not subject to variability, the message is always shown to the user. The next code, which reacts to unread messages, is only included if any unread message notification features are selected. Since only `HighlightedTab` is selected in the configuration, only its code block, that is the call to the method `highlightTab(ChatMessage)`, is part of the preprocessed code. The other two notification features are unselected and therefore their code blocks are commented out by the preprocessor such that the compiler ignores these

code blocks. As a result, when compiled, the preprocessed code for this chat client produces a product that adheres to the specified configuration.

2.3 Software Product Line Analysis

The knowledge of the variability of a software product line may also be incorporated during analysis of the software platform [PBvdL05, TAK+14]. Whereas single-system software engineering would require every single configuration to be built and tested in isolation, software product line engineering can greatly reduce the effort of quality assurance by sampling the configurations more cleverly [ER11]. For instance, for configuration coverage, configurations are selected until all variation points are covered [TLD+11]. Alternatively, taking into account the possibility of feature interactions [CKMRM03, KWG04], correctness is assured up to a certain order using combinatorial interaction testing [AHKT+16, CDS08, JHF12]. Additionally, if a certain feature is known to be used in many configurations, it may be tested more thoroughly. In any case, this is obviously more efficient than testing every possible configuration and more effective than testing only a single standard configuration.

However, before testing is even necessary, other analyses can be employed to aid development and prevent defects [TAK+14]. As with testing in particular, analyses in general profit from taking into account the unique qualities of software product lines instead of simply scaling single-system analyses to the entirety of the products.

The next sections explain the analyses covered by this thesis: analysis of defects in feature models in Section 2.3.1, analysis of configurations in Section 2.3.2, and family-based static analysis in Section 2.3.3.

2.3.1 Feature Model Defects

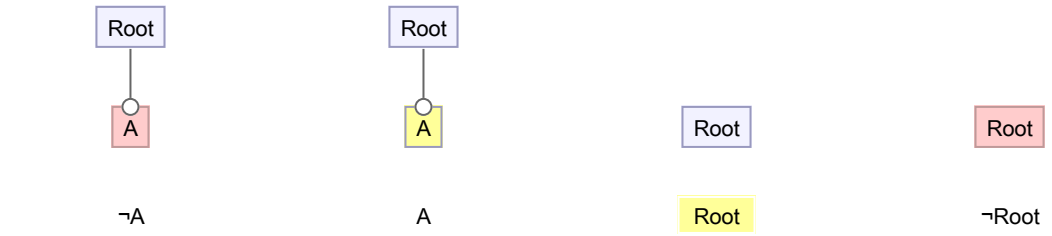
Unfortunately, due to unforeseen evolution of the product line or simply the sheer complexity of the variability to be modeled [BCH15], defects may arise in the feature model [BSRC10, vdML04], which reduces the feature model's expressiveness and may hint at a misunderstanding of the variability. This thesis focuses on the following defects [MTS+17]:

Dead feature A feature is dead iff it is not part of any valid configuration of a non-void feature model [BSRC10, KAT16, SKT+16].

False-optional feature A feature of a non-void feature model is false-optional iff it is selected whenever its parent is selected even though it is not declared mandatory [KAT16].

Redundant constraint A constraint is redundant iff it does not affect which configurations are valid in a non-void feature model [BSRC10, KAT16]. This means that the restriction it expresses is actually already expressed somewhere else in the feature model, thereby rendering the constraint superfluous.

Void feature model A feature model is void iff it does not contain any valid configurations [BSRC10, KAT16, SKT+16].



(a) Dead feature (b) False-optional feature (c) Redundant constraint (d) Void feature model

Figure 2.4: Minimal examples of feature model defects.

Simplistic examples of these defects can be found in Figure 2.4. In all four cases, the sole constraint causes the defect. In the first case, the feature A is dead because it is always deselected by the constraint. In the next case, the feature A is false-optional because it is optional but always selected by the constraint. In the third case, the constraint is redundant because it only selects the root, which is already always selected by definition. In the final case, the feature model is void because the constraint deselects the root, so no configurations are possible.

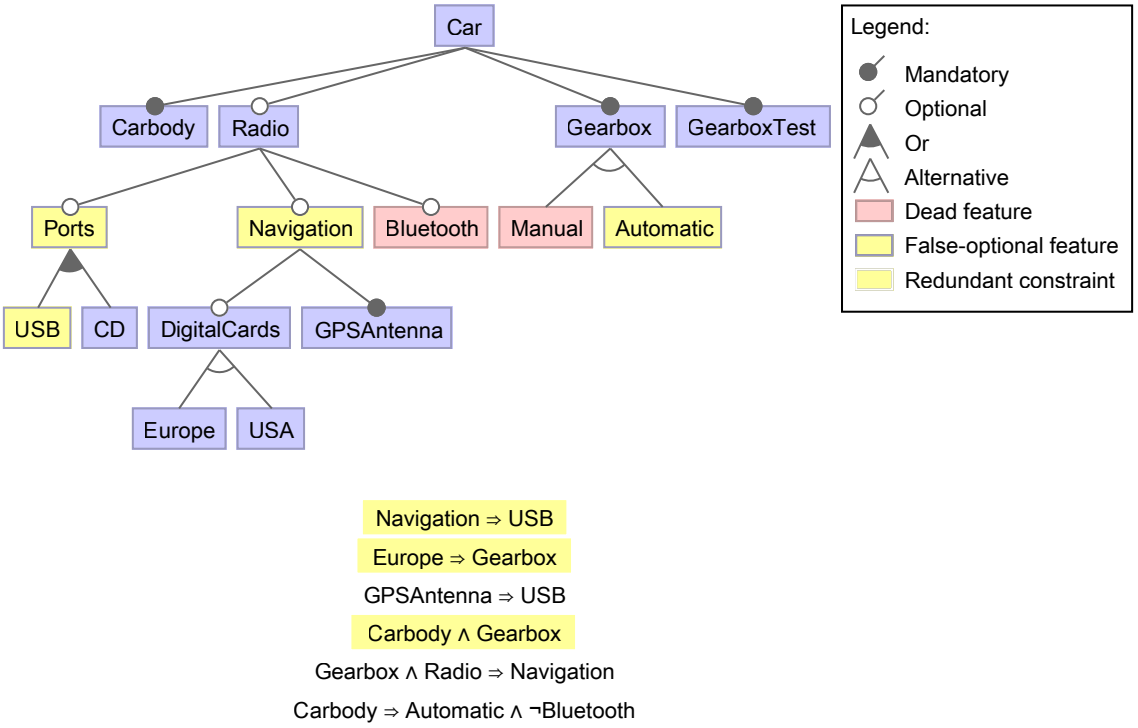


Figure 2.5: A feature model for a car containing several defects [KAT16].

An example of a feature model that more closely resembles one that might actually be found in the wild is depicted in Figure 2.5. The feature model contains features for a car such as a manual versus an automatic gearbox, an optional radio with optional support for Bluetooth, and more. However, the features *Manual* and *Bluetooth* can never be selected because of the last constraint and the alternative group *Manual* is in, making them dead features. *Navigation* is false-optional because of the second from last constraint, which always implies the selection of *Navigation* if its parent

Radio is selected. The third from last constraint, which forces the selection of *Carbody* and *Gearbox*, is redundant because those two features are already always selected due to being mandatory children of the root feature *Car*.

The cause for the remaining defects, i.e., the features *Ports*, *USB*, and *Automatic* being false-optional and the redundancy of the first two constraints, is left as an exercise for the reader. This is meant to show that it can be time-consuming to understand and therefore difficult to fix defects even in such small feature models, which is why the automated analysis of the feature model is an important tool in keeping the feature model free from defects.

Indeed, these defects can be formulated as the following satisfiability queries [Ana16], where SAT is the satisfiability function and fm the feature model formula:

Dead feature $\neg SAT(fm \wedge f)$, where f is the potentially dead feature.

False-optional feature $\neg SAT(\neg(fm \wedge (p \Rightarrow f)))$, where f is the potentially false-optional feature.

Redundant constraint $\neg SAT(\neg(fm \setminus c \Rightarrow c))$, where c is the potentially redundant constraint.

Void feature model $\neg SAT(fm)$.

These satisfiability queries can be used to automatically detect the defects. If they evaluate to true (i.e., the enclosed formula is unsatisfiable), an instance of the respective defect type is found.

2.3.2 Configuration Analysis

Arguably the most important information to know about a given configuration is whether it is valid. A configuration is valid if and only if the respective product is part of the software product line [TAK+14]. In other words, the validity of a configuration decides whether the configuration can be used to identify a product to be built during product derivation.

Fortunately, this can be checked fairly easily using the feature model's formula [Jan08]. To decide the validity of the configuration, it is assumed that each feature evaluates to true if and only if the feature is selected, i.e., part of the configuration. Then, the result of the evaluation of the feature model formula is the same as the validity status of the configuration.

However, only checking whether the configuration is invalid once it is done is not the only useful analysis over configurations. After all, the process of creating a configuration is incremental: Each step towards the full configuration produces a partial configuration of selected, deselected, and undefined features that can be analyzed before taking the next step [TAK+14].

Figure 2.6 shows an example of a partial configuration of the car feature model from Figure 2.5. In this case, all but the features *CD*, *DigitalCards*, *Europe*,

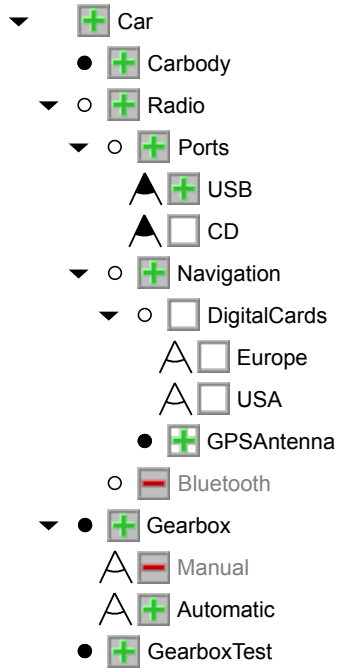


Figure 2.6: An exemplary partial configuration of a car.

and USA are either selected or unselected. These four features have an undefined selection status and can still be selected or deselected later. However, each step has the potential of failure by selecting an additional feature that results in an invalid configuration. For instance, the user could decide to select both *Europe* and USA even though they are alternatives to each other.

The underlying, familiar cause of the issue is the complexity of the feature model. When making a choice to select or deselect a feature, it is difficult to foresee and keep track of all the side effects [BCH15]. While the effect of selecting a feature is straightforward for its own structure, e.g., the alternative relationship it is in, the selection might just logically imply other selections all over different parts of the feature model once constraints come into play.

Luckily, the incremental and interactive nature of the configuration process leaves the option of providing automated guidance throughout. To this end, reasoning over partial configurations is necessary. A partial configuration is considered satisfiable if and only if there is a valid and full configuration the partial configuration is a subset of [KTS+17]. If so, more features can still be added to the partial configuration to eventually reach a configuration that is both valid and full. The satisfiability of a partial configuration can be decided automatically by deciding whether the feature model's formula is satisfiable under the assumptions of the partial configuration. Thus, contradictions can be found during the configuration process and not just at the end, when the full configuration is done.

Then, the idea behind the configuration process is to go from one satisfiable partial configuration to another until a full configuration is found that is valid necessarily. This can only go wrong by selecting or deselecting a feature that leads to an unsatisfiable configuration. For instance, once any one child of an alternative group is

selected, none of the others can be selected without making the configuration unsatisfiable. Presenting this information to the user allows such choices to be avoided.

This can be done by using decision propagation to automatically deselect such features [HSJ+04, KTS+17]. This means that, if selecting a feature would make the configuration unsatisfiable, it is deselected automatically to maintain satisfiability. Likewise, features that have to be selected for satisfiability, e.g., parent features, are selected automatically as well. In the example from Figure 2.6, the only manually selected feature is $GPSAntenna$, whereas all of the other selected or unselected features are only so because of automatic decision propagation. Having these selections made automatically is less time-consuming than having to manually figure out which selections are legal since it prevents backtracking [HSJ+04].

The satisfiability queries that can be used to detect features that need to be selected or deselected automatically given a satisfiable partial configuration are the following, where S is the set of selected features, U the set of unselected features, and $f \notin S \cup U$ is the feature to be checked:

Automatically selected feature $\neg SAT(fm \wedge (\bigwedge_{s \in S} s) \wedge (\bigwedge_{u \in U} \neg u) \wedge \neg f)$

Automatically unselected feature $\neg SAT(fm \wedge (\bigwedge_{s \in S} s) \wedge (\bigwedge_{u \in U} \neg u) \wedge f)$

Still, even with automatic decision propagation in place, there are often decisions to be made when finalizing partial configurations. For example, if the parent of an alternative group is selected, it becomes necessary to select one of the children as well. At this point, the choice of the desired child is best left to the user instead of having the algorithm decide on an arbitrary child. Alternatively, the user is free to deselect the parent again. In both cases, selection and deselection, a previously open clause is satisfied. Thus, by keeping track of the open clauses, the user choices can be guided towards a full configuration one clause at a time [PKM+16].

2.3.3 Family-Based Static Analysis of Preprocessor Directives

The final analysis discussed in this thesis is family-based static analysis [TAK+14]. As the name suggests, a family-based static analysis is a form of static analysis. A static analysis is one that can be done without running the code. Examples include checking whether a called method exists and whether its parameters are of the valid types. However, the task of such an analysis gets more complicated when software product lines come into play [LvRK+13]. If the static analysis also takes the variability of the entire software product line into account, it is called a family-based static analysis.

Such an analysis can for instance be useful if the software product line is implemented using preprocessors. Indeed, many family-based static analyses focus on annotations [TAK+14]. A notable example is the detection of preprocessor directives that do not contribute to the variability of the software product line implementation [TLSSP11]. In particular, the presence condition of a preprocessor directive might

be a contradiction, in which case the corresponding code block is never included in any product and the time spent writing the code possibly wasted. Likewise, the presence condition might be a tautology, in which case the annotation is redundant and merely clutters the code. In both cases, the preprocessor directive in question probably does not achieve what the developer had in mind.

```
1  public class Car {
2      public static void main(String[] args) {
3          System.out.print("Hello, car");
4          // #if Gearbox
5          System.out.print(" with");
6          // #if Manual
7  //@     System.out.print(" a manual");
8          // #elif Automatic
9          System.out.print(" an automatic");
10         // #if Manual
11 //@     System.out.print(" and impossible");
12         // #endif
13         // #endif
14         System.out.print(" gearbox");
15         // #endif
16         // #if Bluetooth
17 //@     System.out.print(" and Bluetooth");
18         // #endif
19         System.out.println("!");
20     }
21 }
```

Listing 2.2: Java code containing invariant presence conditions in Antenna preprocessor directives for the car feature model

Listing 2.2 shows a simple program with Antenna preprocessor directives referring to the car feature model from Figure 2.5. The preprocessor directives of the preprocessor directives contain contradictions and tautologies. Indeed, all of them are invariant and each evaluate to the same value regardless of the configuration. In particular, *Gearbox* and *Automatic* are core features that always need to be selected, so their code blocks do not have to be annotated with preprocessor directives in the first place. The features *Manual* and *Bluetooth*, by contrast, are dead, leaving their code blocks dead as well. As a result, this code will always print "Hello, car with an automatic gearbox!" no matter which (valid) feature configuration is used to configure the preprocessor. Realizing this requires knowledge of the feature model and the combinations of the many features and expressions used throughout the preprocessor directives, all of which can easily get too complex for a developer to always just keep in mind. This is especially true because not all preprocessor directives might be visible at the same time, yet they are relevant to the expressions nested in them. Thus, the detection of invariant presence conditions is highly useful for a correct domain implementation using preprocessors.

Fortunately, contradictions and tautologies can be detected using the following satisfiability queries, where e is the expression of the annotation to be checked and ne is the conjunction of the expressions in which e is nested in (each of which is negated if its branch is excluded due to an else-statement):

Dead code block $\neg SAT(fm \wedge ne \wedge e)$.

Superfluous annotation $\neg SAT(fm \wedge ne \wedge \neg e)$.

If either of these satisfiability queries evaluate to true, the expression e is invariant. It should be noted that the detection of a dead code block should take precedence over the detection of superfluous annotations. In other words, speaking of a superfluous annotation only makes sense if it does not also cause a dead code block. This is because the annotation containing e is not really the root of the problem if $fm \wedge ne$ is already a contradiction.

2.4 Summary

In summary, the Boolean satisfiability problem can be used to express important issues in a software product line. This chapter discussed the three use cases potentially containing such issues: feature models, which might be void or might otherwise contain dead features, false-optional features, and redundant constraints, configurations, which might require features to be selected or deselected automatically throughout their creation process, and finally code containing preprocessor annotations that might cause dead code blocks or might be superfluous.

The satisfiability queries used to express these issues can be decided using SAT solvers. Still, a Boolean "yes" or "no" is often not enough to understand what is causing the issue. Therefore, the next chapter discusses how to find explanations for these satisfiability queries.

3. Explaining Satisfiability Queries

The various issues that can occur throughout the software product line engineering process established in the previous chapter can take considerable amounts of time and effort to solve. Thus, it would be useful if the developer had explanations providing further insight into the causes of these issues. By explaining why something is problematic rather than just pointing out that it is, the developer can more easily identify the causes of the issue that need to be undone to solve the problem. Ultimately, this makes software product line engineering more cost-effective.

Out of this motivation, this chapter discusses how to tackle the task of finding explanations in a software product line engineering context. To this end, the explanation approach presented in this thesis is discussed from a conceptual perspective, which lays the foundations for implementing the approach to actually be able to use it. More importantly, the advantages and disadvantages of the various design decisions are highlighted to provide the understanding required for future work related to this thesis.

First, the definition of explanations and approaches for finding them are discussed in Section 3.1. In Section 3.1.1, special attention is paid to an explanation approach based on Boolean constraint propagation that serves as a theoretical foundation for this thesis. Being of particular prominence to this thesis, this includes a more detailed analysis of its most significant shortcoming: its incompleteness. This shortcoming is remedied with another approach presented in Section 3.1.2. Finally, the application of the latter approach to concrete use cases in software product lines is outlined in Section 3.2 and its subsections: explanations for feature model defects in Section 3.2.1, for configurations in Section 3.2.2, and for code with preprocessor directives in Section 3.2.3.

3.1 Finding Explanations

The subsections of Section 2.3 describe several circumstances that can be detected automatically using a satisfiability query. For instance, deciding whether the feature

model is void is equivalent to checking whether the Boolean formula of the feature model is unsatisfiable [KAT16].

However, as the complexity of the model increases, simply being told *that* something is the case is often not enough to understand *why* it is the case. Rather, an accurate understanding of the situation required to solve the underlying problem in turn also requires an explanation. For the purpose of this thesis, an explanation is an intuitive description of the cause of a circumstance. In other words, the goal of an explanation is to help understand why something is the case.

In the approach presented in this thesis, detecting the circumstance is separate from explaining it. This means that the algorithm for finding an explanation is only run once the circumstance to explain has been identified. The alternative would be to do it all in one go to leverage the internal state of the SAT solver for a performance boost in finding the explanations.

While reusing the state might make sense when only a single circumstance is detected, doing so becomes problematic when multiple circumstances are detected. In such such a case, there are two ways to reuse the state: Either the state for each circumstance is remembered for the explanation algorithm to come or the state is reused by the explanation algorithm immediately before the state changes when the next circumstance is detected. The former approach is not a reasonable option since it causes bad spatial performance. The latter approach makes it necessary to always find an explanation even if it ends up not being used. When explaining many or even every single defect of the model, reusing the state enables optimizations that probably make finding explanations faster. By contrast, separating the tasks of detecting circumstances and explaining them saves time and memory when many circumstances are detected, yet only a few of them need to be explained. This scenario, i.e., finding a single explanation, changing the model, and running the analysis again, seems like the more realistic use case for explanations. After all, there is no point in presenting all explanations since nobody would want to understand every single defect before fixing any of them. Once a defect is fixed and the model changes, the analysis for detecting the defects and therefore the explanation algorithm need to run from scratch anyway. This speaks in favor of separating the two tasks.

Additionally, while the main argument for merging the two tasks is the performance of finding an explanation in isolation of other circumstances, separating them also allows for other types of optimization. For example, the separation makes it possible to use one SAT solver to detect the circumstance and another for finding the explanations, where each SAT solver is optimized for its specific task. It is even possible to optimize the satisfiability query itself.

Another aspect is the separation of concerns. By separating the two tasks, the algorithm for finding the explanations does not require any knowledge of the algorithm for detecting the circumstance. The advantage is thus the ability to easily extend applications to provide explanations without touching the existing detection algorithm.

In short, the explanation workflow starts after the circumstance to explain is detected. The detection algorithm decides one of the Boolean formulas described in

the subsections of Section 2.3. The result of the detection algorithm presents two cases: Either the formula is satisfiable or it is unsatisfiable.

If the formula is satisfiable, an extra explanation is not even necessary. After all, the satisfiability is already proven by any satisfying interpretation found by the detection algorithm. For example, when deciding whether a feature model is void, any valid configuration shows that it is not. Indeed, all that can be said about why the formula is satisfiable is that it contains no clauses that make it unsatisfiable, which is trivial given the knowledge that the formula is satisfiable. This point deserves emphasis because it entails that, in order to allow meaningful explanations, all the circumstances requiring an explanation are best expressed as formulas that are unsatisfiable rather than satisfiable when the circumstance of interest (such as the existence of a dead feature) occurs.

Thus, the interesting case is the formula being unsatisfiable. Whereas satisfiability means absence of contradicting clauses, unsatisfiability means existence of contradicting clauses. Therefore, instead of just stating *that* there are contradicting clauses, it is possible to also point out *which* clauses are contradicting in an unsatisfiable formula. That means that the task of finding an explanation can be reduced to explaining the unsatisfiability of the formula by finding the clauses that make it unsatisfiable.

Of course, these clauses to look for are just a minimal unsatisfiable subset, which is why minimal unsatisfiable subsets lend themselves well for explanations [GMP08a]. Conveniently, considering the main task of finding an explanation is finding a minimal unsatisfiable subset, a lot of the work can be outsourced to a minimal unsatisfiable subset extractor.

The minimal requirements to the minimal unsatisfiable subset extractor used in finding these subsets for the explanation are soundness and completeness [FdK93]. Soundness denotes that the algorithm never returns incorrect results. In the case of explanations, soundness implies that the found explanation actually explains the circumstance. However, while soundness ensures the correct content of the result, it does not say anything about the result being found in the first place. Completeness means that the algorithm returns all correct results for all input, though it could also return an incorrect result for any specific input. Thus, if the algorithm is both sound and complete, it always returns the correct result if it exists, which is expected from the explanation algorithm presented in this thesis.

The next two subsections explain two approaches for finding these subsets for explanations. The first one in Section 3.1.1 is sound but incomplete. Therefore, it is superseded by the sound and complete approach presented in Section 3.1.2.

3.1.1 Explaining Using Boolean Constraint Propagation

A previous approach for finding explanations involves Boolean constraint propagation [Ana16, KAT16]. Once the detection algorithm has decided that the Boolean formula describing the circumstance is unsatisfiable, Boolean constraint propagation is applied as outlined in Section 2.1.1. Here, Boolean constraint propagation is not used to find a satisfying interpretation since the formula is known to be unsatisfiable

anyway. Instead, it is used to find a contradiction. More importantly, while it is doing so, its reasoning is recorded. Along the way to finding the contradiction, the unit-open clauses that are propagated make up a minimal unsatisfiable subset. By returning these clauses, an explanation is found.

However, while Boolean constraint propagation is fast and sound, it is unfortunately also incomplete [Ana16, FdK93]. This means that, even though Boolean constraint propagation will never incorrectly signal a contradiction when there is none, sometimes it might not signal a contradiction when there is one. Therefore, sometimes this approach fails to find an explanation for a defect.

More specifically, Forbus and de Kleer [FdK93] identify two classes of formulas for which Boolean constraint propagation does not give an answer. The first is defined by refutation incompleteness, which occurs when a contradiction that is implied logically is not found by Boolean constraint propagation. An example of this is the following formula:

$$(A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee B) \wedge (\neg A \vee \neg B) \tag{3.1}$$

This is like requiring every possible truth value assignment at the same time, which is unsatisfiable:

$$\begin{aligned} & (A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee B) \wedge (\neg A \vee \neg B) \\ \equiv\ & (A \wedge (B \vee \neg B)) \wedge (\neg A \wedge (B \vee \neg B)) \\ \equiv\ & (A \wedge \top) \wedge (\neg A \wedge \top) \\ \equiv\ & A \wedge \neg A \\ \equiv\ & \bot \end{aligned}$$

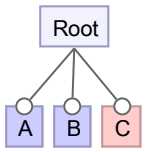
However, Boolean constraint propagation cannot find the contradiction in Equation 3.1 as not a single clause is unit-open. It would have to look at multiple clauses at the same time to deduce the conflict.

Indeed, as can be seen in Figure 3.1, it is possible to construct feature models with defects that cannot be explained using only Boolean constraint propagation because they contain Equation 3.1. In all four examples, the sole constraint is $\alpha \Rightarrow f$, where f is Equation 3.1, which makes α false and causes the defect.

Granted, with the contradiction obviously contained in the constraint, these examples are rather constructed. In contrast, Figure 3.2 is a void feature model that contains Equation 3.1 in a slightly more opaque manner. Here, the unexplainable contradiction only reveals itself once the entire feature model is written as a formula in conjunctive normal form:

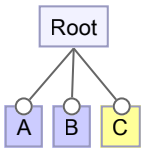
$$\mathit{Root} \wedge (\neg A \vee \mathit{Root}) \wedge (\neg B \vee A) \wedge (\neg A \vee B) \wedge (A \vee B) \wedge (\neg A \vee \neg B)$$

Finally, the other class of incompleteness is literal incompleteness [FdK93]. It occurs when it is possible to deduce the truth value assignment of a variable logically but



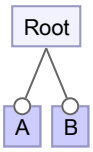
$C \Rightarrow (A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee B) \wedge (\neg A \vee \neg B)$

(a) Dead feature



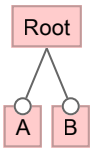
$\neg C \Rightarrow (A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee B) \wedge (\neg A \vee \neg B)$

(b) False-optional feature



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

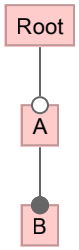
(c) Redundant constraint



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

(d) Void feature model

Figure 3.1: Feature models with defects showcasing refutation incompleteness.



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

Figure 3.2: A void feature model showcasing refutation incompleteness.

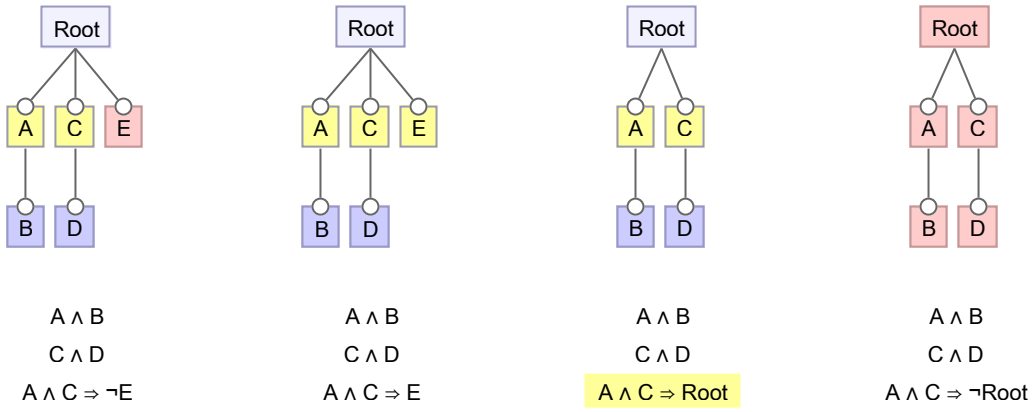
not by using Boolean constraint propagation. The following formula is an example of that:

$$(A \vee B) \wedge (A \vee \neg B) \tag{3.2}$$

These two clauses require the variable A to be true:

$$\begin{aligned} & (A \vee B) \wedge (A \vee \neg B) \\ \equiv\ & A \vee (B \wedge \neg B) \\ \equiv\ & A \vee \bot \\ \equiv\ & A \end{aligned}$$

Once more, Boolean constraint propagation fails due to a lack of unit-open clauses.



(a) Dead feature (b) False-optional feature (c) Redundant constraint (d) Void feature model

Figure 3.3: Feature models with defects showcasing literal incompleteness.

Examples of feature models with defects that cannot be explained using only Boolean constraint propagation due to literal incompleteness are shown in Figure 3.3. These examples are a bit trickier than the examples for refutation incompleteness as an analogous constraint would end up being unit-open. Instead, Equation 3.2 is applied twice, once to make A true and once more to make C true. Then, in each example, the constraint of the form $A \wedge C \Rightarrow \alpha$ renders α true, which causes the defect. None of these examples can be explained using only Boolean constraint propagation.

3.1.2 Explaining Using Minimal Unsatisfiable Subset Extractors

Even though Boolean constraint propagation is incomplete, there is a way to extend it to obtain a complete explanation approach. As mentioned in Section 2.1.1, adding backtracking to Boolean constraint propagation allows the approach to function as a complete SAT solver of the Davis-Putnam-Logemann-Loveland kind [DLL62, DP60,

NOT06]. This way, it can be used to extract the minimal unsatisfiable subsets of any Boolean formula for the desired explanation [BLMS12].

Indeed, considering that the Davis-Putnam-Logemann-Loveland procedure is just one of many ways to implement a SAT solver, it makes sense to generalize further. Instead of relying on this one specific SAT solving approach, the SAT solver is treated as a black box. This means that, for this explanation approach, it does not matter how the SAT solver provides minimal unsatisfiable subsets. Which SAT solver to use is therefore a question to answer during the implementation of the explanation approach, though out of performance concerns, it should support incremental satisfiability queries [ES03, ALS13].

Treating the SAT solver as a black box has several advantages. First, by not extending existing SAT solving approaches, the explanation approach is made much simpler because no knowledge of any specific SAT solving technique is necessary. The simplicity in turn makes it easier to implement and maintain. Finally, by keeping the SAT solver exchangeable, it can profit from past, present, and future advances in SAT solving technology. As such, this is the explanation approach used in this thesis.

3.2 Explanations for Software Product Lines

To recapitulate, in the approach presented in this thesis, explanations are found by using a SAT solver to find a minimal unsatisfiable subset of an unsatisfiable formula describing the circumstance of interest from Section 2.3. By describing the elements that cause the unsatisfiability, the minimal unsatisfiable subsets assume the role of the "description" part of the definition of explanations from the beginning of this chapter.

At this point, the minimal unsatisfiable subset could already be presented to the user. While it does capture the essence of the unsatisfiability of the formula, a number of clauses of a Boolean formula with no context hardly pass the "intuitive" requirement of the definition. The issue is that there are two level of abstractions: the model the user is working with on the one hand and its representation as a Boolean formula on the other. When the user asks for an explanation for a circumstance, a reasonable expectation is that the explanation uses the same level of abstraction as the user, which is the model being worked on. This assumption is violated when presenting a Boolean formula.

To overcome this difference and allow for a meaningful interpretation of the clauses, the clauses need to be transformed back to the abstraction level of the model. Unfortunately, for a complex model type, deducing the connection between the two representations might not always be easy. Indeed, the connection between a Boolean formula and the feature model it was found from is ambiguous [Ana16, CW07]. After all, the semantics of the model is lost during transformation to a Boolean formula for automated analysis. As such, the information of what each clause means in terms of the model must not be discarded during transformation. Instead, it must be possible to recall the reason for the inclusion of each clause when presenting the explanation to the user.

While the question of how to do this is left to the implementation, the information that needs to be remembered, that is the meaning of each clause, is already presented in the following subsections. In all of the following use cases, the circumstance is formulated as a Boolean formula to be explained with a minimal unsatisfiable subset plus the meaning of each clause.

3.2.1 Explanations for Feature Model Defects

When explaining a feature model defect, the first thing to do is to formulate the defect as a satisfiability query like in Section 2.3.1, which invariably involves transforming the feature model to a Boolean formula. The resulting formula in conjunctive normal form is a conjunction of clauses. An explanation is later found by extracting a minimal unsatisfiable subset from the formula containing the feature model formula. However, while the clauses of the minimal unsatisfiable subset do make up the explanation, when presenting the explanation to the user, not only the clauses need to be shown but their meaning as well.

Clause	Meaning
r	r is the root.
$\neg f \vee p$	f is a child of p.
$\neg p \vee f$	f is a mandatory child of p.
$\neg p \vee f_1 \vee \cdots \vee f_n$	$f_1, \ldots,$ and f_n are or-children of p.
	$f_1, \ldots,$ and f_n are alternative children of p.
$\neg f_1 \vee \neg f_2$	f_1 and f_2 are alternatives.
$c_{i,j}$	c_i is a constraint.

Table 3.1: Meaning of each clause of a feature model.

The meaning of each clause of the feature model formula is listed in Table 3.1, where $c_{i,j}$ is the j-th clause of the i-th constraint in conjunctive normal form and the remaining variables are as defined in Section 2.2.2. It should be noted that a clause can be more specific than a feature structure as a whole. For example, an or-group results in not only each child implying the parent, but also the parent implying at least one child. The direction of this relationship should be considered in the explanation to express as many details of the reasoning denoted by the minimal unsatisfiable subset as possible. Constraints are a special case in that they may end up being split into multiple clauses when transformed into conjunctive normal form, yet any one of the clauses is interpreted to stand for the respective constraint as a whole because subformulas of constraints cannot be referenced in a feature model.

To get the meaning of each clause, one might hope to be able to simply deduce the meaning of each clause from the clause alone. Unfortunately, this is not possible. For instance, the clause stemming from an or-relationship is the same as the one from an alternative relationship; the only difference is the inclusion of further alternative clauses, which may or may not in truth be clauses of a constraint instead of an alternative relationship. Indeed, a clause of a constraint may look the same as any other regular clause created for a structure in the feature model. Thus, the implementation of this explanation approach requires a way to store the meaning of each clause as they are being created from the feature model (see Section 4.3).

In any case, once the defect is formulated as a Boolean formula that is confirmed to be unsatisfiable, the explanation algorithm begins. It finds the minimal unsatisfiable subset of the formula using a SAT solver. Which formula it actually uses depends on the defect. In general, the formula is the respective one of those listed at the end of Section 2.3.1. Those are the same formulas already used in Ananieva's approach based on Boolean constraint propagation [Ana16]. An exception to this is the formula for redundant constraints, for which Ananieva combines the explanations for multiple formulas, specifically one formula for each satisfying assignment of the constraint in and of itself to avoid negating the redundant constraint. However, this requires checking multiple formulas, so in this approach, the formula is simplified into a single one with the same satisfiability.

Table 3.2 lists the expected explanations for the defects in Figure 2.5 already explained informally in Section 2.3.1. In all cases, the explanation consists of the clauses of the minimal unsatisfiable subset as well as the meaning of each clause. Being explanations, these example results should be self-explanatory. However, for the sake of simplicity, explanations do not include assumptions made such as that the parent feature is selected when talking about a false-optional feature.

3.2.2 Explanations for Configurations

The process of finding explanations for configurations is analogous to the one for defects in feature models. First, the circumstance to explain is expressed as a satisfiability query which is in conjunctive normal form and unsatisfiable. However, instead of only considering the feature model, clauses representing selected and unselected features from the (full or partial) configuration are included, too.

Throughout the configuration process, each feature may be selected or deselected manually by the user. These manual configuration choices require no explanation as they do not involve any reasoning unknown to the user. Still, they might cause other features to be selected or deselected automatically to maintain satisfiability as discussed in Section 2.3.2. These automatic selections do involve automated reasoning over both the manual decisions and the feature model. Because the reasoning behind these automatic decisions might not always be obvious, it might help to have them explained.

For explaining why a feature is automatically selected or automatically unselected, the question becomes why it would not be possible to make a different decision regarding the selection of the feature than has already be done automatically. After all, if the opposite decision leads to an invalid configuration, the automatically made decision is the only valid choice left. This is in turn the same as checking the satisfiability query from Section 2.3.2.

However, only manually selected or manually unselected features need to be included as part of the configuration because automatic ones (except for the one being currently explained) can be deduced anyway. In fact, the automatically selected or automatically selected features must not be included because, otherwise, the explanations for automatic decisions could lead to circular reasoning between each other.

Clause	Meaning
Car	Car is the root.
$\neg Car \vee Carbody$	$Carbody$ is a mandatory child of Car.
$\neg Carbody \vee \neg Bluetooth$	$Carbody \Rightarrow Automatic \wedge \neg Bluetooth$ is a constraint.

(a) Explanation why *Bluetooth* is a dead feature.

Clause	Meaning
Car	Car is the root.
$\neg Car \vee Carbody$	$Carbody$ is a mandatory child of Car.
$\neg Carbody \vee Automatic$	$Carbody \Rightarrow Automatic \wedge \neg Bluetooth$ is a constraint.
$\neg Manual \vee \neg Automatic$	$Manual$ and $Automatic$ are alternatives.

(b) Explanation why $Manual$ is a dead feature.

Clause	Meaning
Car	Car is the root.
$\neg Car \vee Gearbox$	$Gearbox$ is a mandatory child of Car.
$\neg Gearbox \vee \neg Radio \vee Navigation$	$Gearbox \wedge Radio \Rightarrow Navigation$ is a constraint.

(c) Explanation why $Navigation$ is a false-optional feature.

Clause	Meaning
Car	Car is the root.
$\neg Car \vee Carbody$	$Carbody$ is a mandatory child of Car.
$\neg Car \vee Gearbox$	$Gearbox$ is a mandatory child of Car.

(d) Explanation why the third from last constraint is redundant.

Table 3.2: Explanations for feature model defects in Figure 2.5.

Then, a minimal unsatisfiable subset of the formula is found using any minimal unsatisfiable subset extractor, e.g., a SAT solver or Boolean constraint propagation. Each resulting clause stems from either the feature model with the meaning from Table 3.2 or the configuration with the meaning always being the selection state of the feature, i.e., "f is selected" for the clause f and "f is unselected" for the clause $\neg f$. For explanations of automatic selections, this always refers to the manual selection state.

Clause	Meaning
$\neg Navigation \vee Radio$	*Radio* is a child of *Navigation*.
$\neg GPSAntenna \vee Navigation$	*Navigation* is a child of *GPSAntenna*.
$GPSAntenna$	*GPSAntenna* is selected.

(a) Explanation why *Radio* is automatically selected.

Clause	Meaning
Car	*Car* is the root.
$\neg Car \vee Carbody$	*Carbody* is a mandatory child of *Car*.
$\neg Carbody \vee \neg Bluetooth$	$Carbody \Rightarrow Automatic \wedge \neg Bluetooth$ is a constraint.

(b) Explanation why *Bluetooth* is automatically unselected.

Clause	Meaning
$\neg GPSAntenna \vee Navigation$	*GPSAntenna* is a child of *Navigation*.
$GPSAntenna$	*GPSAntenna* is selected.

(c) Explanation why *Navigation* is automatically selected.

Table 3.3: Explanations for automatic configuration selections in Figure 2.6.

Table 3.3 shows various explanations for why certain features are automatically selected or automatically unselected in the car configuration from Figure 2.6. The explanation for why *Radio* is automatically selected (Table 3.3a) most notably states the selection of the feature *GPSAntenna* as a reason. The other two reasons refer to the feature model as known from Section 3.2.1.

Indeed, the entire explanation might reference only the feature model and never the configuration, as seen in the explanation for *Bluetooth* being automatically unselected (Table 3.3b). This is a special case of explaining why the dead feature *Bluetooth* cannot be selected. The explanation of its selection status is the same as the earlier explanation for why *Bluetooth* is dead (Table 3.2a). This is because the configuration cannot have a bearing on the selection status of a dead feature.

Still, in some cases, by taking the configuration into account, a configuration explanation might be different from the corresponding explanation for the feature model defect. This is seen in the explanation for *Navigation* (Table 3.3c), which is shorter because it directly references the selection status of its parent *GPSAntenna* instead of first having to deduce the truth value through various constraints and structures in the feature model (Table 3.2c).

On the one hand, referencing the configuration might make the explanation of dead and false-optional features shorter and more comprehensible. On the other hand, in these cases, the configuration is not really involved in the automatic selection of the feature, so it might be misleading to reference the configuration as if it did. Thus, implementations of this approach might choose to first check whether the feature is dead or false-optional instead of finding an explanation involving the configuration.

3.2.3 Explanations for Preprocessor Directives

The final use case for explanations considered in this thesis is that of code annotated with preprocessor directives. This use case is more complex because explanations not only reference the feature model but also the various preprocessor directives. The preprocessor directives might be nested, thus implying or excluding each other. They could be defect in and of themselves or in conjunction of the feature model. At the same time, explanations are particularly relevant here. For example, the user might not even be aware of all the many nested expressions due to them being displaced by the code they annotate. To the author's knowledge, this is the first time explanations are found for defects in preprocessor directives in a software product line context.

Specifically, this section revolves around finding explanations for the invariant expressions discussed in Section 2.3.3. As before, finding explanations in this context first requires an unsatisfiable satisfiability query from which a minimal unsatisfiable subset is extracted. The used satisfiability queries are those from that chapter. The resulting clauses reference either the feature model or the propositional expressions of the preprocessor directives used throughout the code.

Contradictions are checked before tautologies because a tautology nested inside a contradiction should still be considered a contradiction. After all, its corresponding code block will never be selected, which is the issue related to contradictions as opposed to tautologies. Once the issue is identified, a minimal unsatisfiable subset is extracted as usual.

```
1  // #if Radio
2  // #if !Radio
3  // #endif
4  // #endif
5
6  // #if Manual
7  // #else
8  // #endif
9
10 // #if GPSAntenna | Navigation
11 // #if !USB
12 // #endif
13 // #endif
```

Listing 3.1: Antenna preprocessor directives with invariant presence conditions for the car feature model

Listing 3.1 shows a number of Antenna preprocessor directives (without actual code blocks for the sake of simplicity), all of which except those in Line 1 and 10 have

Clause	Meaning
$Radio$	$Radio$ is a parent expression.

(a) Explanation why $\neg Radio$ in Line 2 is a contradiction.

Clause	Meaning
Car	Car is the root.
$\neg Car \vee Carbody$	$Carbody$ is a mandatory child of Car.
$\neg Carbody \vee Automatic$	$Carbody \Rightarrow Automatic \wedge \neg Bluetooth$ is a constraint.
$\neg Manual \vee \neg Automatic$	$Manual$ and $Automatic$ are alternatives.

(b) Explanation why $Manual$ in Line 6 is a contradiction and why its negation through the else-statement in the next line is a tautology.

Clause	Meaning
$\neg Navigation \vee USB$	$Navigation \Rightarrow USB$ is a constraint.
$\neg GPSAntenna \vee USB$	$GPSAntenna \Rightarrow USB$ is a constraint.
$GPSAntenna \vee Navigation$	$GPSAntenna \vee Navigation$ is a parent expression.

(c) Explanation why $\neg USB$ in Line 11 is a contradiction.

Table 3.4: Explanations for invariant presence conditions in Listing 3.1.

invariant presence conditions. They all reference the car feature model from Figure 2.5. The explanations for them can be found in Table 3.4.

The first contradiction in Line 2 is as simple as possible. In Line 1, *Radio* is required to be selected for the inclusion of the code block from Line 2 to 3. At the same time, *Radio* is required to be deselected in Line 2. As such, the inner expression is a contradiction. The explanation (Table 3.4a) simply refers to the outer expression it is nested in.

However, the feature model can also come into play when evaluating expressions for preprocessor directives. Line 6 requires the dead feature $Manual$ to be selected, but being dead, this is not possible. As such, the explanation (Table 3.4b) explains why the feature $Manual$ is dead. The same explanation can be used for why the else-statement in the next line is superfluous as the negation of a contradiction is a tautology.

Finally, the most interesting case is when both the feature model and other preprocessor directives come together and form an invariant presence condition. This is the case with the contradiction in Line 11. Its parent expression requires $GPSAntenna$ or $Navigation$ to be selected. However, as the explanation points out (Table 3.4c), the feature model contains constraints which enforce the selection of USB in either case. With USB thus always being unselected in this code block, $\neg USB$ becomes a contradiction.

3.3 Summary

To sum up, the previous approach for finding explanations using Boolean constraint propagation is incomplete. The approach presented in this thesis overcomes this issue by using any SAT solver as minimal unsatisfiable subset extractor. After the satisfiability query to explain is identified as unsatisfiable, a minimal unsatisfiable subset is extracted from its underlying formula. By remembering the meaning of each clause specific to the use case, a meaningful explanation can be presented to the user.

Whereas the previous approach only covers feature model defects, the approach presented in this thesis additionally handles configurations as well as code with preprocessor directives. More importantly, it can be applied to any use case in software product line engineering given the satisfiability query to explain and the meaning of each clause.

However, to make quantitative comparisons between this approach and the one based on Boolean constraint propagation, it first needs to be implemented. This is covered in the next chapter.

4. Implementation in FeatureIDE

The explanation finding algorithm designed in the previous chapter has been implemented in the course of this thesis in the software product line development tool FeatureIDE [MTS+17, TKB+14]. This chapter discusses the inner workings of the implementation as well as the reasoning behind the various design choices specific to the implementation. This chapter is mainly targeted at developers of FeatureIDE seeking to modify or extend the implementation presented here but would also be of use to developers wishing to apply this approach to other applications. In any case, an understanding of object-oriented programming is required.

Specifically, Section 4.1 serves as an introduction to FeatureIDE and why it is a useful foundation for the implementation. The core of the implementation, the data types of explanations and finding them using minimal unsatisfiable subset extractors, is discussed in Section 4.2. This includes the three use cases of feature model defects (Section 4.2.1), configurations (Section 4.2.2), and preprocessor directives (Section 4.2.3). For finding these explanations, a trace model and a solver are required, discussed in Section 4.3 and Section 4.4 respectively. Finally, Section 4.5 details how the explanations are displayed visually for the users of FeatureIDE.

It should be noted that some of these sections make use of class diagrams that do not model every single detail of the implementation but rather illustrate the classes and their connections for a more general overview. For example, even though at several points the implementation uses private class members with public accessors and mutators as is common in Java, the class diagram containing such a class lists it as a public class member instead. Moreover, even though often times subclasses override methods of the superclass with a different implementation or by specifying a more specific return type, methods redeclared in a subclass are not redeclared in the class diagram. Finally, if a class has been explained in another class diagram already but it is referenced by some other class in another class diagram, the members of the referenced class are omitted.

4.1 FeatureIDE

FeatureIDE [MTS+17, TKB+14] consists of a number of plug-ins for Eclipse, a widespread and easily extensible integrated development environment most Java developers are familiar with. Being an open-source project, the code of FeatureIDE and therefore the code of this implementation can be found online.[1] This implementation is part of FeatureIDE version 3.4.0.

FeatureIDE comes with several useful functionalities that the implementation builds upon and integrates into such as a feature model editor, a configuration editor [PKM+16], and a text editor that recognizes preprocessor directives referencing the feature model [MTS+16].

Since FeatureIDE is written in Java, the programming language of choice is Java. For one, many of Java's design goals [GM96] such as simplicity and portabiltiy overlap with those of the implementation. Moreover, Java offers a wide range of preexisting functionalities from not only its internal API but also from external frameworks and libraries.

In particular, FeatureIDE uses Eclipse's Graphical Editing Framework, which is a model-view-controller [GHJV95] framework for such model-based graphical editors. That is, FeatureIDE is split into three parts with different purposes: the model, which holds the data to be edited such as a feature model, the view, which determines how this data is displayed to the user, and finally the controller, which manages the interaction between the two by translating user input in the graphical editor to actions in the model and updating the view when the model changes. This separates the model from the view, therefore reducing the side-effects and thus effort of changing, maintaining, or even replacing parts of the application.

This is important because most of the effort of this implementation is geared towards the model. After all, finding explanations involves reasoning over models such as feature models, configurations, and code annotated with preprocessor directives. However, an explanation is useless without it being noticed by the user, so of course the implementation also includes additions to the view and the controller, discussed in detail in Section 4.5.

FeatureIDE also comes with its own library for propositional formulas, Prop4J, which is used throughout the implementation to store and transform formulas. In general, when a formula is used at any point in the implementation, this refers to an instance of Prop4J's class `Node`.

4.2 Generalizing Explanations

A major contribution of this implementation is the explanations architecture. It resides in FeatureIDE's central plug-in for feature models and other related models such as configurations, which is called `de.ovgu.featureide.fm.core`. Besides the definition and implementation of these models, this plug-in also provides auxiliary functionalities such as feature model analyses and format transformations. Being

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

a form of analysis that also involves feature models, the functionality for finding explanations naturally lives in this plug-in, specifically in the subpackage `explanations`. This is the same package in which Ananieva [Ana16] originally implemented FeatureIDE's first approach for finding explanations using Boolean constraint propagation.

Abstract Explanations

As the approach of Ananieva only considers the use case of feature model defects, their semantics were hardcoded into the data structures and algorithms. By contrast, the approach presented in this thesis is not limited to only feature model defects. Instead, all circumstances that can be formulated as a satisfiability query are considered. For that reason, the concept of explanations is generalized in this thesis. This means that the abstract concept of an explanation as defined at the beginning of Section 3.1 is separated from its concrete use cases such as feature model defects.

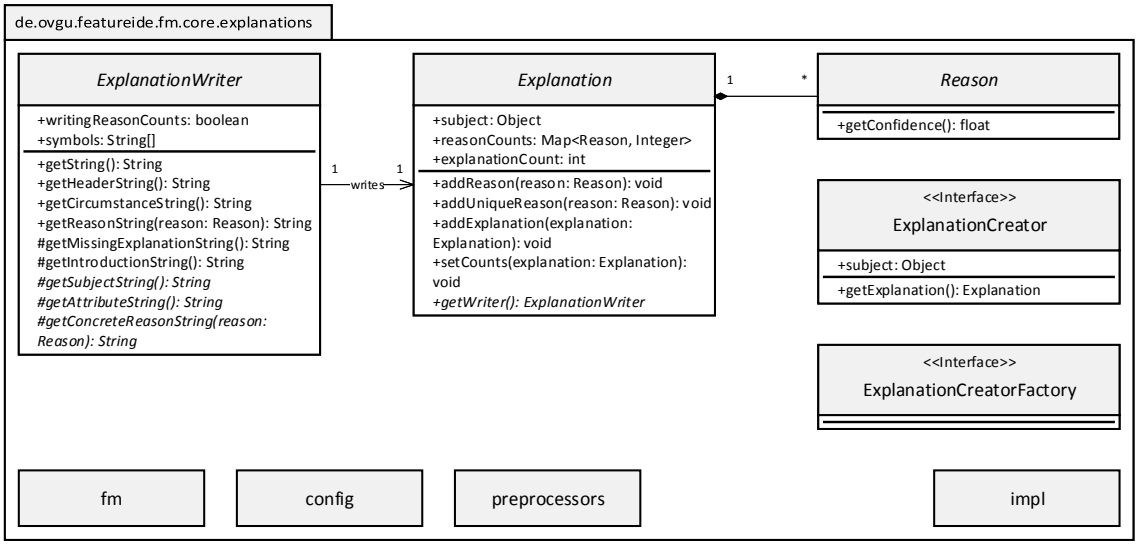


Figure 4.1: Class diagram for explanations.

The separation of abstract and concrete explanations is reflected in the implementation. Figure 4.1 shows the direct contents of the root level package for explanations, which holds all of the interfaces and abstract classes for abstract as opposed to concrete explanations. All of these types are separate of any use case and need to be extended to be filled with the semantics of a specific use case. Not only does this grant the usual benefits of polymorphism such as modularity, type safety, and the reduction of code duplication, but the explanation finding algorithm can be easily extended to support future use cases that are not considered in this thesis.

Concrete Explanations

The subpackages in the bottom left corner of Figure 4.1, i.e., `fm`, `config`, and `preprocessors`, handle these various concrete use cases and are detailed in Section 4.2.1, Section 4.2.2, and Section 4.2.3 respectively. They all reside in this package merely

due to the fact that they all share a usage of the feature model. In theory, these packages for concrete use cases could live anywhere, even an entirely different plug-in, and implement explanations for any conceivable circumstance. In other words, the semantics of these explanations are extensible.

Oracles

The semantics of the explanations are just one dimension of extensibility. In addition, the explanations are agnostic to the concrete algorithm used in finding them. This is where the `impl` package comes into play. It holds the implementations of algorithms that can be used to reason over circumstances in an abstract way, i.e., without being bound to the specific semantics of the use case such as feature model defects. This of course means that these algorithms can be used in finding the concrete explanations of the aforementioned packages. Henceforth, such algorithms are called "oracles", which, in the scope of this thesis, are either a minimal unsatisfiable subset extractor or Boolean constraint propagation.

By this description, one might be lead to believe that the minimal unsatisfiable subset extractors fundamental to the explanation finding approach presented in this thesis live inside this `impl` package. In truth, the minimal unsatisfiable subset extractors are abstracted even further and are hidden behind a solver facade living in another package as detailed in Section 4.4, which, if anything, just proves the extensibility of this architecture.

The only oracle that actually resides in the `impl` package is the logical truth maintenance system of the previous explanation approach based on Boolean constraint propagation [Ana16]. Of course, in order to be compatible with the generalized explanation finding structure, it had to be refactored considerably, part of which was already done as part of a project work [Gü16]. As a result, it can now be used to find explanations not only for feature model defects but for other use cases as well because it essentially just returns the minimal unsatisfiable subset of the formula it is given as input. Unfortunately, its incompleteness described in Section 3.1.1 remains an issue that is motivation enough to use a dedicated minimal unsatisfiable subset extractor in its place.

Data Types: `Explanation` and `Reason`

To finally go into the details of the architecture, the abstract class `Explanation` is the core data structure exposed to clients such as the feature model editor. To recall, an explanation is in essence a minimal unsatisfiable subset for which the semantics of the model it originates from is retained. The difference between a minimal unsatisfiable subset and an explanation is that the latter comes with additional semantics.

Part of the semantics is knowing what the explanation is even about in the first place. Every instance of `Explanation` revolves around a subject, which is an object that could be anything from a feature in a feature model to a selection in a configuration. However, the subject alone does not describe the circumstance to be explained, as for example an explanation for a feature could be about the feature being dead or alternatively about it being false-optional. As such, the subject also has an

attribute, e.g., dead or false-optional. The attribute is captured via polymorphism, meaning every type of circumstance to be explained comes with its own subclass of `Explanation`. Compared to the previous implementation of storing a value of an enumeration of all known explainable circumstance types, storing this information via polymorphism is more extensible.

To account for the part of the definition of explanations besides semantics, i.e., the aspect of them being minimal unsatisfiable subsets, instances of `Explanation` consist of any number of instances of `Reason`. A reason could theoretically be anything, though given the interpretation of minimal unsatisfiable subsets used throughout this thesis, it always corresponds to exactly one of the clauses of the minimal unsatisfiable subset the explanation represents. The set of all reasons that makes up the explanation thus capture the minimal unsatisfiable subset. A reason could for example refer to a relationship between features in a feature model or the selection state of a feature in a configuration. In any case, this information is once more captured using polymorphism, meaning there is a subclass of `Reason` for each type of reason, e.g., whether it stems from a feature model or a configuration. By chaining these reasons together, coherent reasoning emerges.

Explanations may also be merged while retaining the information of how often each reason was added in total. This is useful when multiple explanations for the same circumstance are found. This is the case for the previous approach based on Boolean constraint propagation, which finds multiple explanations and picks the shortest one. By remembering how often all of these other explanations were found as well as the reasons contained in these explanations, it is possible to deduce which reasons are part of every explanation and therefore likely a key reason in the explanation of the circumstance. In contrast, reasons that are not part of every explanation are less relevant as they play an exchangeable role in the causation of the circumstance. The numeric percentage of all the found explanations that include a given reason is thus called the reason's confidence.

Transformation to Natural Language: `ExplanationWriter`

Given all this information, instances of `Explanation` can be transformed into model space for the user to understand more easily. A straightforward target representation is natural language. The transformation to natural language is the job of the abstract class `ExplanationWriter` as handling it all in the `toString()` method of the class `Explanation` instead would get messy quickly. Each instance of `Explanation` knows which concrete subclass of `ExplanationWriter` in turn knows all the semantic details of properly transforming that instance of `Explanation`. For example, an instance of `ExplanationWriter` for an instance of `Explanation` explaining a dead feature would know that the subject can be written as the feature name and that the attribute can be written as "dead". In any case, the explanation in natural language is an introduction describing the circumstance to be explained, i.e., the subject and its attribute, followed by a series of reasons in natural language transformed similarly. It may also print additional data such as each reason's confidence. Moreover, the way any potentially occurring formulas are printed may be customized by providing a different set of symbols for the operators.

Finding Explanations: `ExplanationCreator`

Finding meaningful explanations is the task of the interface `ExplanationCreator`. Besides an accessor and a mutator for the subject of the explanation to be found, the only method it defines is `getExplanation()`. This might come across as somewhat surprisingly simple compared to the complexity of the other classes and the task at hand. Less surprisingly, though, this complexity is as always handled through polymorphism in the subclasses. Extending interfaces define additional members to be able to actually receive the crucial information such as the subject of the explanation to be found. The concrete subclasses of `ExplanationCreator` are the core of the explanation finding algorithm as they take the circumstance in model space, translate it to a satisfiability query, and construct a sensible explanation from it. As detailed in the following subsections, this is done in the combination of the two dimensions of firstly the concrete use case (feature models vs. configurations vs. preprocessors) and secondly the oracle used to find them (minimal unsatisfiable subsets extractor vs. Boolean constraint propagation).

Because there are multiple approaches for finding explanations, each use case additionally defines its own abstract factory [GHJV95] that implements the tagging interface `ExplanationCreatorFactory`. Each of these abstract factories is then subclassed by concrete factories that only return concrete instances of `ExplanationCreator` that use the same underlying algorithm. This way, switching out the approach for finding explanations for another one is as easy as using a different concrete factory.

The following subsections details how explanations are found for the three concrete use cases considered in this thesis: feature model defects in Section 4.2.1, configurations in Section 4.2.2, and preprocessor directives in Section 4.2.3.

4.2.1 Explanations for Feature Model Defects

The subpackage `fm` holds all of the code necessary for finding explanations that involve a feature model. This means two things. First and foremost, it contains the functionality necessary for finding explanations for feature model defects. Secondly, however, its classes may also be extended by classes of use cases for which the explanations involve a feature model, most notably configurations and preprocessor directives. While this distinction entails some finer implementation details, it mostly becomes relevant in the sections for these other use cases, and so this section is simply dedicated to the implementation of explanations of feature model defects.

Figure 4.2 shows the various classes of the subpackage `fm` in relation to their abstract superclasses detailed in the previous section. They all apply the semantics of the concrete use case of feature models to the abstract classes they extend. In other words, whereas the previously discussed abstract explanations do not have any meaning on their own, the concrete explanations discussed from this point on provide this meaning by referring to a tangible feature model.

Data Types: `FeatureModelExplanation` **and** `FeatureModelReason`

The abstract class `FeatureModelExplanation` represents explanations involving a feature model. As a subclass of `Explanation`, instances of `FeatureModelExplanation` may contain any number of instances of `Reason`. For explanations of feature

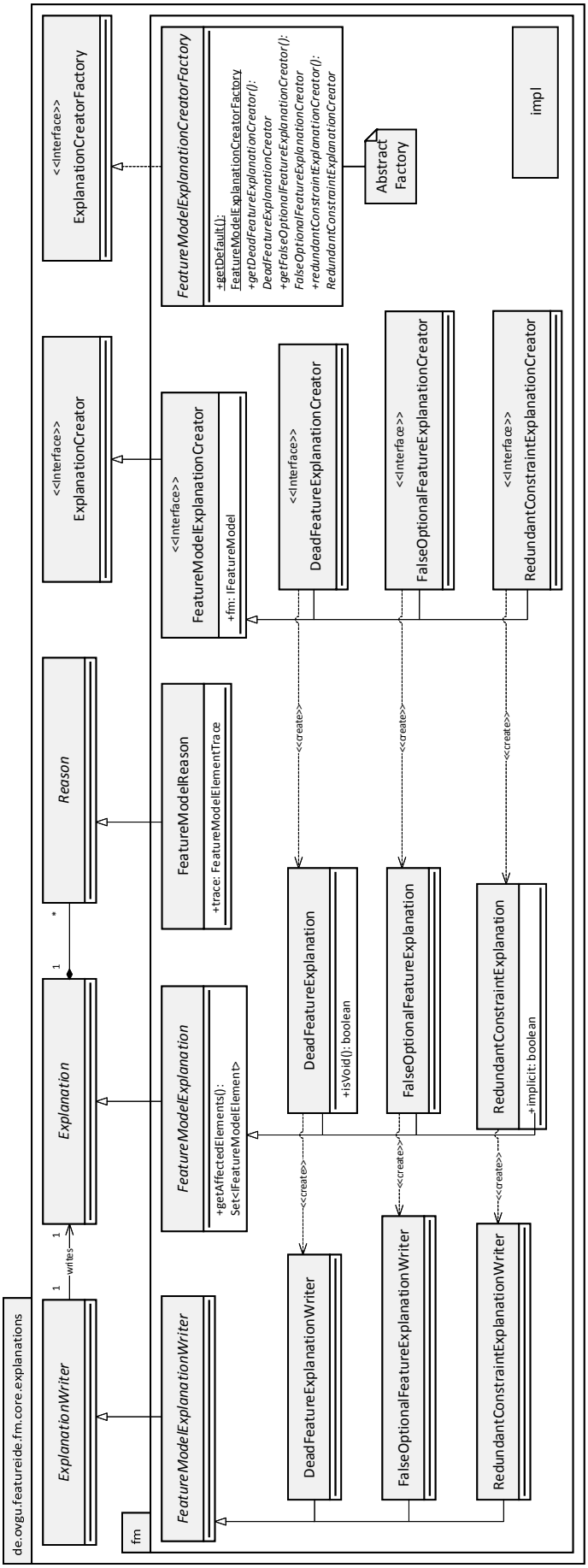


Figure 4.2: Class diagram for explanations for feature models.

model defects, those are in particular instances of `FeatureModelReason`. These reasons simply refer to a trace of the trace model described in Section 4.3. In essence, a trace maps a clause of the feature model as a formula in conjunctive normal form to the feature model element it originates from. That way, the semantic connection between the feature model and the clauses of the minimal unsatisfiable subset the explanation consists of is maintained. This is necessary for the explanation to be transformed back into the model space that the user is comfortable with.

The `FeatureModelExplanation` class also provides a helper function for figuring out which feature model elements are affected by the explanation. A feature model element is considered to be affected by the explanation if it is somehow involved in the explanation, i.e., it is the subject of the explanation or the subject of one of its reasons. This is useful for quickly working with them in the client such as the feature model editor, in which affected elements are highlighted (cf. Section 4.5.1).

Each concrete subclass of `FeatureModelExplanation` in this package is responsible for one of the specific types of feature model defects mentioned in Section 2.3.1. `DeadFeatureExplanation` explains dead features. It also explains void feature models if the root feature is dead as the underlying formula is the same as the one that would be used for void feature models if it were handled separately anyway (cf. Section 2.3.1). Next, `FalseOptionalFeatureExplanation` explains false-optional features. Finally, `RedundantConstraintExplanation` explains redundant constraints. It also explains transitive constraints [KAT16] if the constraint in question is implicitly defined [AKTS16] through feature model slicing [Kan16, KSTS16]. In all three cases, the subject accessors are overridden to return the specific type of subject instead of simply `Object`.

Transformation to Natural Language: `FeatureModelExplanationWriter`

The same hierarchical pattern can be found with the abstract class `FeatureModelExplanationWriter`. Its concrete subclasses each write instances of the concrete subclasses of `FeatureModelExplanation` in natural language. For example, `DeadFeatureModelExplanationWriter` implements the abstract `getSubjectString()` by describing the dead feature that is the subject of the explanation with its name or alternatively by returning "feature model" if the explanation is about a void feature model. Similarly, `getAttributeString()` either returns "dead" or "void". The rest of the implementation remains the same as in `FeatureModelExplanationWriter` and `ExplanationWriter`. Thus, this architecture results in as little code duplication as possible by only requiring the bare minimum of the knowledge specific to the circumstance type of each concrete class.

Finding Explanations: `FeatureModelExplanationCreator`

The only interfaces in this package are the ones for finding explanations of these three concrete defect types. The interface `FeatureModelExplanationCreator` extends `ExplanationCreator` with the feature model context and hence defines an accessor and a mutator for it. The three interfaces that in turn extend it, `DeadFeatureExplanationCreator`, `FalseOptionalFeatureExplanationCreator`, and `RedundantConstraintExplanationCreator`, handle the various defects by assuming a more specific subject type.

These three interfaces are the abstract products of the abstract factory [GHJV95] `FeatureModelExplanationCreatorFactory`. The concrete factory and the concrete products are defined in the subpackage `impl`. This strict separation of the interfaces and their implementation is to make the different approaches for finding explanations interchangeable.

The subpackage `impl` is visualized in Figure 4.3. It contains two further subpackages, `mus` and `ltms`, for the implementation of finding explanations of feature model defects using either a minimal unsatisfiable subset extractor or a logical truth maintenance system based on Boolean constraint propagation.

Finding Concrete Explanations: `AbstractFeatureModelExplanationCreator`

The only class that lives on the root level of the package `impl` is the abstract class `AbstractFeatureModelExplanationCreator`, which provides some core functionalities that are used by both approaches. In particular, it manages the various resources used in finding explanations. These are the feature model as a formula in conjunctive normal form and the trace model to retain the feature model semantics in the formula (cf. Section 4.3). To generate these two, it requires the feature model as well as an instance of `AdvancedNodeCreator`, which not only handles the transformation but also builds the trace model while doing so.

In order to avoid unnecessary computations, these resources are cached by the class `AbstractFeatureModelExplanationCreator`. In particular, the transformation of the feature model to a formula in conjunctive normal form is a rather costly operation and definitely needs to be cached. The trace model is cached in the class `AdvancedNodeCreator` already, but is cached by reference here as well for consistency and to remain agnostic of such implementation details of other classes that might change in the future. Finally, even the instance of `AdvancedNodeCreator` is cached.

To ensure correctness, these cached resources need to be kept up-to-date. Fortunately, the dependency graph of the resources is fairly simple. When the feature model is changed, all of the other resources need to be refreshed. However, to avoid unnecessarily generating all these resources whenever the feature model is changed, the class relies heavily on lazily generating resources when needed as opposed to greedily as soon as possible. This is achieved by marking the resource as dirty by setting it to `null` and recreating it in the accessor method when this state is encountered. That way, committing to costly operations such as generating the feature model formula is done as often as necessary and as rarely as possible.

Apart from managing resources, the `AbstractFeatureModelExplanationCreator` also provides some functionality for building an explanation from a minimal unsatisfiable subset. The method `getExplanation(Set<Integer>)` takes a set of clauses, i.e., the minimal unsatisfiable subset, where each clause is referenced by its index in the formula, and returns an explanation containing one reason for each clause. For feature model defects, these reasons are instances of `FeatureModelReason`, which just contain these traces taken from the trace model. Naturally, in order to do so, it needs to construct a new instance of some concrete subclass of `Explanation`. This is

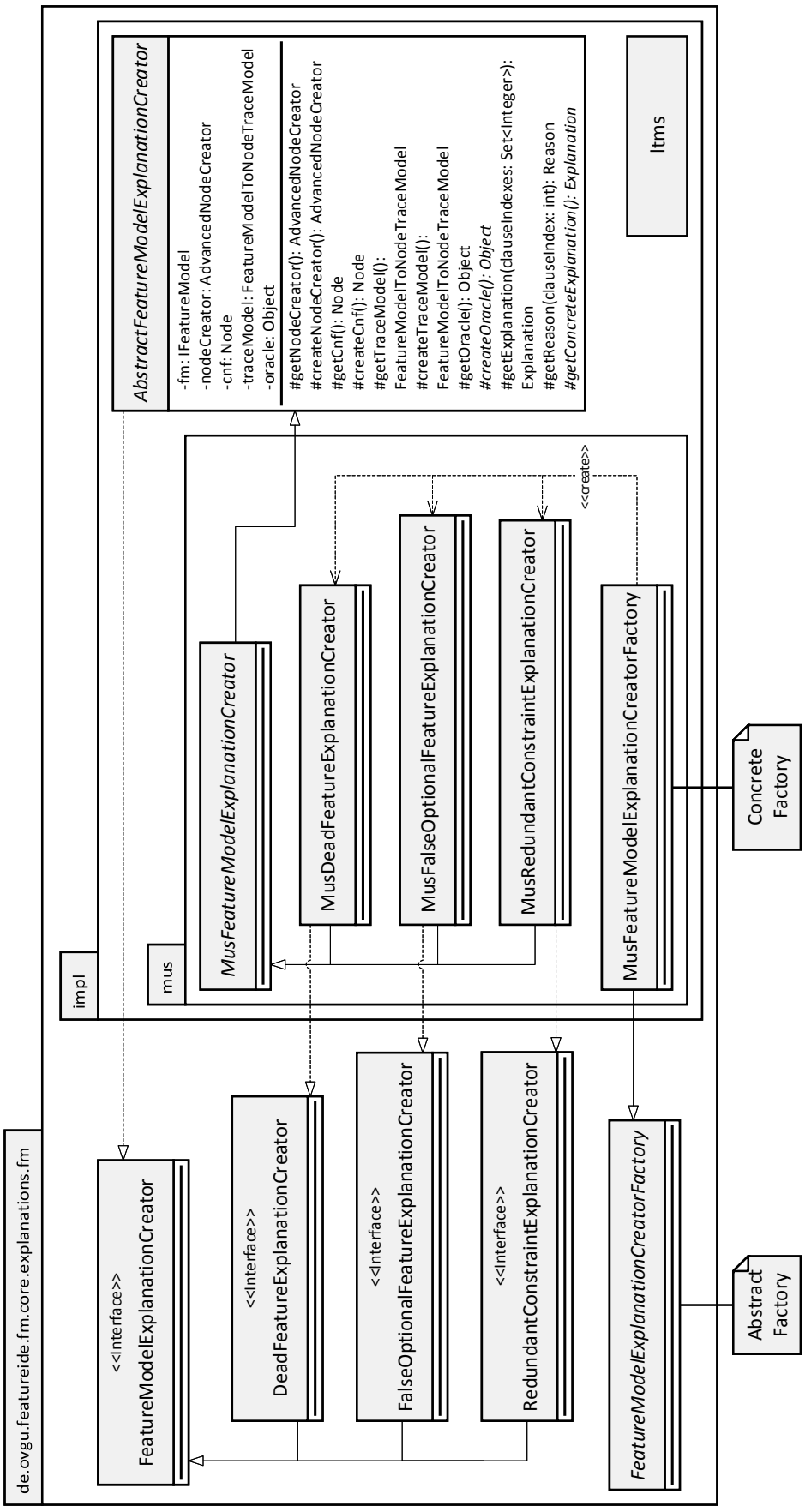


Figure 4.3: Class diagram for the implementation of explanations for feature models.

what the abstract method `getConcreteExplanation()` does. It needs to be implemented by the concrete subclasses of `AbstractFeatureModelExplanationCreator` to return a new and empty but concrete explanation with the subject already set.

Finding Concrete Explanations Using a Minimal Unsatisfiable Subset Extractor: `MusFeatureModelExplanationCreator`

The concrete subclasses of `AbstractFeatureModelExplanationCreator` all find explanations for a specific type of feature model defects using either a minimal unsatisfiable subset extractor or a logical truth maintenance system based on Boolean constraint propagation and are implemented in the subpackages `mus` and `ltms` respectively. However, being the focus of this thesis, only the former is discussed. Besides, the latter is completely analogous to the former anyway and more or less only uses a different oracle and finds not just one explanation but multiple ones to choose the shortest one among them.

Indeed, the oracle is the focus of the intermediate abstract class `MusFeatureModelExplanationCreator` that extends the class `AbstractFeatureModelExplanationCreator`. The oracle is an instance of `MusExtractor` (cf. Section 4.4) and does the actual reasoning over the formula by extracting a minimal unsatisfiable subset from it. The oracle is cached just like the resources of the superclass. It is reset when the feature model formula changes and is recreated when needed, i.e., when an explanation is being found. The feature model formula is automatically added upon creation of the oracle as all defects use the feature model formula.

All that is left to do for the three concrete subclasses of `MusFeatureModelExplanationCreator` is to add the remaining clauses of the satisfiability query to the oracle, execute it to obtain the minimal unsatisfiable subset, and return the explanation found using the method `getExplanation(Set<Integer>)`.

`MusDeadFeatureExplanationCreator` finds explanations for dead features. It adds the assumption to the oracle that the dead feature is false before executing the satisfiability query for the minimal unsatisfiable subset. To explain void feature models, the root feature is used as the dead feature, but `MusDeadFeatureExplanationCreator` is not aware of this. It simply treats the root feature like any other dead feature. In any case, after the execution of the query, the assumption is removed from the oracle using the method `MutableSatSolver#pop()` detailed in Section 4.4. By only removing the assumption, the feature model formula does not have to be added again when finding the explanation for the next dead feature in the same feature model. Instead, only the assumption changes. This leaves the oracle the option to make use of incremental SAT solving techniques for a performance boost [ES03].

`MusFalseOptionalFeatureExplanationCreator` explains false-optional features. It adds two assumptions to the oracle: that the false-optional feature is false and that its parent feature is true. From that point on, it works the same way as `MusDeadFeatureExplanationCreator`.

Finally, `MusRedundantConstraintExplanationCreator` finds explanations for redundant constraints. However, because the satisfiability query includes the formula

of the feature model without the redundant constraint, the generation of the feature model formula resource is done differently. The instance of `AdvancedNodeCreator` is told not to include any constraints in the formula. Instead, the constraints are added to the oracle manually when finding the explanation so the redundant constraint can be negated before being added. All added constraints are also removed afterward to reset the state of the oracle for the next call to `getExplanation()`.

These three concrete subclasses of `MusFeatureModelExplanationCreator` are the concrete products of the concrete factory `MusFeatureModelExplanationCreatorFactory`. As such, clients can easily access them together using the factory methods of `MusFeatureModelExplanationCreatorFactory` instead of explicitly calling their constructors.

4.2.2 Explanations for Configurations

The next use case for explanations are configurations. Due to the strong link between configurations and feature models, finding explanations for configurations automatically involves finding explanations for feature models. Therefore, finding explanations for configurations reuses some of the classes for finding explanations for feature models.

Figure 4.4 shows the contents of the subpackage `config`, which deals with finding explanations for configurations. The extension pattern is the same as for feature models, except that, instead of three concrete explanation types, there is only one, specifically for automatically selected and automatically unselected features.

Data Types: `ConfigurationExplanation` and `ConfigurationReason`

Unsurprisingly, the abstract class `ConfigurationExplanation` is an extension of `Explanation` holding the configuration context. Likewise, just as `FeatureModelReason` contains an element of the feature model, `ConfigurationReason` contains an element of a configuration, i.e., a feature selection. The feature selection object keeps track of automatic as well as manual selections, though only manual selections are used as reasons, whereas automatic ones are explained as pointed out in Section 3.2.2. These explanations of automatic selections are represented by the concrete class `AutomaticSelectionExplanation`, which assumes that the subject is the automatic selection part of a feature selection object.

Transformation to Natural Language: `ConfigurationExplanationWriter`

The transformation to natural language is accomplished using the abstract class `ConfigurationExplanationWriter`. When transforming an instance of `ConfigurationReason`, it simply refers to the manual selection state of the contained feature selection. However, since instances of `ConfigurationExplanation` can hold not only instances of `ConfigurationReason` but also instances of `FeatureModelReason`, `ConfigurationExplanationWriter` needs to be capable of transforming these as well. To this end, it extends `FeatureModelExplanationWriter` and delegates the transformation to its superclass to handle any instances of `FeatureModelReason` it encounters. An alternative to this design would be to have a writer type for each reason type, i.e., `FeatureModelReasonWriter` and `ConfigurationReasonWriter`. Creating another object for each item of the explanation just to transform it to natural language is rather excessive, though.

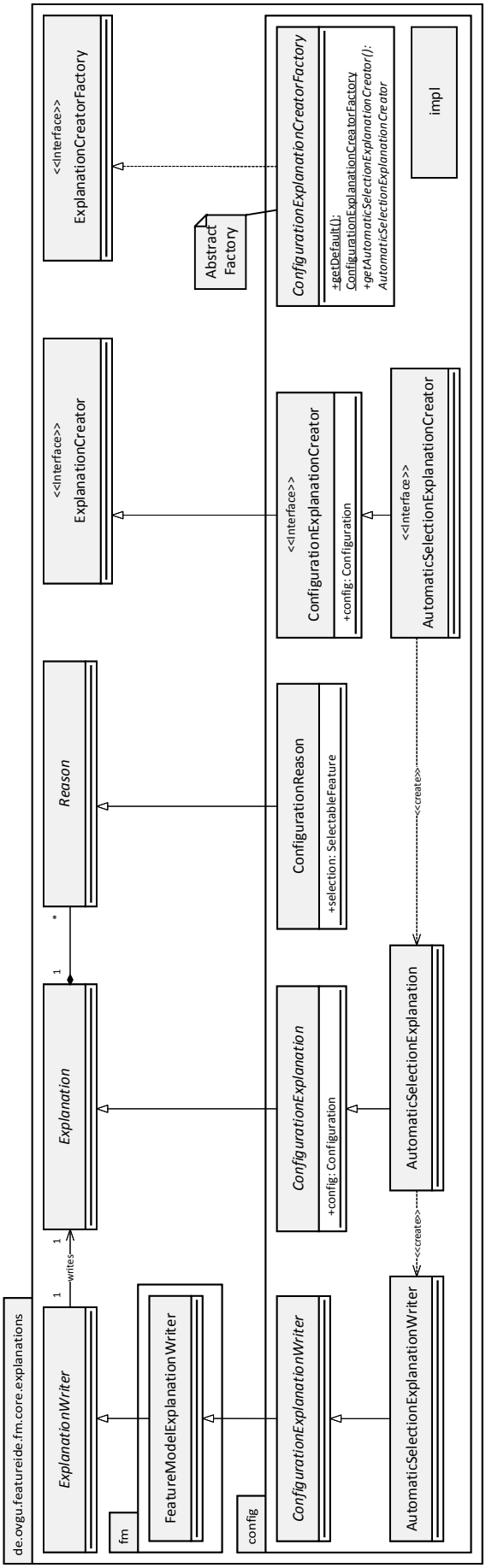


Figure 4.4: Class diagram for explanations for configurations.

Finding Explanations: `ConfigurationExplanationCreator`

The interface `ConfigurationExplanationCreator` handles finding explanations for configurations. It is only extended by the interface `AutomaticSelectionExplanationCreator`, which handles finding explanations for automatic selections in configurations and thus assumes an automatic selection as its subject.

Accordingly, the abstract factory [GHJV95] `ConfigurationExplanationCreatorFactory` is only capable of producing instances of `AutomaticSelectionExplanationCreator`. The concrete factories and their products using either of the two oracles, i.e., minimal unsatisfiable subset extractors and Boolean constraint propagation, are contained inside the subpackages of the subpackage `impl`.

The contents of the package `impl` are visualized in Figure 4.5. It contains the subpackages `mus` and ltms for finding explanations using the respective approaches.

Finding Concrete Explanations: `AbstractConfigurationExplanationCreator`

The package `impl` holds a single abstract class `AbstractConfigurationExplanationCreator`, which in turn contains the functionality for finding explanations involving configurations used by either of the two approaches. It reuses all of the functionality of `AbstractFeatureModelExplanationCreator`, i.e., managing the resources such as the oracle and the feature model formula in conjunctive normal form. Here, the distinction between finding explanations involving feature models and finding explanations for feature model defects mentioned at the beginning of Section 4.2.1 becomes important. Because `AbstractFeatureModelExplanationCreator` only finds explanations involving feature models, it can be extended by `AbstractConfigurationExplanationCreator`, which needs to find explanations involving not only configurations but also feature models by association. Hence, no code for finding explanations in any specific context is duplicated.

Finding Concrete Explanations Using a Minimal Unsatisfiable Subset Extractor: `MusConfigurationExplanationCreator`

The functionality for finding explanations for configurations in general and automatic selections in specific using minimal unsatisfiable subset extractors resides in the subpackage `mus`. The abstract class `MusConfigurationExplanationCreator` handles the creation of the oracle, that is the instance of `MusExtractor` from the SAT solver facade (cf. Section 4.4).

Using this oracle, its only concrete subclass `MusAutomaticSelectionExplanationCreator` extracts a minimal unsatisfiable subset from the satisfiability query detailed in Section 2.3.2. This subset is expressed as an explanation in model space using the trace model for clauses originating from the feature model and using the respective feature selection for clauses originating from the configuration. The latter does not require an additional trace model because its clauses only ever contain exactly one distinct feature and are therefore trivial to trace back.

The concrete factory creating instances of `MusAutomaticSelectionExplanationCreator` is `MusFeatureModelExplanationCreatorFactory`. It makes it possible to easily exchange the algorithm used to find the explanations.

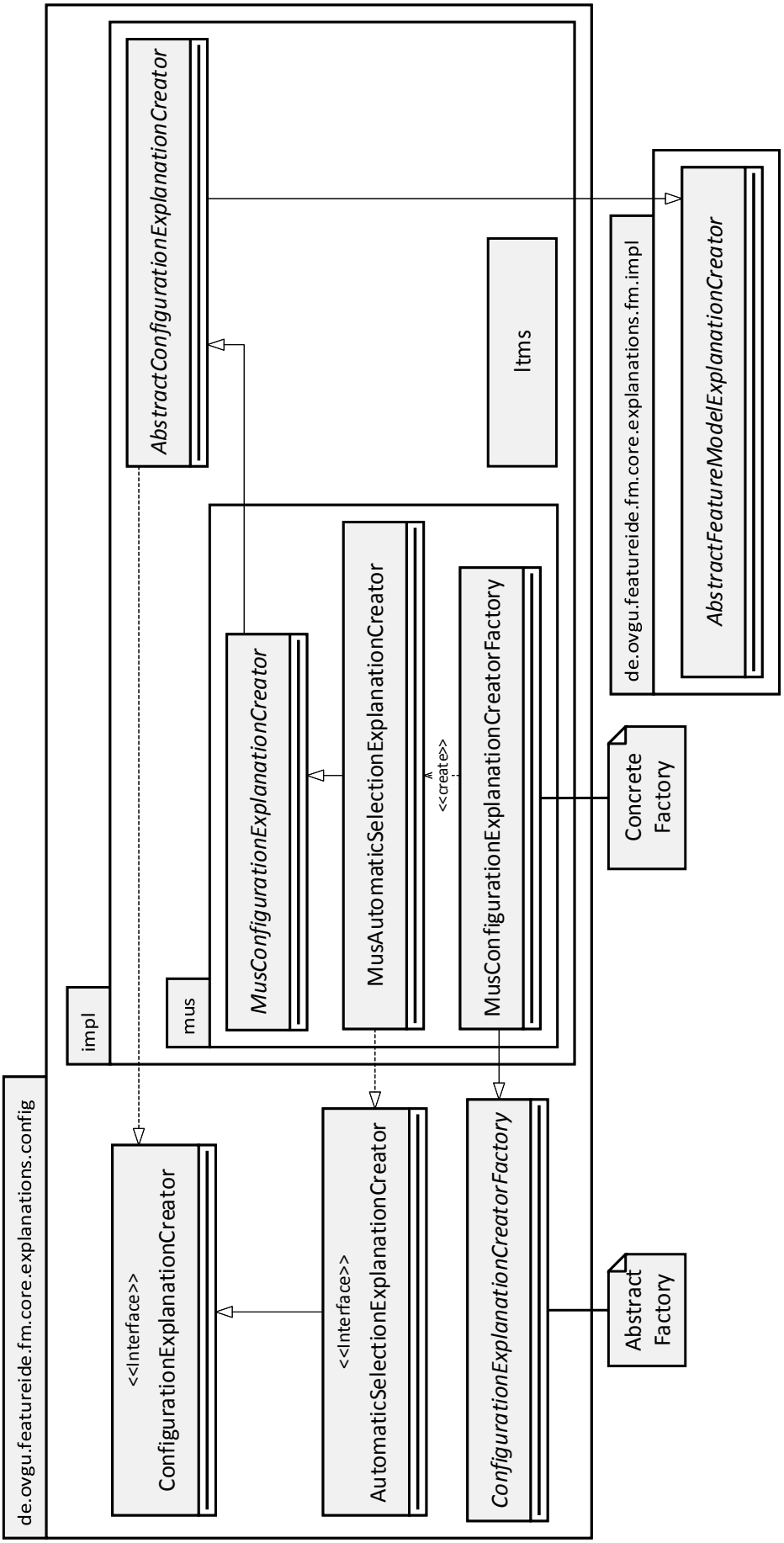


Figure 4.5: Class diagram for the implementation of explanations for configurations.

4.2.3 Explanations for Preprocessor Directives

The final use case for explanations are preprocessor directives. Finding these is analogous to finding explanations for configurations. In both cases, because the feature model is involved, functionality for finding explanations for feature models is reused.

Figure 4.6 shows the contents of the subpackage `preprocessors`, which is responsible for finding explanations involving preprocessors. The class hierarchy should seem familiar at this point.

Data Types: `PreprocessorExplanation` and `PreprocessorReason`

Unlike the other direct subclasses of `Explanation`, the abstract class `PreprocessorExplanation` does not define a context, i.e., a preprocessor object in this case. This is because FeatureIDE does not model preprocessors in the first place. After all, whereas feature models and configurations are created and maintained by FeatureIDE, preprocessors are merely used by FeatureIDE in the form of libraries.

As usual, instances of `PreprocessorExplanation` contain any number of reasons, specifically instances of `FeatureModelReason` and more importantly instances of `PreprocessorReason`. Each instance of `PreprocessorReason` refers to a preprocessor directive's expression, which is simply a propositional formula involving features from the feature model.

The only concrete explanation type for preprocessors is the class `InvariantPresenceConditionExplanation`. It denotes an explanation for either a contradiction or a tautology in a presence condition. To know which one of these two cases the specific instance is about, each instance stores a Boolean flag.

Transformation to Natural Language: `PreprocessorExplanationWriter`

The transformation to natural language using the class `PreprocessorExplanationWriter` works as usual, with this class supplying the logic for the preprocessor context. Like `ConfigurationExplanationWriter` for transforming explanations for configurations, it extends `FeatureModelExplanationWriter` in case it needs to transform instances of `FeatureModelReason`.

Finding Explanations: `PreprocessorExplanationCreator`

The interface `PreprocessorExplanationCreator` allows finding explanations involving preprocessors. To this end, it requires the expression stack of the preprocessor. The expression stack contains all of the expressions that the expression to explain is nested in. Each of these expressions is already negated properly in case it is part of the if-branches that were not taken in favor of an else-statement. On top of the stack is the expression to be explained.

The interface `InvariantPresenceConditionExplanationCreator` extends `PreprocessorExplanationCreator` to allow finding explanations for invariant presence conditions. In order to do so, it needs to know whether it should explain a contradiction or a tautology. For this purpose, each instance stores a corresponding Boolean flag just like the instances of `InvariantExpressionExplanation`.

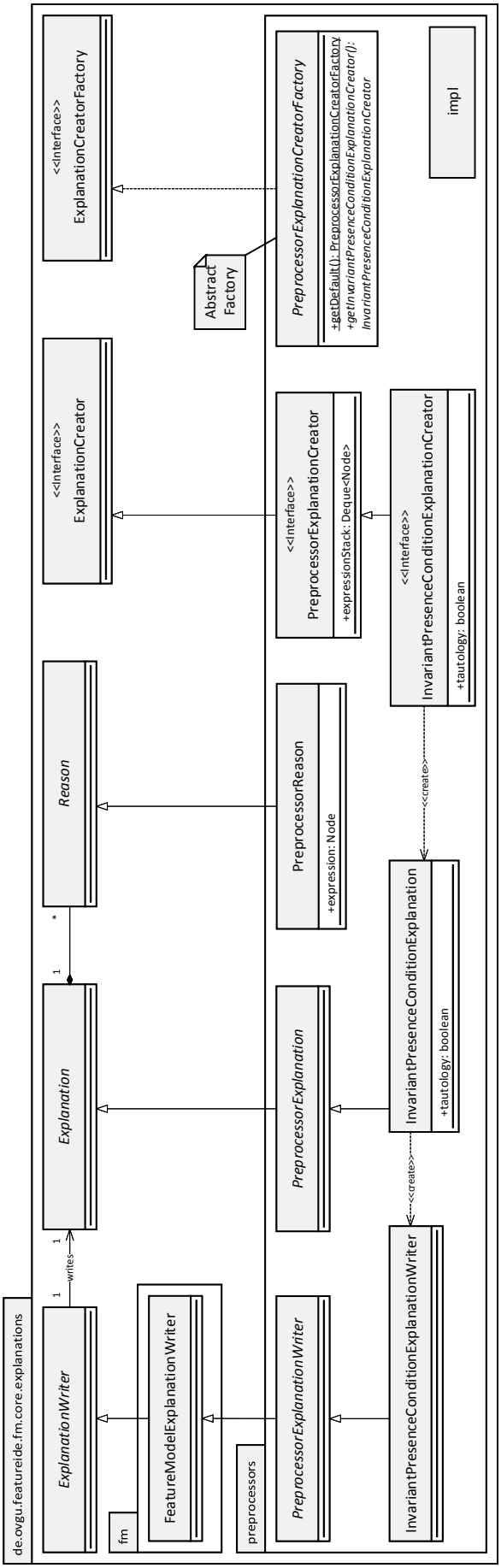


Figure 4.6: Class diagram for explanations for preprocessors.

The abstract factory [GHJV95] `PreprocessorExplanationCreator` provides means for instantiating concrete instances of `InvariantPresenceConditionExplanationCreator`. As always, this allows the underlying algorithm to be exchanged easily.

The concrete instances are contained in the subpackages of the subpackage `impl`, the contents of which are shown in Figure 4.7. Specifically, they either use minimal unsatisfiable subset extractors (in the `mus` subpackage) or Boolean constraint propagation (in the `ltms` subpackage).

Finding Concrete Explanations: `AbstractPreprocessorExplanationCreator`

The functionality used by both approaches, i.e., storing the expression stack, lives in `AbstractPreprocessorExplanationCreator`. Just like `AbstractConfigurationExplanationCreator`, it extends `AbstractFeatureModelExplanationCreator` to reuse its functionality regarding finding explanations involving feature models.

Finding Concrete Explanations Using a Minimal Unsatisfiable Subset Extractor: `MusPreprocessorExplanationCreator`

The classes using minimal unsatisfiable subset extractors to find explanations for preprocessor directives reside in the package `mus`. The basic explanation creator for that is `MusPreprocessorExplanationCreator`, which uses an instance of `MusExtractor` from the SAT solver facade (cf. Section 4.4) as oracle.

The only subclass of `MusPreprocessorExplanationCreator`, `MusInvariantPresenceConditionExplanationCreator`, uses that oracle to find explanations for invariant presence conditions by extracting the minimal unsatisfiable subset from the satisfiability queries from Section 2.3.3. For contradictions, the satisfiability query is built by forming the conjunction of the feature model formula and the presence condition, i.e., each expression from the expression stack. The same is done for tautologies, except that the top element of the expression stack, which is the expression to be explained, is negated first. In both cases, it is remembered which clauses stem from which expression so they can be traced back when building the explanation from the minimal unsatisfiable subset. Due to the simplicity of this, no extra trace model is defined.

Finally, the instances of `MusInvariantPresenceConditionExplanationCreator` are created by the concrete factory `MusPreprocessorExplanationCreatorFactory`. Using this factory pattern makes it possible to quickly change to a different approach for finding explanations such as Boolean constraint propagation.

4.3 From Formula to Feature Model Using a Trace Model

As mentioned in Section 3.2, finding meaningful explanations requires the clauses from the minimal unsatisfiable subset to be transformed from their formula representation to the representation in the model they originate from. For preprocessor directives and configurations, this is straight-forward or even trivial. However, for feature models, it is more complex. Because the mapping between clauses and

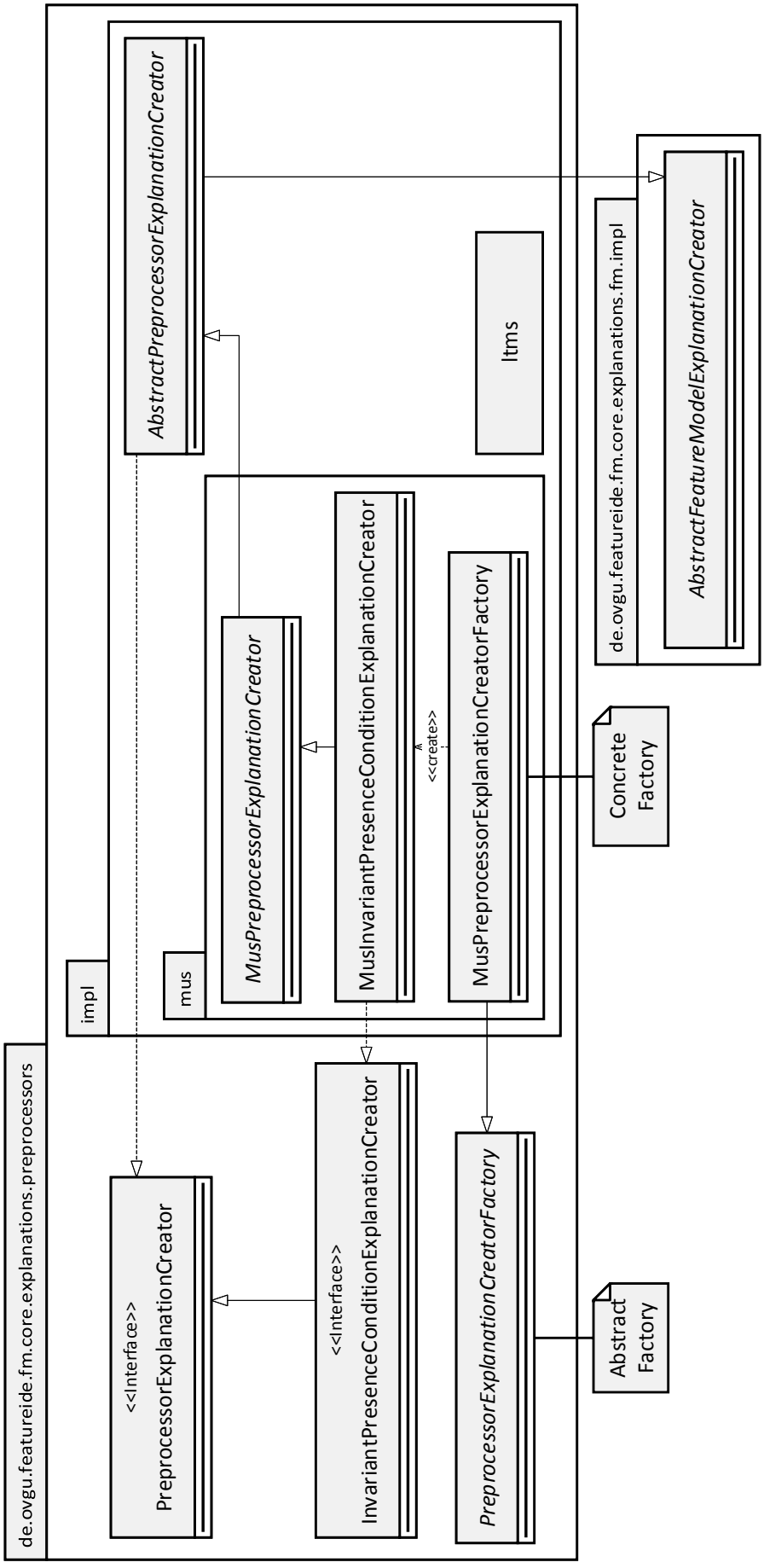


Figure 4.7: Class diagram for the implementation of explanations for preprocessors.

feature model elements is ambiguous [Ana16, CW07], there needs to be a way to remember the mapping.

This is what the trace model does. After a model is transformed from one format to another, in this case a feature model to its representation as a propositional formula, the link between the elements of the two is usually lost. To solve this, a trace model keeps track of how the elements in the two models correspond to one another. Languages designed with model-to-model transformations in mind such as Query/View/Transformation (QVT) and the Atlas Transformation Language (ATL) often already come with automatically created trace models [Bie10].

However, because FeatureIDE's transformation from feature model to formula is written in plain Java, the trace model needs to be written by hand. In the implementation of the approch for finding explanations using Boolean constraint propagation [Ana16], the tracing information is encoded in an integer value added to every literal of the formula. The issue with this technique is that it binds the feature model semantics to propositional formulas, creating a dependency to a conceptually unrelated model.

To maintain the separation of concerns with the approach presented in this thesis, the tracing information is separated from the formula. This is done with the trace model class `FeatureModelToNodeTraceModel`. It resides in the package `de.ovgu.featureide.fm.core.editing`, as does the class `AdvancedNodeCreator` for transforming feature models to propositional formulas. The `AdvancedNodeCreator` simply needs to populate the trace model while performing the transformation.

The first step in modeling the contents of the trace model is to find the conceptual connection between the transformation's source and target models. The source model, i.e., the feature model, contains a number of elements, specifically features in a tree structure as well as constraints. The target model, i.e., the propositional formula, contains a number of clauses. Each feature model element is transformed to at least one clause as specified at the end of Section 2.2.2. To turn this into a more detailed, unambiguous mapping, the source elements are interpreted to be of finer granularity than the feature model elements. In particular, the various structures from Table 3.1 are used as sources to be more specific about the source elements.

This mapping is modeled in `FeatureModelToNodeTraceModel` by storing an instance of `FeatureModelElementTrace` for each target element. The traces in the trace model are the individual links of the mapping. Each trace remembers the role of each source element in the transformation, e.g., which one is the parent and which ones are the children.

In order to be able to do so, the trace also needs to know which type of transformation the source element is involved in in the first place. Hence, this information is stored as one of the following values of the enumeration `Origin`, where the variables are to be interpreted as in Section 2.2.2:

`ROOT` For the clause of the root feature.

`CHILD_UP` For clauses of the form $f \Rightarrow p$. Every relationship between a child and a parent feature involves such a clause because parent features are always selected if any of its children is selected.

`CHILD_DOWN` For clauses of the form $p \Rightarrow f$ in case of mandatory features or $p \Rightarrow \bigvee_{f \in F} f$ in case of or-groups and alternative groups.

`CHILD_HORIZONTAL` For clauses involving only children and no parent. Alternative groups create one of these for every combination of two children.

`CONSTRAINT` For the clauses of a constraint.

In the approach based on Boolean constraint propagation [Ana16], the three child relationship types are not differentiated. However, the detail of which part of the relationship is causing a defect is worthwhile information that should not be ignored when finding an explanation. As such, the approach presented in this thesis differentiates every clause if possible. With this information, every source element can be uniquely identified.

To identify the other end of the mapping, i.e., the clauses the source elements were transformed to, it would be possible to store references to the clause objects. However, querying the traces by clause object is more expensive as it requires object comparison. Also, duplicate clauses in the formula could not be discerned. Therefore, the clauses and therefore the traces are identified by the clause index, which is always known given the target formula.

Finally, it should be noted that generating the trace model is not a cheap operation due to the many references that need to be stored. As such, generating the trace model is disabled by default. Clients wishing to use the trace model, especially the algorithms for finding explanations, need to enable tracing for their own instance of `AdvancedNodeCreator` before transforming the feature model to a formula.

4.4 Exchanging SAT Solvers Using a Solver Facade

An important ingredient in the implementation of the approach for finding explanations presented in this thesis are minimal unsatisfiable subset extractors. The Prop4J library for propositional formulas that FeatureIDE comes with provides not only data classes for formulas but also adapters [GHJV95] that enable using them with the SAT solvers provided by the Sat4J library [LBP10]. Fortunately, the Sat4J library offers minimal unsatisfiable subset extractors. Unfortunately, the Prop4J library does not provide adapters for them.

This section discusses the implementation of a facade [GHJV95] for SAT solvers and especially minimal unsatisfiable subset extractors using Sat4J to solve this issue. Being a facade, its goal is to simplify the use of the wrapped library by hiding the implementation details of the underlying implementation through abstraction. The opportunity is used to design the facade to be extensible such that not only Sat4J may be used in the future but also any other SAT solver library.

The SAT solver facade is located in the `org.prop4j.explain.solvers` package, which is part of the `de.ovgu.featureide.fm.core` plug-in for ease of access for FeatureIDE developers. The contents of the package are shown in Figure 4.8. The

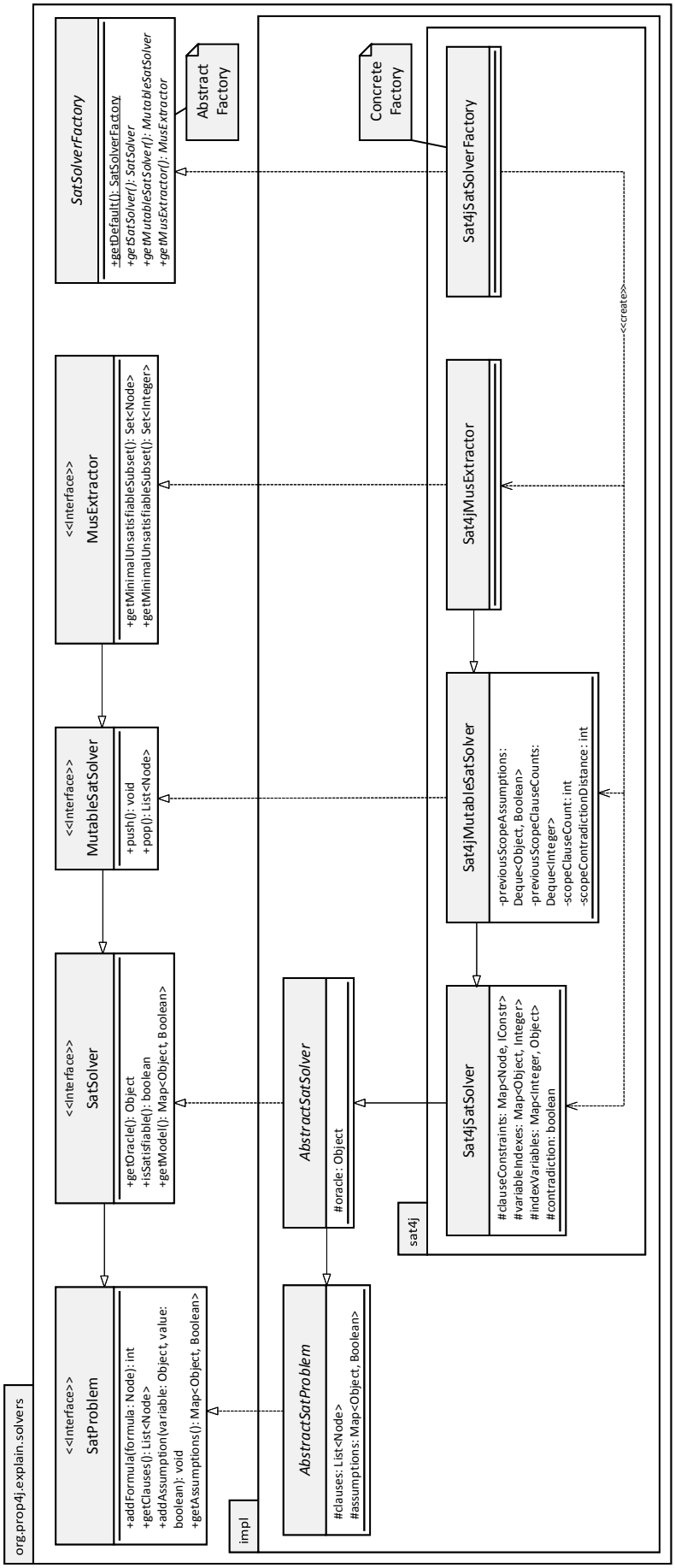


Figure 4.8: Class diagram for the SAT solver facade.

facade strictly separates the implementation from the interfaces defined on its root level. In general, the interfaces are kept as small as possible so they can be implemented using any SAT solver regardless of the number of features it offers. The extension hierarchy of the interfaces also makes it possible to provide functionality only up to the point supported by the concrete SAT solver, e.g., checking satisfiability queries for satisfiability but not extracting minimal unsatisfiable subsets. The extension hierarchy additionally makes the separation of concerns in the implementation of the interfaces easier. The rest of this section leads through the extension hierarchy.

Data Type: `SatProblem`

The most basic interface of this facade is `SatProblem`. It only defines functionality for adding formulas in conjunctive normal form and truth value assumptions for given variables to it. This interface is fairly standard in SAT solving and allows for incremental SAT solving [ES03].

Each added formula is transformed to conjunctive normal form if necessary and stored as a list of clauses to be queried later [EMS07]. It is worth noting that, in order to keep the demands of the interface to a minimum, it provides no functionality for removing any of the clause once added, which happens later in `MutableSatSolver`.

Assumptions may also be added but not removed. An assumption means that the given variable is always assigned the given truth value when reasoning over the formula. Theoretically, assumptions can be seen as clauses that only contain the literal of the given variable with the respective negation conforming to the truth value. In practice, many SAT solvers work faster with explicit assumptions than with singleton clauses. Furthermore, since they are not actually clauses of the formula, assumptions do not show up in minimal unsatisfiable subsets and therefore also not in explanations, which makes them useful for hiding trivial information.

SAT Solving: `SatSolver`

The first functionality for executing the satisfiability query added to an instance of `SatProblem` is defined in the interface `SatSolver`. Its most important method is `isSatisfiable()` for deciding the satisfiability of the contained formula with the contained assumptions. If that turns out to be satisfiable, the satisfying truth value assignments found can be queried using `getModel()`. This is probably done using a SAT solver from a library, though the oracle could also refer to `this` or even another instance of `SatSolver` to be decorated.

Mutable SAT Solving: `MutableSatSolver`

Further functionality is added to `SatSolver` in the interface `MutableSatSolver`. `MutableSatSolver` provides functionality for removing clauses previously added to the solver.

This is achieved by treating the solver as a stack of scopes, like onion layers made of formulas and assumptions that can be added or removed. A scope can be added to the stack with `push()`. Then, all formulas and assumptions added to it are

simultaneously removed when its scope is removed with `pop()`. By adding multiple scopes to the stack, each scope can be made to contain only specific parts of the formula that can then be removed in as fine a granularity as wanted. Of course, executing the satisfiability query needs to take all scopes of the stack into account.

The idea of this design of a solver as a stack is lifted from SMT-LIB [BFT17], a language for interacting with satisfiabiltiy modulo theories solvers. Here, it is applied to SAT solvers. The advantage of this design is that it makes removal easier by treating clauses that belong together in the formula as one as defined by the scope boundaries. Additionally, this does not require the client to keep any handles on the added clauses to remove them later. For example, a clause added to a Sat4J solver can only be removed using the handle of the type `IConstr` returned after adding it. Instead, all that needs to be remembered is which scope of the stack the clauses where added to, which is often trivial due to the formula layout.

Extracting Minimal Unsatisfiable Subsets: `MusExtractor`

The last interface in the SAT solver hierarchy is the interface `MusExtractor`. It is as simple as one would expect and only provides functionality for extracting a minimal unsatisfiable subset from the added formula [BLMS12]. This can be done either while staying in the abstraction level of the `Node` class of Prop4J or by referencing the clauses by the index in which they occur in the formula. The latter is useful in case multiple semantically equal instances of `Node` were added as clauses to the solver. The difference between these clauses might for instance matter when looking them up in the trace model (cf. Section 4.3) and thus resulting in a different trace for the instance of `FeatureModelReason` (cf. Section 4.2.1).

The three solver interfaces `SatSolver`, `MutableSatSolver`, and `MusExtractor` are the abstract products of the abstract factory [GHJV95] `SatSolverFactory`. This is to easily access and exchange their specific related implementations.

Implementation Using Sat4J

The implementations of the SAT solver facade interfaces are located inside the subpackage `impl`. The abstract classes `AbstractSatProblem` and `AbstractSatSolver` provide abstract implementations of the respective interfaces they implement. The former keeps track of the added formulas and assumptions and in particular ensures that the clauses are added in conjunctive normal form while the latter stores the oracle for its concrete subclasses.

Currently, the only concrete subclasses are located in the subpackage `sat4j` and use the solvers from the Sat4J library [LBP10]. The first concrete subclass is `Sat4jSatSolver`, which extends `AbstractSatSolver`. It uses the default oracle provided by the class `SolverFactory` from Sat4J to execute the added satisfiability query.

SAT Solving Using Sat4J: `Sat4jSatSolver`

The class `Sat4jSatSolver` extends the behavior of `AbstractSatProblem` by making sure that all the added formulas and assumptions also make it to the underlying

oracle. To this end, the solver facade first needs to declare every variable used throughout the formula. This in turn requires translating the `Node` instances from Prop4J to the format used by Sat4J. The solver brings several helper functions for converting the formula formats of Prop4J and Sat4J, but since they are fairly specific to these libraries, they are not discussed here for the sake of brevity.

Should a contradicting clause be added to the oracle, Sat4J signals an exception, which is caught and remembered by the solver facade. In such a case, checking satisfiability does not even require a call to the oracle as the formula is known to be unsatisfiable anyway. If no immediate contradiction was found while adding clauses, the satisfiability check can go on as normal using the oracle with its added formulas and additional assumptions, the latter of which need to be added again for every satisfiability query.

Mutable SAT Solving Using Sat4J: `Sat4jMutableSatSolver`

The class `Sat4jMutableSatSolver` provides functionality for removing clauses and assumptions using the stack-based approach discussed earlier in this section. To model this stack, the class `Sat4jMutableSatSolver` keeps track of the clauses and the assumptions that were added in each scope. This is done by storing the data from the previous scopes in addition to the data from the superclass. To keep the superclass method implementations intact, the accessor methods of its resources are overridden to take into account not only the current scope but all previous scopes as well.

The new assumptions of the current scope and the assumptions that were added in each previous scope are stored separately. Thus, when accessing the assumptions for executing the satisfiability query, the assumptions are merged together, starting at the bottom of the stack so newer assumptions override older ones. Instead of storing these assumption deltas that then need to be resolved when accessing the assumptions, it would also be possible to store copies of all assumptions valid in each scope that are then simply restored when changing the scope, though this would come at the cost of a bigger memory footprint.

For the clauses, only the number of clauses that were added in each scope needs to be stored. Those many of the newest clauses are removed from the oracle using the respective clause handles when `pop()` is called. However, Sat4J does not free up the constraint's index from the vocabulary when a clause is removed.[2] In the local clause list, the resulting gaps in the index range are modeled using `null` values. To account for this, the accessor methods related to clauses are overridden to skip `null` values in the clause list.

The last resource that is affected by scopes and removal is the flag of whether a contradiction was just added. Removing the clause that introduced the contradiction clears the contradiction. As such, the scope containing that clause needs to be remembered. This is done by storing the distance to it, i.e., how often `pop()` needs to be called until the scope containing the contradiction is reached. Calling `push()` increases that number while `pop()` decreases it. If it is at 0, the current scope is the one containing the contradiction, and calling `pop()` removes the contradiction, which is reflected by updating the contradiction flag.

[2] http://www.sat4j.org/maven234/apidocs/org/sat4j/specs/ISolver.html, accessed 2017-09-11

Extracting Minimal Unsatisfiable Subsets Using Sat4J: `Sat4jMusExtractor`

Finally, the class `Sat4jMusExtractor` extends the class `Sat4jMutableSatSolver` to add the functionality of extracting minimal unsatisfiable subsets. This is done by decorating the oracle with an instance of `Xplain` from Sat4J, which grants access to minimal unsatisfiable subsets from the underlying solver. Once more, the format is automatically transformed by the solver facade. The resulting minimal unsatisfiable subset can be used by any client, most notably for finding explanations.

All three of these concrete solver classes using a Sat4J oracle, `Sat4jSatSolver`, `Sat4jMutableSatSolver`, and `Sat4jMusExtractor`, are the concrete classes of the concrete factory `Sat4jSatSolverFactory`. That makes it currently the only concrete factory extending the abstract factory `SatSolverFactory`. Still, should other SAT solving libraries be added to the solver facade in the future, using the new implementation only requires switching the factory instance.

4.5 Visualizing Explanations

Up to this point, only components falling into the model category of the model-view-controller pattern underlying FeatureIDE were discussed. This section introduces functionality belonging to the other categories of the model-view-controller pattern. In other words, this section details how the explanations may be accessed through FeatureIDE's graphical user interface and how, once found, the explanations actually end up being visualized for the user to make sense of. If the visualization is intuitive, this greatly helps the user to parse the explanation.

As always, each concrete use case is discussed in its own subsection: feature model defects in Section 4.5.1, configurations in Section 4.5.2, and preprocessor directives in Section 4.5.3.

4.5.1 Explanations for Feature Model Defects

The first use case is that of feature model defects. In this implementation, visualizing explanations for feature model defects is more refined than for the other two use cases. Whereas explanations in configurations and preprocessor directives are visualized as more or less only text, the explanations in feature model are visualized as highlighted graphcial elements in the feature diagram.

The visualization of explanations for feature model defects was first implemented in FeatureIDE as part of a project work [Gü16] building upon the explanations found by the approach using Boolean constraint propagation [KAT16]. This section also outlines the implementation of that project work as it is obviously relevant to this thesis, yet it was never documented in text form before. Besides, this thesis features major changes to that implementation.

FeatureIDE's plug-in handling everything related to the graphical user interface of feature models is `de.ovgu.featureide.fm.ui`. In its subpackage `editors` lives the class `FeatureDiagramEditor`, which hooks into Eclipse to provide a graphical editor for feature diagrams. It acts as the starting point for finding and subsequently visualizing the explanations.

When a defect is found in a feature model, it is highlighted the way it is known from the feature diagrams shown throughout this thesis. The selection of such a defect feature model element is interpreted as interest in an explanation for the defect. Hence, once a defect element is selected, an explanation is found using one of the three concrete explanation creators from Section 4.2.1 (or taken from the editor's cache if an explanation for the selected defect was already found before). Only up to one explanation may be active at a time. Then, the explanation is shown to the user.

To start the visualization of the explanation, the feature diagram editor notifies each feature model element's edit part that there is a new active explanation. To avoid having each element in turn search the entire explanation for what reason it is involved in, the feature diagram editor already specifies the corresponding reason for each involved element when notifying.

The edit part of each involved element then forwards the reason to the figure of the element. Upon this, the figure, being part of the view, finally changes its appearance in accordance to the received reason. What this actually looks like depends on the figure and the reason. In general, the respective graphical element is highlighted by setting its line color to a color on the gradient from red to black depending on the reason's confidence (cf. Section 4.2).

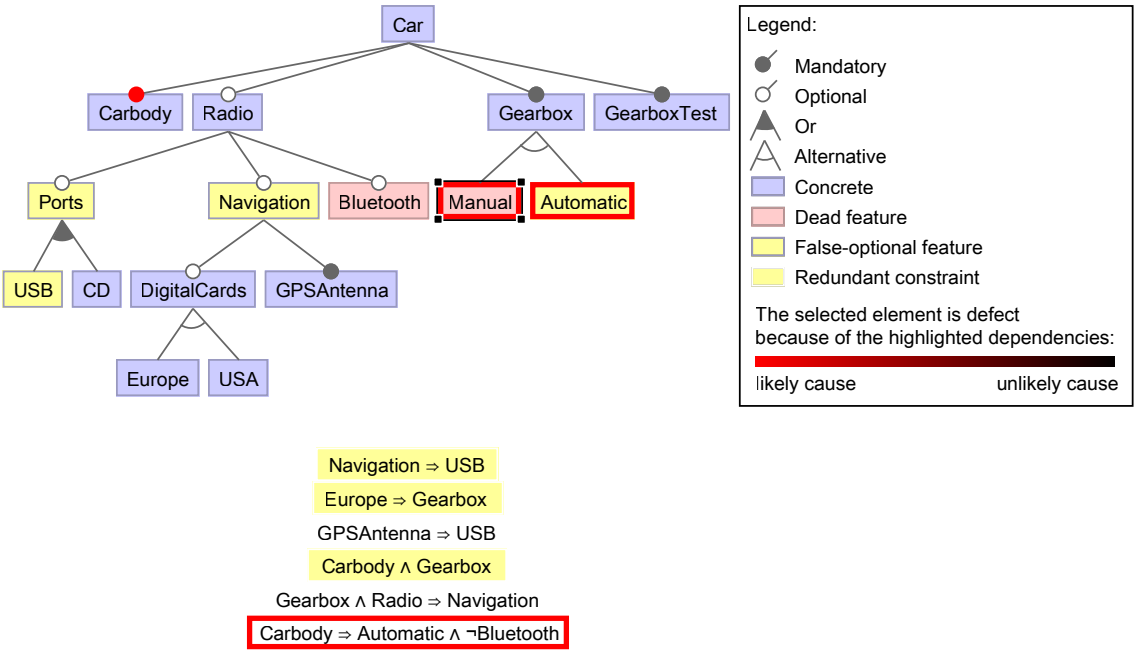
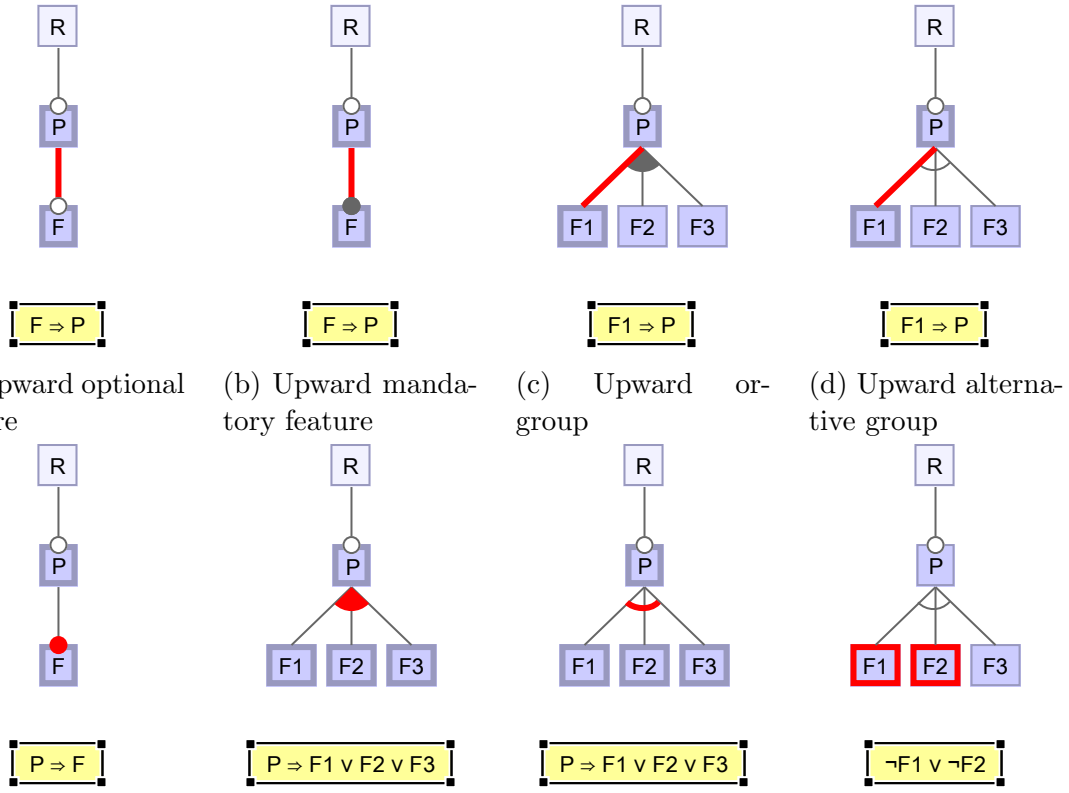


Figure 4.9: Visual explanation for why *Manual* is a dead feature.

An example of how this looks is shown in Figure 4.9. In this example, the explanation why *Manual* is a dead feature is visualized. The explanation in text form can be found in Table 3.2b on page 30. Looking at that table, it should be obvious how each element is highlighted according to its involvement in the explanation. The mandatory decoration of *Carbody* is highlighted because of the second reason (it is a mandatory child of *Car*), the last constraint because of the third one (it is a constraint), and *Manual* and *Automatic* because of the fourth one (the two are

alternatives). Only the first reason regarding the root feature is not visualized since it is trivial. On top of the visualization, the tooltip of each highlighted element contains the textual reason for clarification.



(a) Upward optional feature (b) Upward mandatory feature (c) Upward or-group (d) Upward alternative group

(e) Downward mandatory feature (f) Downward or-group (g) Downward alternative group (h) Horizontal alternative group

Figure 4.10: Visual explanations for feature structures.

As this example suggests, the visualization of each feature structure depends on which reasons pertain to it. Indeed, Figure 4.10 shows the visualization for each possible combinations of structure type (optional feature, mandatory feature, or-group, and alternative group) and child relationship direction (upward, downward, and horizontal) from the trace model (cf. Section 4.3). In all these examples, the sole constraint expresses one of these combinations, which makes the constraint redundant. As a result, the explanation of the redundancy contains only the reason for that combination to serve as an illustration of how that specific combination is visualized.

In the implementation of the approach using Boolean constraint propagation [Ana16], none of these directions are differentiated. Therefore, when a feature structure is the reason of an explanation found with that implementation, all parts of the feature structure have to be highlighted. The big difference between the visualization from the project work [Gü16] and the visualization presented in this thesis is that now the direction is differentiated as explained above. Each part of the feature structure is only highlighted if applicable. Thus, because no information is lost in the transformation from clause of the minimal unsatisfiable subset to its visualization, the user can understand the reasoning underlying the explanation more easily.

4.5.2 Explanations for Configurations

Next, explanations for configurations are visualized. This is implemented in the same plug-in as the visualization of explanations for feature model defects, that is `de.ovgu.featureide.fm.ui`. The editor that handles configurations, `ConfigurationTreeEditorPage`, lives in the subpackage `editors.configuration`. Analogous to the feature model editor, the configuration editor uses the explanation creators from Section 4.2.2 to find the explanations when necessary.

The graphical visualization of the explanations for configurations is more difficult than the visualization of explanations for feature model defects. After all, not only configuration elements inside the configuration editor need to be highlighted but also any referenced elements in the feature model. Therefore, in addition to the configuration editor, the feature diagram would have to be visible whenever an explanation for a configuration is visualized graphically. This cannot simply be assumed to be the case, so the feature diagram would have to be opened for the user and displayed next to the configuration when an explanation is required.

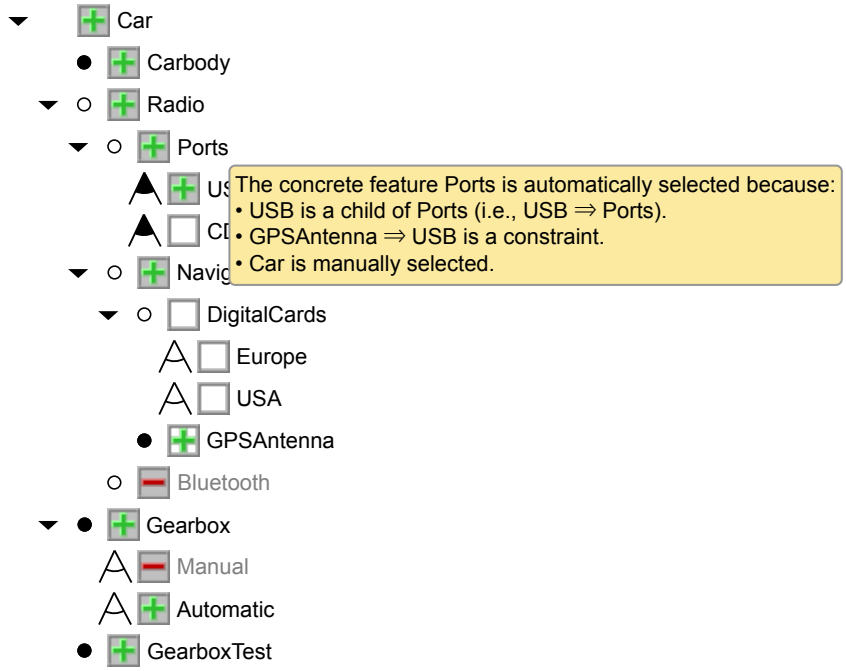


Figure 4.11: Textual explanation for the automatically selected feature *Ports*.

Instead, to keep the user interaction simple, the visualization of explanations for configurations is not done graphically but textually. Specifically, the explanations for automatic selections are written in natural language in the tooltip of the automatically selected feature in the configuration editor. As an example, Figure 4.11 shows what it looks like when the user hovers the mouse cursor over the automatically selected feature *Ports* in the configuration from Figure 2.6. The transformation to natural language is done by the explanation writers from Section 4.2.2.

4.5.3 Explanations for Preprocessor Directives

Finally, explanations for preprocessor directives are visualized. This is analogous to the visualization of explanations for configurations. Because explanations for pre-

processor directives also involve the feature model, the same reasoning for choosing a textual over a graphical visualization applies.

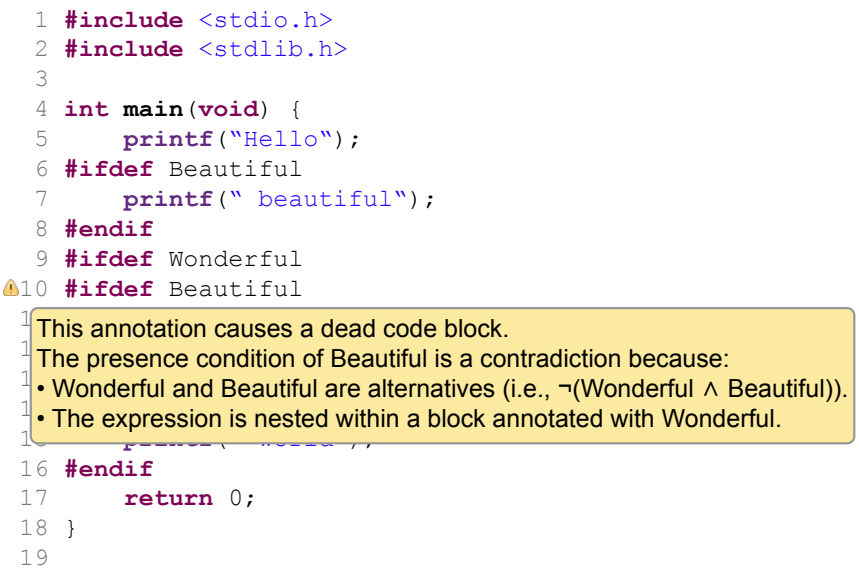


Figure 4.12: Textual explanation for a contradiction in a CPP preprocessor directive.

Unlike the visualization of the explanations for the other two use cases, the visualization of explanations for preprocessor directives is implemented in the plug-in `de.ovgu.featureide.core`. The class `PPComposerExtensionClass` in the subpackage `builder.preprocessor` checks for invariant presence conditions in preprocessor directives. As usual, it uses the explanation creators from Section 4.2.3 to find an explanation when it encounters such a defect. The explanation is then added to the warning marker that is used to notify the user of the defect. An example of how it looks when the mouse curvor is hovered over such a warning marker can be found in Figure 4.12.

4.6 Summary

This chapter showed one possible way to implement the approach presented in this thesis using a highly extensible architecture for finding abstract explanations. This involved a trace model and minimal unsatisfiable subset extractors accessed through a SAT solver facade implemented with Sat4J. The resulting explanations are visualized graphically in feature diagrams or written in natural language in the cases of configurations and code with preprocessor directives.

The question remains how the approach presented in this thesis performs in comparison to the previous explanation approach based on Boolean constraint propagation. This is answered in the next chapter.

5. Evaluation

The previous chapters presented an approach for finding explanations for satisfiability queries in a software product line context using minimal unsatisfiable subsets. The conceptual framework was realized as an addition to the software product line development tool FeatureIDE. Given this implementation, it is now possible to examine the algorithm from a practical point of view. Thus, in this chapter, the algorithm is evaluated using qualitative observations and quantitative measurements. It is also compared against the previous approach for finding explanations using Boolean constraint propagation [KAT16].

The models and the code used for this evaluation can be found online in the artifact repository for this thesis.[1] Alternatively, most of the models can also be found inside example projects that can be imported from FeatureIDE into Eclipse.

Before the evaluation begins, the evaluation criteria are detailed in Section 5.1. The "soft" criteria are evaluated using a qualitative analysis performed in Section 5.2. Section 5.3, by contrast, covers criteria revolving around aspects such as performance that can be measured more definitively in numbers. A conclusion from the results is drawn in Section 5.4.

5.1 Evaluation Criteria

Throughout this evaluation, various relevant characteristics of the algorithm presented in this thesis are examined. This ensures its usefulness and allows a comparison against existing approaches. This section discusses which characteristics are considered particularly relevant and are therefore evaluated in the analysis to come.

The first thing to determine is whether the explanations found by the explanation algorithm are semantically correct. This is the case if the reasons of the explanation in question do in fact imply the circumstance to explain.

However, by this definition, an explanation that contains all of the possible reasons of the context (e.g., the entire feature model when explaining a feature model defect) is

[1] https://github.com/TimoGuenther/explaining-sat-queries-for-spls

always correct. Because such an explanation is not particularly useful, it is necessary that, in addition to being correct, the explanation significantly reduces the number of possible reasons. Such explanations are more sensible because they can help the user pinpoint the relevant causes of the circumstance.

Because of this, explanations should be as short as possible. Therefore, it is analyzed how much the approach presented in this thesis affects the length of explanations compared to the approach based on Boolean constraint propagation [KAT16].

Still, even a short explanation is useless if the user does not want to wait for it to be found automatically or if the user can manually figure out the cause of the circumstance manually before an explanation is found. Hence, the performance of this approach is considered as well.

Finally, the explanation needs to be found in the first place. As detailed in Section 3.1.1, Boolean constraint propagation is incomplete, which means that explanation approaches relying on it do not always find an explanation for a given circumstance to explain. By contrast, the approach presented in this thesis is based on minimal unsatisfiable subset extractors. Therefore, it is determined how often an explanation is found using this approach where none could be found using Boolean constraint propagation and vice versa.

All of this needs to be considered not only for small models but also for large ones. Especially the large models have too many details for the user to just keep in mind all the time. In the following, these criteria are determined using both a qualitative analysis (Section 5.2) and a quantitative analysis (Section 5.3).

5.2 Qualitative Analysis

The qualitative properties of the approach for finding explanations as presented in this thesis are checked by examining the output of the algorithm by hand. Part of this can be done using the examples shown throughout this thesis as they all are the result of the approach presented in this thesis. However, the evaluation should include as many different examples as possible to discover edge cases that the algorithm struggles with. Fortunately, Ananieva already compiled a set of varied examples of small feature models containing defects to explain [Ana16].

The minimal unsatisfiable subset approach manages to find an explanation for all of these examples. The findings of the author are that all of the explanations are semantically correct, i.e., logically lead to the conclusion that the defect element is indeed defect. This is because minimal unsatisfiable subsets are unsatisfiable. They are also sensible in that they all reduce the searching space for the involved feature model elements. This is because minimal unsatisfiable subsets are minimal subsets.

The feature model defect explanations found using a minimal unsatisfiable extractor are fairly similar to those found using Boolean constraint propagation. One difference is that the former typically express the same reasoning using structural information, while the latter favors constraints. However, this should vary depending on the SAT solver used and the order of the clauses of the formula that the minimal unsatisfiable subset is extracted from. After all, which of the many minimal unsatisfiable subsets is returned depends on the underlying algorithm [dlBSW03].

In summary, the explanation approach presented in this thesis leads to correct and sensible results. However, this reveals nothing about performance, explanation lengths, or availability for models from real software projects. To quantify these remaining characteristics, a quantitative analysis follows.

5.3 Quantitative Analysis

The quantitative analysis answers the remaining research questions using concrete measurements. Of interest are the performance impact of finding explanations using minimal unsatisfiable subset extractors and the length of the explanations, i.e., the number of reasons they contain. Additionally, because Boolean constraint propagation is incomplete as detailed in Section 3.1.1, it is measured how often this actually becomes a problem and Boolean constraint propagation fails to find an explanation. The models used to test the algorithm on are presented in the following.

Feature Model	Features	Constraints	DFs	FOFs	RCs
SortingLine	39	11	0	0	6
PPU	52	15	5	4	10
Violet	101	27	0	1	1
uClibc	313	56	31	10	0
E-Shop	326	21	0	1	1
WaterlooGenerated	580	61	10	13	8
Busybox_1.18.0	854	123	18	3	0
XSEngine	1273	886	42	166	51
uClibc-Distribution	1580	197	1	6	0
Automotive01	2513	2833	195	80	869

Table 5.1: Evaluation models containing feature model defects. “DFs” is the number of dead features, “FOFs” the number of false-optional features, and “RCs” the number of redundant constraints.

The first use case is that of feature model defects. The feature models containing defects to be explained in this quantitative analysis are listed in Table 5.1. The feature models range in size, from a small feature model with just 39 features to one with 2513 features. This allows evaluating the scalability of the approach presented in this thesis. Each feature model in turn contains a varying number of defects, i.e., dead features, false-optional features, and redundant constraints. The distribution of the feature model defect types varies in all feature models. This allows identifying defect types that tend to be more difficult to explain. During the analysis, all feature model defects are explained.

The smallest two feature models in the list are from the industrial case studies of the Sorting Line and the Pick-and-Place Unit [FLVH15, KLL+14]. The largest feature model is a real example stemming from the automotive industry. Ananieva uses these three feature models in the evaluation of the explanation approach based on Boolean constraint propagation [Ana16]. For the intermediate sizes, Ananieva uses automatically generated feature models. However, real feature models are preferable as they contain defects that evidently actually come up in practice. Hence, the

remaining feature models for this analysis are instead taken from a study with the explicit intent of gathering feature models from real software projects [KTM+17]. The feature models are chosen in such a way as to establish a dense range of feature model sizes.

Feature Model	Features	Constraints	Configuration	ASs
SortingLine	39	11	00012	29
PPU	52	15	00006	44
Violet	101	27	00033	3
uClibc	313	56	00019	182
E-Shop	326	21	00042	51
WaterlooGenerated	580	61	00270	120
Busybox_1.18.0	854	123	00102	57
XSEngine	1273	886	00167	228
uClibc-Distribution	1580	197	01337	55
PROFilE-ERP-System	1920	59044	10001	223
PROFilE-E-Agribusiness	2238	0	34819	148
Automotive01	2513	2833	02017	1366

Table 5.2: Evaluation models for configurations. "ASs" is the number of automatically selected or automatically unselected features.

Next, Table 5.2 lists the models used for finding explanations for configurations. For all of the feature models already mentioned, all configurations covering all pairwise feature interactions [CKMRM03, KWG04] are generated using the IncLing algorithm [AHKT+16]. This is currently the fastest algorithm for doing so, thereby allowing to analyze configurations for larger feature models as well. In addition, this set includes two feature models and their real-world configurations, which have been used in prior studies [PMK+16]. However, because analyzing all configurations would take too long, only one configuration is tested per feature model. The configurations are chosen arbitrarily and randomly in case the order of the configurations contains biases. For each chosen configuration, all automatic configuration propagations are detected and explained.

Unfortunately, no code annotated with preprocessor directives is analyzed in this evaluation. This is because the code annotated with preprocessor annotations used to implement software product lines available online is typically configured using build systems. For instance, Linux does not use FeatureIDE feature models but KConfig models [TLSSP11]. Transforming these would require normalizing them first, in turn requiring specialized tool support. Developing such a tool, however, is out of the scope of this thesis.

5.3.1 Performance

Regarding performance, while the explanation algorithm is deterministic (as long as the used minimal unsatisfiable subset extractor is deterministic), in practice, the duration of its execution naturally varies depending on nondeterministic effects encountered on any modern computing device, e.g., multithreading and process scheduling.

As such, to perform accurate performance measurements, a few technical details have to be considered.

The first means taken to ensure accurate measurements in all of the following performance tests is that of a warm-up phase before each test. During the warm-up phase, the test is executed but the measurements are discarded. This is to mitigate effects that might skew the results due to program branches being executed for the first time, such as the Java virtual machine loading classes and doing just-in-time compilation in the background. Furthermore, the warm-up phase is prolonged to at least 5 seconds by running the test repeatedly in case the processor is not yet running at full power after just a single iteration of a quick test case.

After the warm-up phase, the actual test is executed and the measurements are stored. However, processor load peaks caused by activating background processes might still interfere with an accurate measurement of any single data point. Therefore, to minimize the average error, each test is repeated 10 times and the average is taken. All measurements are taken on an AMD FX-6300 CPU at 3.5 GHz.

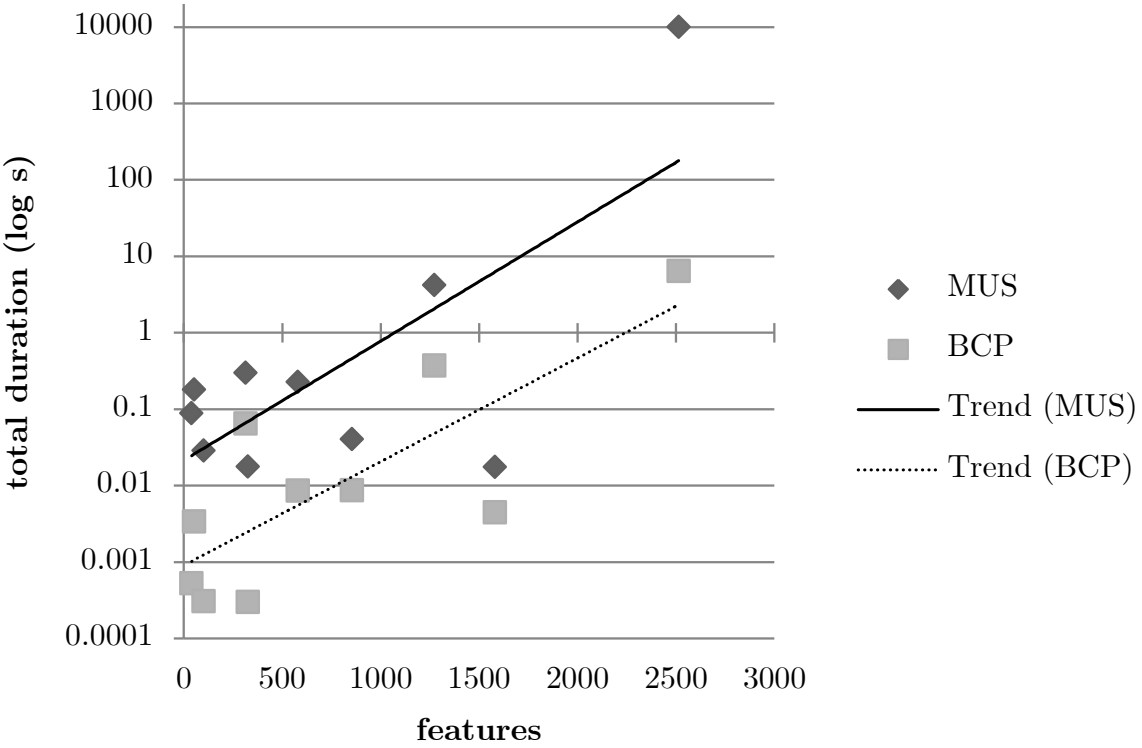


Figure 5.1: Computation time of all explanations for feature model defects.

Figure 5.1 shows the results of the performance tests for explaining the feature model defects in the models listed in Table 5.1. For each feature model, the time it takes to find the explanations for all feature model defect is measured. The measurements are taken for both the approach based on Boolean constraint propagation [KAT16] and the approach based on minimal unsatisfiable subset extractors as presented in this thesis.

For both approaches, unsurprisingly, the total explanation time depends largely on the number of defects to explain and is exponential in the number of features. In

the worst case of the largest feature model, finding all explanations using minimal unsatisfiable subset extractors takes almost 3 hours. Because nobody wants to wait that long and because the user is probably just interested in one defect at a time anyway, it is important to only find explanations when necessary instead of all of them in advance. This is done in the implementation of this approach as the explanation algorithm is only started when a defect element is selected (cf. Section 4.5.1). This way, the factor that affects the user experience more is the duration it takes to find an explanation for a single defect.

Hence, the duration of finding an explanation for a single defect is measured as well. This is shown in Figure 5.2 with the shortest duration to find an explanation, the longest one, and the quartiles between. For both approaches, the duration to find an explanation scales well with the feature model size. Except for minimal unsatisfiable subset extractors with the largest feature model tested, explaining a single defect takes a reasonable amount of time.

Still, comparing the timings of the two approaches immediately reveals that Boolean constraint propagation is magnitudes faster than using minimal unsatisfiable subset extractors. This is especially true considering that Ananieva's approach using Boolean constraint propagation continues running after finding the first explanation to possibly find a shorter one. The slow-down factor of the latter compared to the former is 4 in the best case (for the second largest model) and 1565 in the worst case (for the largest model). The performance difference is so substantial that, for some feature models, finding a single explanation using minimal unsatisfiable subset extractors can take longer than finding an explanation for every defect using Boolean constraint propagation.

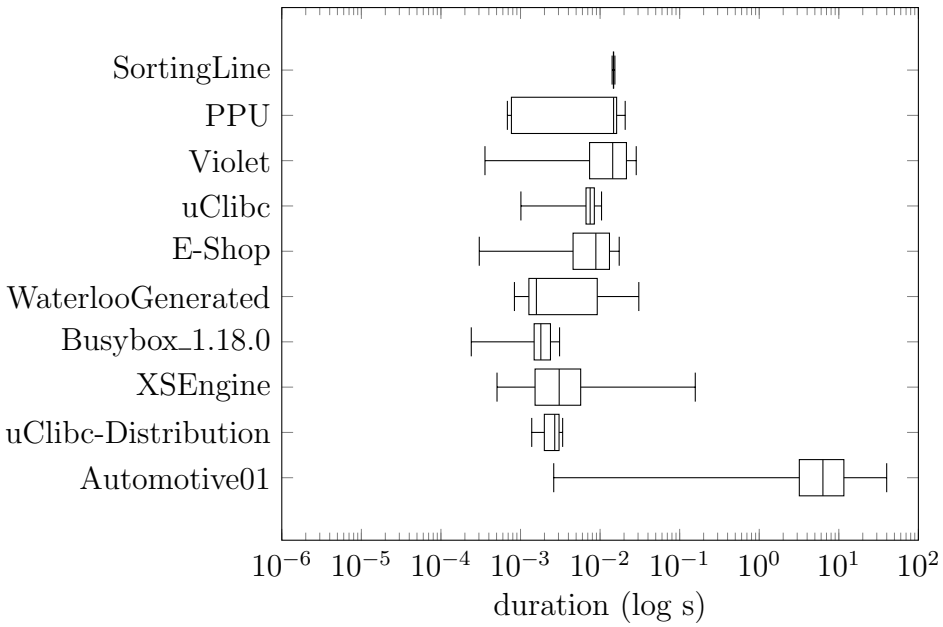
The performance of finding explanations for configurations is similar. The timings of finding explanations for all automatic selections in configurations are visualized in Figure 5.3. Here, too, do the approaches scale well with the input sizes.

Figure 5.4 also confirms this. It shows the ranges of the individual timings of finding explanations for all automatic selections in configurations. Once more, using minimal unsatisfiable subset extractors is magnitudes slower than using Boolean constraint propagation.

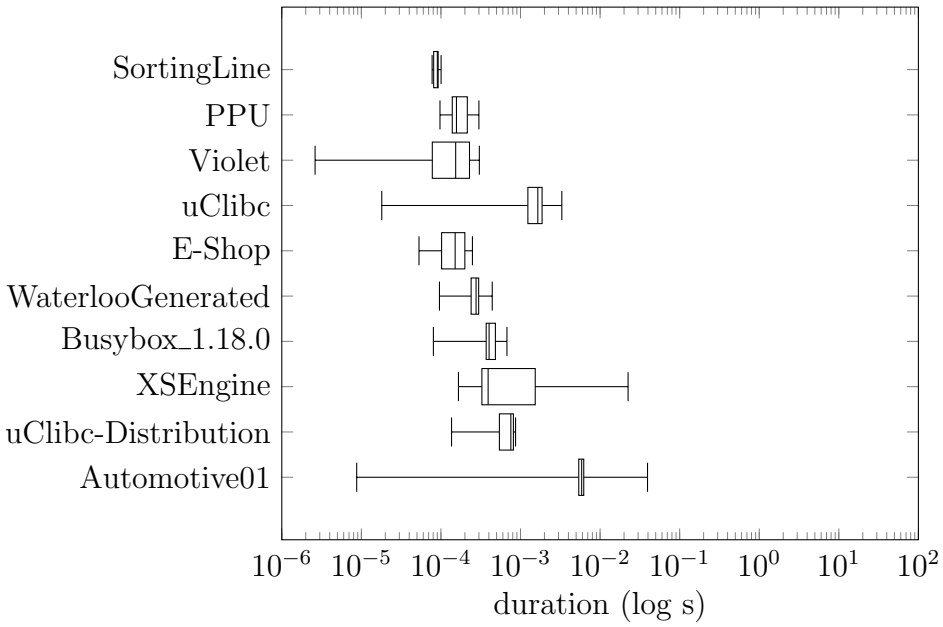
5.3.2 Explanation Lengths

Next, the lengths of the found explanations are examined. This is important because an explanation becomes more difficult to comprehend as it gets larger. Here, the length is measured by the number of reasons it contains. Each such reason is one of the elements that the explanation could possibly reference in the model, such as the structural relationship between two features in a feature model. In the case of explanations based on minimal unsatisfiable subsets, the length of the explanation is the cardinality of the minimal unsatisfiable subset.

As with the performance tests, all of the feature model defects from Table 5.1 are explained. Box plots of the lengths of all the explanations found are shown in Figure 5.5. For each feature model, this details the shortest explanation found, the longest explanation found, and the quartiles between these extremes.



(a) Computation time using minimal unsatisfiable subset extractors.



(b) Computation time using Boolean constraint propagation.

Figure 5.2: Computation time of individual explanations for feature model defects.

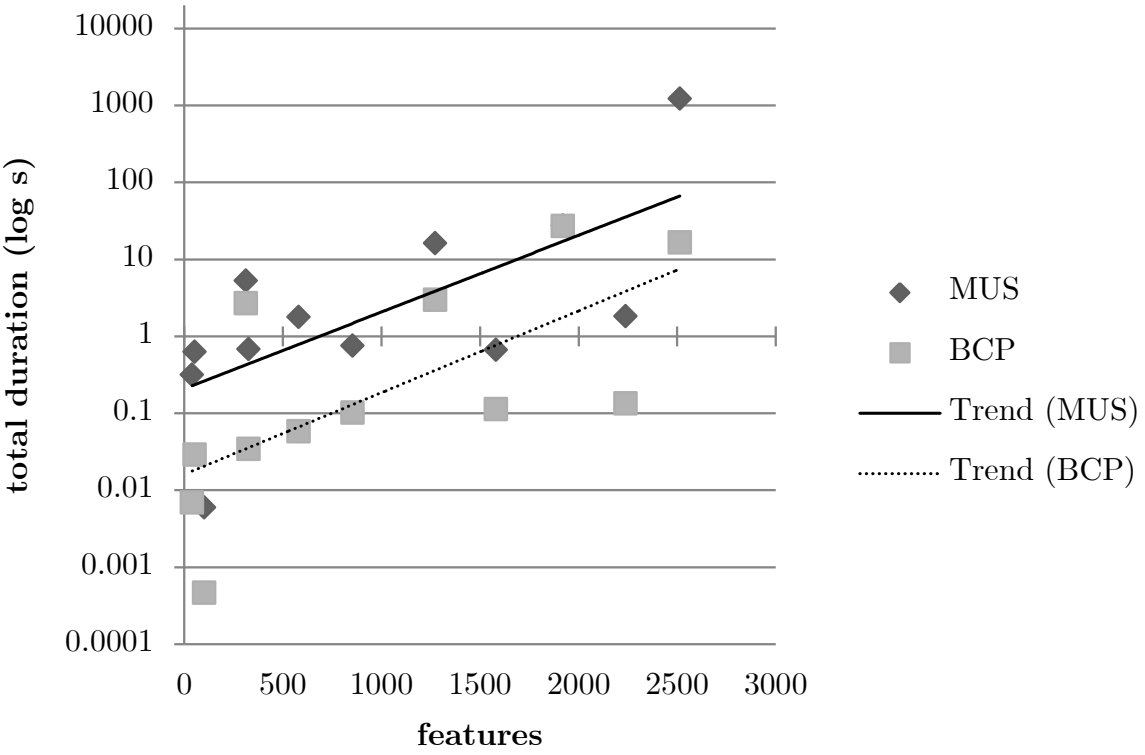


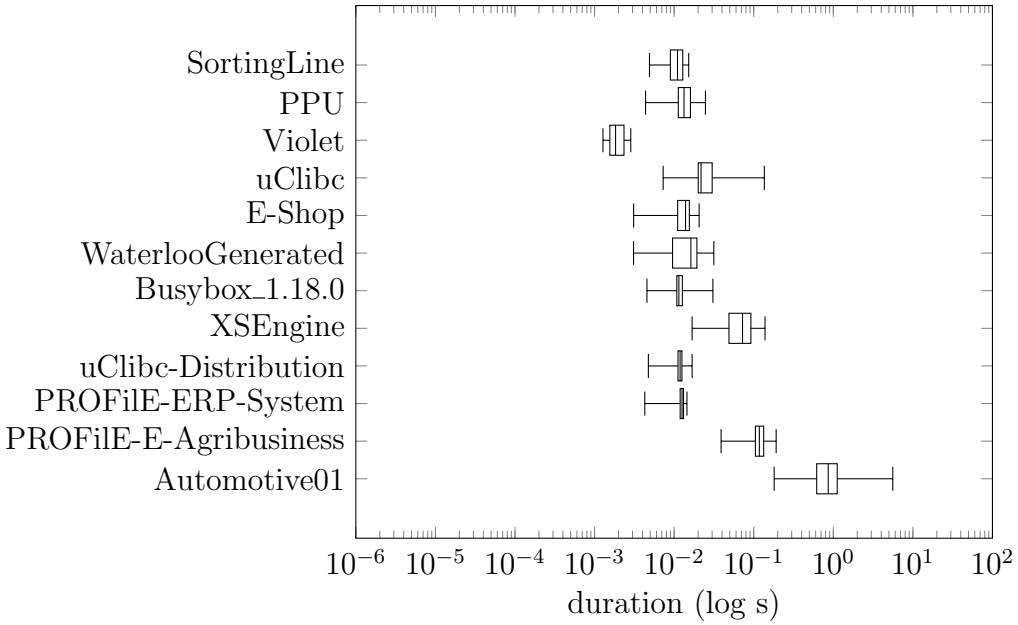
Figure 5.3: Computation time of all explanations for configurations.

For all but the smallest two feature models, the shortest explanation is as short as possible, consisting of only a single reason. The other end of the spectrum, the longest explanation, which consists of 51 reasons in total, is found in the largest feature model. This shows that the likelihood of the occurrence of very trivial defects (such as the existence of two logically equal constraints) and very complex defects increases with the size of the feature model and the number of defects it contains.

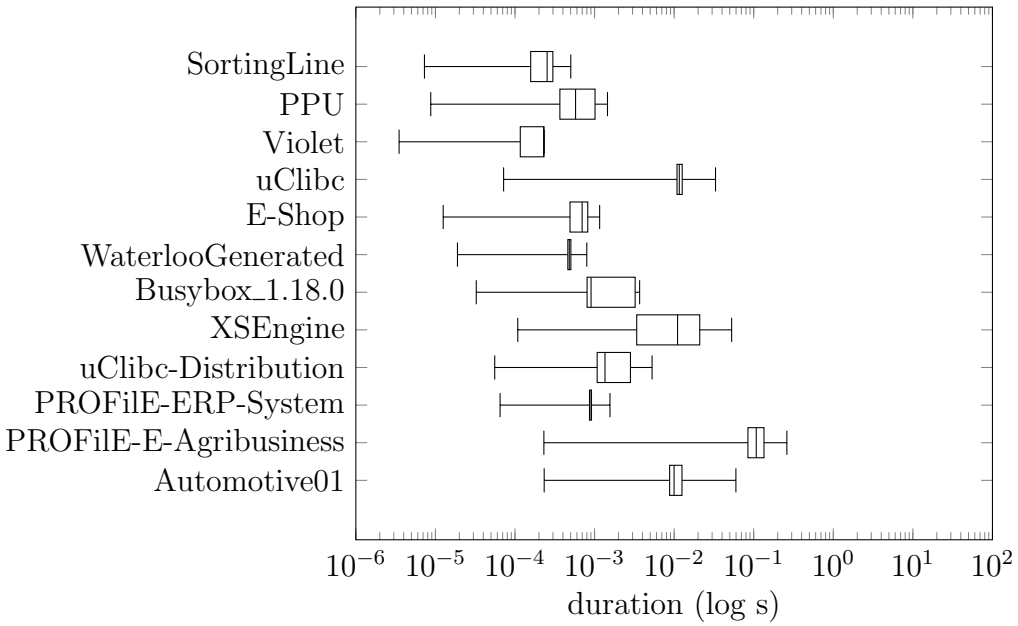
Since this arguably says more about the feature model than the approach used to find the explanations, the median length of the found explanations is probably a better indicator of the compactness of the explanations. Comparing the median length of the explanations found by the two approaches reveals that neither consistently outperforms the other. While in some cases one approach or the other finds slightly shorter explanations on average, which one that is depends on the feature model. All in all, the explanation lengths are fairly similar.

Fortunately, for both approaches, the average length is relatively small. Especially for the largest feature model, an explanation with a median length of below 10 reasons is definitely short enough to be useful. This shows that the approach presented in this thesis, like the approach based on Boolean constraint propagation, scales well with the size of the feature model in terms of the length of the explanations it finds.

The same can be seen with the explanation lengths of explanations for configurations shown in Figure 5.6. The lengths between the results of the two algorithms do not differ much. Apart from a few very large outliars in the largest feature model, the vast majority of explanations have a reasonable length. Given how long explanations can get in large feature models, it is noteworthy that most of them are short enough to be understood easily.

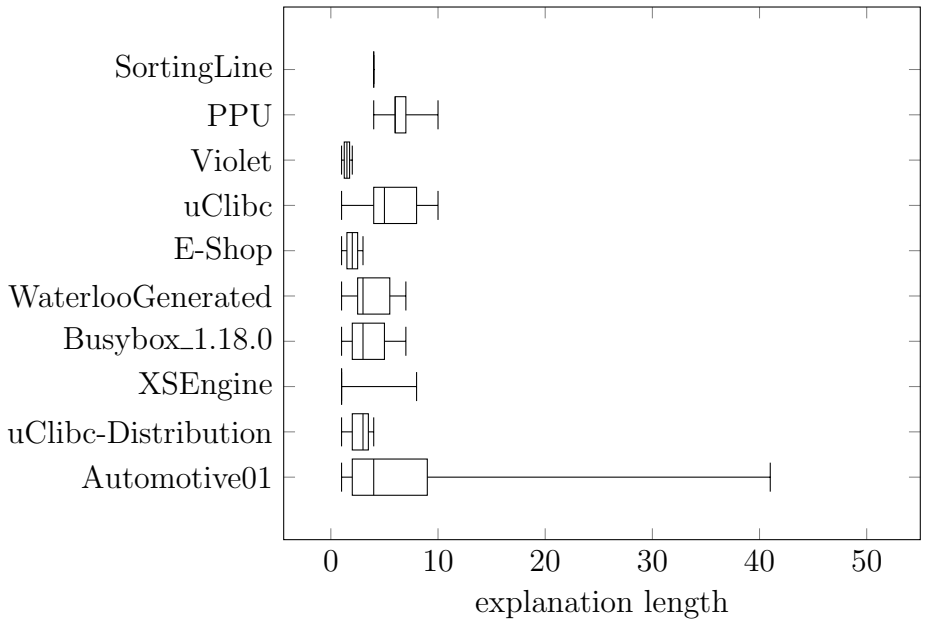


(a) Computation time using minimal unsatisfiable subset extractors.

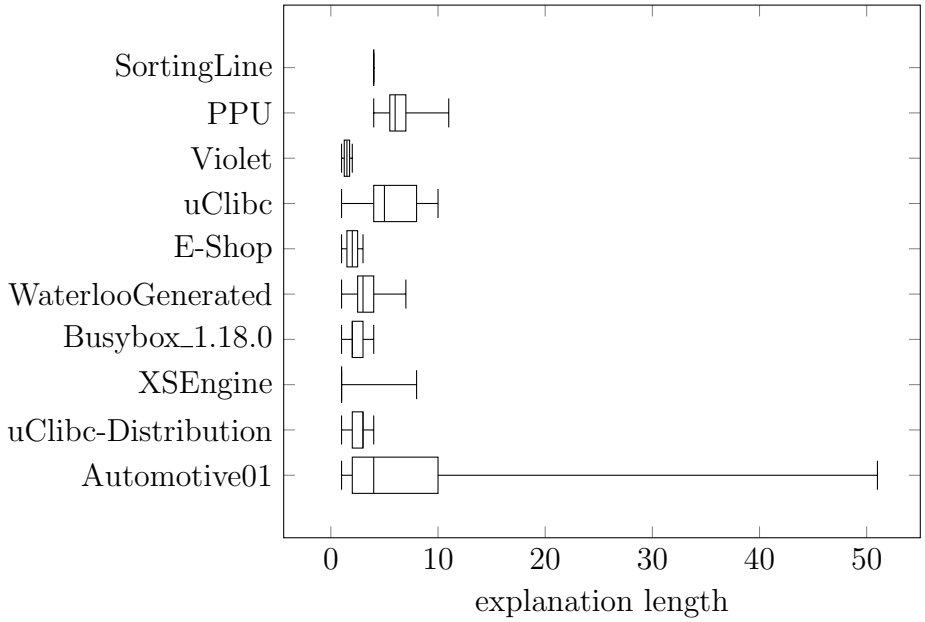


(b) Computation time using Boolean constraint propagation.

Figure 5.4: Computation time of individual explanations for configurations.

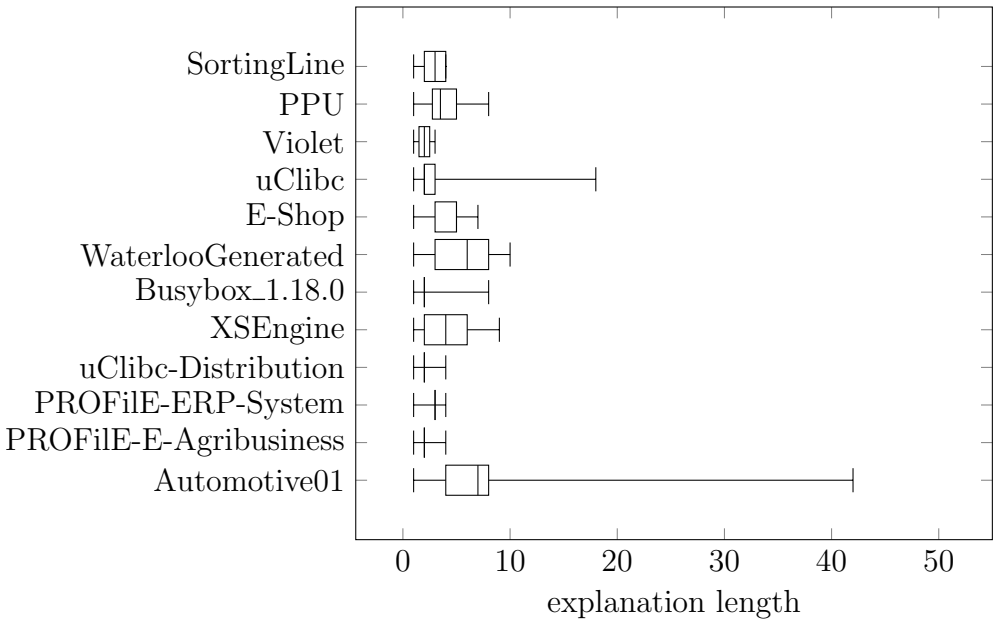


(a) Lengths using minimal unsatisfiable subset extractors.

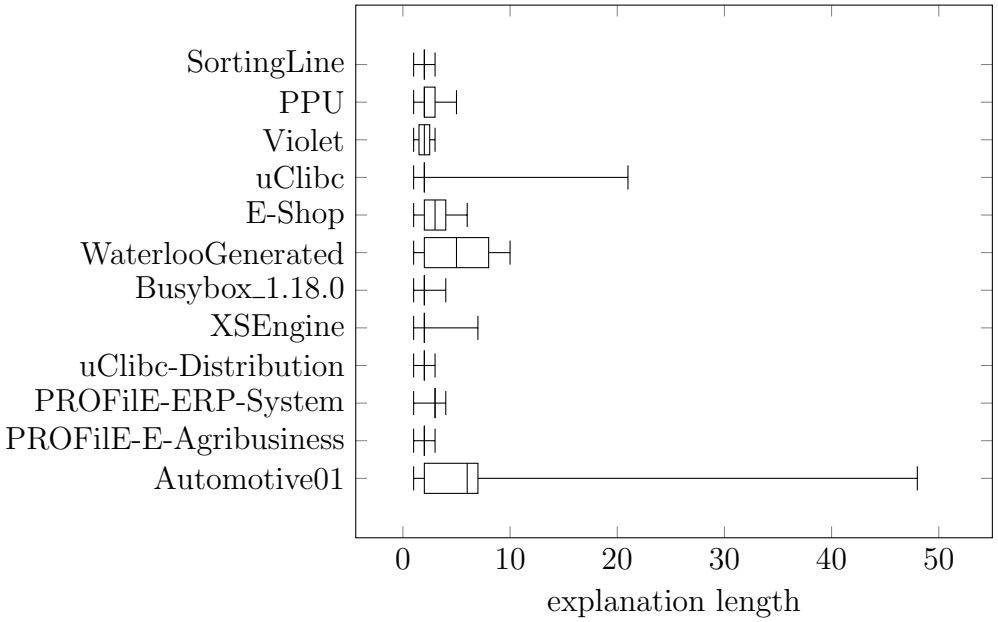


(b) Lengths using Boolean constraint propagation.

Figure 5.5: Lengths of explanations for feature model defects.



(a) Lengths using minimal unsatisfiable subset extractors.



(b) Lengths using Boolean constraint propagation.

Figure 5.6: Lengths of explanations for configurations.

5.3.3 Explanation Availability

Moving on to the last of the criteria evaluated in this thesis, this section deals with the availability of explanations. As mentioned before in Section 3.1.1, Boolean constraint propagation is incomplete, meaning it sometimes fails to find an explanation at all. To solve this, the approach presented in this thesis uses minimal unsatisfiable subset extractors instead. Thus, in this section, the availability of the explanations found using the two approaches is compared.

Feature Model	DFs		FOFs		RCs		No Expl.
	BCP	MUS	BCP	MUS	BCP	MUS	BCP
SortingLine	0	0	0	0	6	6	0
PPU	5	5	4	4	10	10	0
Violet	0	0	1	1	1	1	0
uClibc	31	31	10	10	0	0	0
E-Shop	0	0	1	1	1	1	0
WaterlooGenerated	10	10	13	13	8	8	0
Busybox_1.18.0	12	18	3	3	0	0	6
XSEngine	40	42	166	166	51	51	2
uClibc-Distribution	1	1	6	6	0	0	0
Automotive01	191	195	77	80	869	869	7
Total	290	302	281	284	946	946	15

Table 5.3: Availability of explanations for feature model defects.

The number of explanations found for each type of feature model defect per feature model is listed in Table 5.3. Comparing these values against those of Table 5.1 shows that the approach presented in this thesis manages to find an explanation for every defect it is given as input. In other words, explaining using minimal unsatisfiable subset extractors is complete as long as the underlying extractor is complete.

By contrast, the approach based on Boolean constraint propagation has trouble finding an explanation for every defect. In particular, of the 10 feature models analyzed, 3 contain defects that cannot be explained using Boolean constraint propagation. Those defects are mainly dead features, though the automotive feature model also contains 3 false-optional features for which Boolean constraint propagation returns no result. The percentage of the defects that cannot be explained using Boolean constraint propagation is 29% in Busybox, 0.8% in XSEngine and 0.6% in the automotive feature model. The many unexplainable dead features in Busybox, for example, are cases of literal incompleteness (cf. Section 3.1.1).

For configurations, the availability of explanations is similar, as can be seen in Table 5.4. Using the approach presented in this thesis, all automatic selections can be explained. As for Boolean constraint propagation, it once more fails to find an explanation for every automatic selection. The incompleteness reveals itself only in the feature models that contain feature model defects that already cannot be explained using Boolean constraint propagation. The number of circumstances that cannot be explained using Boolean constraint propagation remain the same compared to those for feature model defects. An exception is the automotive feature

Feature Model	ASs		No Expl.
	BCP	MUS	BCP
SortingLine	29	29	0
PPU	44	44	0
Violet	3	3	0
uClibc	182	182	0
E-Shop	51	51	0
WaterlooGenerated	120	120	0
Busybox_1.18.0	51	57	6
XSEngine	226	228	2
uClibc-Distribution	55	55	0
PROFilE-ERP-System	223	223	0
PROFilE-E-Agribusiness	148	148	0
Automotive01	1364	1366	2
Total	2496	2506	10

Table 5.4: Availability of explanations for configurations.

model, which contains only 2 instead of 7 unexplainable circumstances. After all, compared to feature model defects, the satisfiability queries involving configurations only add unit-clauses. These unit-clauses can always be propagated into unless they are satisfied or contradicted already and hence make it more likely that an explanation is found.

5.4 Conclusion

In conclusion, the approach presented in this thesis allows finding explanations with many positive criteria. The explanations it finds are correct and sensible as well as short enough to be helpful. This holds true not only for small models but also large ones. Its main advantage over the existing explanation approach using Boolean constraint propagation [KAT16], which is incomplete, is that it is complete and therefore finds explanations for any circumstance it is given as input. As the analysis shows, circumstances for which Boolean constraint propagation fails to find an explanation do in fact occur in real models but are very seldom.

Unfortunately, the completeness of this approach comes at a cost. The performance of this approach is several orders of magnitudes worse than that of the approach based on Boolean constraint propagation. For this reason, it is inadvisable to use minimal unsatisfiable subset extractors in cases where Boolean constraint propagation does the job. However, it is typically not known whether a given circumstance can be explained using Boolean constraint propagation before actually attempting to do so.

Therefore, given both approaches, the optimal solution is to combine them to get the best of both worlds. This can be done using a two-step algorithm. First, an explanation is attempted to be found using Boolean constraint propagation. This is fairly fast but might fail to return an explanation. If it succeeds in finding an explanation, the explanation can be returned immediately. However, if it fails to find

an explanation, an explanation is (definitely) found using a minimal unsatisfiable subset extractor. On average, the performance overhead for sometimes running two explanation approaches is offset by usually not running the costly minimal unsatisfiable subset extraction.

Indeed, as part of this thesis, this combined approach is implemented in response to this evaluation. It is implemented as another set of subclasses of `ExplanationCreator` and `ExplanationCreatorFactory` (cf. Section 4.2) in the package `composite` next to the packages `ltms` and `mus` for every use case. It is basically a decorator [GHJV95] that delegates to any number of decorated explanation creators. When explaining, it uses the decorated explanation creators in their specified order until one successfully finds an explanation. By default, the abstract factory `CompositeExplanationCreatorFactory` creates such instances of `CompositeExplanationCreator` composing first an explanation creator using Boolean constraint propagation and second one using a minimal unsatisfiable subset extractor. Due to conforming to the abstract factory pattern [GHJV95], it is plugged into the existing application seamlessly.

The effect of the combined algorithm is as expected. Often, the explanation is quickly found using Boolean constraint propagation right away, causing no slowdown. For models containing some circumstances that cannot be explained using Boolean constraint propagation, the combined algorithm is, averaged across all explanations, slower than Boolean constraint propagation but much faster than using minimal unsatisfiable subset extractors all the time. The slow-down factor of finding explanations for all feature model defects using the combined approach compared to Boolean constraint propagation is 2.96 (instead of 4.66 using minimal unsatisfiable subset extractors) for Busybox, 1.01 (instead of 11.3) for XSEngine, and 1.06 (instead of 1565) for the automotive feature model. This is a significant performance improvement over using minimal unsatisfiable subset extractors all the time. Most importantly, the combined algorithm always finds an explanation.

6. Related Work

This chapter discusses work similar to this thesis. This serves as an overview of the research field to point the reader in the right direction for further reading. In addition, the commonalities and differences are highlighted for a better understanding of the contributions of this thesis.

Feature Model Defects

Much of the automated analysis of software product lines [TAK+14] leverages a connection between the model to be analyzed and the satisfiability problem. Most notably, feature models can be transformed to propositional formulas, which enables a way of reasoning over feature models [Man02, Men09] that can be used to analyze a wide variety of properties of feature models and their elements [BSRC10]. Of particular relevance to this thesis is the analysis of feature model defects [vdML04], especially regarding dead features [BSRC10, SKT+16], false-optional features, redundant constraints [BSRC10], implicit constraints, and void feature models [BSRC10, Hem08, SKT+16]. However, these works on analyzing feature model defects only involve the detection of defects instead of also providing means for solving them. This is unfortunate because having to figure out how to solve the defect takes time away from other tasks of the engineering process. With the explanation approach presented in this thesis, the developer is given explanations that help understand and thus solve the issue. This is especially relevant for large feature models, where many elements of the feature model interact in complex ways.

As mentioned in Section 2.1.1, Boolean constraint propagation lies at the heart of most SAT solvers developed before the rise of conflict-driven clause learning engines [CESS08]. Ananieva [Ana16, KAT16] employs a logical truth maintenance system with Boolean constraint propagation at its core to find explanations for defects in feature models. First, the various defects are formulated as a satisfiability query just like in Section 3.2.1. If the formula is unsatisfiable, Boolean constraint propagation [McA90] is applied, i.e., unit-open clauses are used to derive truth value assignments one variable at a time. During propagation, the unit-open clauses involved in making the formula unsatisfiable are recorded. The set of all these propagated

unit-open clauses is a minimal unsatisfiable subset and serves as the explanation for the unsatisfiability and therefore the defect. Unfortunately, as was shown in this thesis in Section 3.1.1, Ananieva's approach using Boolean constraint propagation is incomplete, meaning it does not always find an explanation for a given defect. Specifically, of all the feature model defects tested in the evaluation part of this thesis, 1% could not be explained using Boolean constraint propagation. The approach contributed in this thesis, on the other hand, is complete and always finds a correct explanation for every recognized defect. Additionally, this thesis addresses most of the future work proposed by Ananieva. However, the evaluation in this thesis has also shown that Boolean constraint propagation is magnitudes faster than using SAT solvers to extract minimal unsatisfiable subsets. Therefore, as another contribution of this thesis, the two approaches are combined to mitigate the downsides of both. By attempting to find an explanation using Boolean constraint propagation first and a minimal unsatisfiable subset extractor second, the combined approach is as fast as Boolean constraint propagation if it finds an explanation. Otherwise, the combined algorithm still manages to find an explanation using minimal unsatisfiable subset extractors. Considering how rarely the latter case occurs and how much faster Boolean constraint propagation is, the overhead of having tried out Boolean constraint propagation before the minimal unsatisfiable subset extraction is negligible.

Mauro et al. [MNSY17] contribute the tool HyVarRec to detect and explain dead features and void feature models in context-aware feature models [MNSY16]. In context-aware feature models, features may be attributed with restrictable values. The analysis of such feature models requires quantifiers, which is why these defects are expressed in terms of satisfiability modulo theories instead of propositional formulas. The approach presented in this thesis, in contrast, deals with pure feature models. Therefore, SAT solvers suffice for the analysis. Additionally, this thesis takes more feature model defect types into account, in particular false-optional features, redundant constraints, and implicit constraints.

Wang et al. [WXH+14] propose an interactive approach for fixing feature model defects using constraint hierarchies [BFBW92]. In that approach, the application keeps suggesting the developer one of the many concrete solutions to the defect by removing a low-priority constraint from the over-constrained feature model. While doing so, each decision of the developer to accept or refuse a given solution is taken into account for further suggestions. Thus, the constraint hierarchy gradually adapts to the developer's confidence on the constraints, leading to more desirable suggestions. By contrast, the approach presented in this thesis does not attempt to figure out the thoughts of the developer. Instead, it finds explanations that narrow down the number of elements that the developer needs to consider for removal or adoption. With the knowledge provided by the explanation, the developer is then free to realize the optimal solution autonomously.

Compared to all of those approaches mentioned so far that find explanations, another advantage of the approach contributed in this thesis is the visualization of explanations for feature model defects. Whereas the explanations found by the previous approaches are displayed textually, often not even in natural language, this approach provides a more intuitive representation of the explanation. To this end,

the explanation is represented visually by highlighting relevant graphical elements of the feature diagram. This is implemented as part of the software product line development tool FeatureIDE [MTS+17, TKB+14]. FeatureIDE offers many examples of feature models from real software projects of various sizes that are used in this thesis for a more comprehensive evaluation of the scalability than is typically the case for the works mentioned before. Additionally, in this thesis, the concept of explanations is generalized and applied to more use cases than just defects in feature models, specifically configurations and code annotated with preprocessor directives.

Analysis of Configurations

Building upon the ability to express feature models as propositional formulas, SAT solvers can also be used to analyze configurations [Jan08]. In particular, configurations may be checked for validity [TAK+14], i.e., whether the configuration conforms to the feature model and refers to a product that can actually be derived. Throughout the configuration process, this can be ensured using decision propagation [HSJ+04, KTS+17], in which case selections are propagated automatically to avoid backtracking. As with the analysis of feature model defects, the analyses of configurations in these works do not provide an explanation of the detected property. Thus, the developer might end up being confused why a certain selection or deselection occurred and why its selection status can no longer be changed. The explanation approach presented in this thesis clears up any potential confusion regarding the causes of automatic configuration propagations.

Batory [Bat05] proposes an explanation approach using a logical truth maintenance system based on Boolean constraint propagation. That approach is implemented in the tool GUIDSL. In GUIDSL, features of a feature model may be selected or deselected manually, causing automatic decision propagations for which explanations are generated using Boolean constraint propagation. Unfortunately, the explanations are formulated in the problem space, i.e., they are formulas. Conversely, the approach contributed in this thesis finds explanations that are formulated in model space, i.e., they explicitly reference elements from the feature model and the configuration. This is a much more intuitive and user-friendly representation. In addition, because Batory's approach is based on Boolean constraint propagation, it is incomplete, meaning it occasionally fails to find an explanation at all. In particular, it does not manage to find explanations for the two classes of feature model defects detailed in Section 3.1.1, whereas the approach presented in this thesis does. On top of that, the approach contributed in this thesis provides explanations not only during the configuration phase but also earlier during the feature modeling phase.

Trinidad et al. [TBD+08] contribute the tool FAMA, which detects and explains feature model defects and invalid configurations. This is done using a solver for constraint satisfiability problems. However, this is not applied to redundant constraints. In contrast, the approach presented in this thesis uses a SAT solver and explains redundant constraints in addition to the other defects. Additionally, in FAMA, the explanations are not written in natural language but a very abbreviated text form. Finally, FAMA is not evaluated against feature models as large as those used in the evaluation of this thesis.

Analysis of Preprocessor Directives

The satisfiability problem can also be applied to code annotated with preprocessor directives [TAK+14]. For the implementation of a software product line, preprocessor directives contain expressions in the form of propositional formulas over features from the feature model. This makes possible what is here called invariant presence conditions, that is presence conditions that always evaluate to true, rendering the annotation superfluous, or presence conditions that always evaluate to false, thereby causing dead code blocks [LvRK+13]. Tartler et al. apply this analysis to the Linux kernel [TLD+11, TLSSP11, TSSPL09]. However, these works on analyzing code annotated with preprocessor conditions only provide algorithms for detecting invariant presence conditions but not for explaining why they occur. Yet, such explanations are especially useful in this use case because of the complex interactions between the various nested code blocks. When combined with the complexity of the feature model, figuring out the cause of defects can easily become overwhelming. With the approach presented in this thesis, it is possible to find explanations that allow the developer to easily track down the cause of invariant presence conditions. To the author's knowledge, this is the first time an algorithm for finding explanations in code annotated with preprocessor directives is conceptualized and respective tool support is provided.

Minimal Unsatisfiable Subsets

A minimal unsatisfiable subset is subset of a propositional formula for which no satisfying truth value assignment exists and which cannot be reduced further without becoming satisfiable. By extracting minimal unsatisfiable subsets [BLMS12, dlBSW03] from formulas, explanations of their unsatisfiability can be found [GMP08a]. The approach presented in this thesis applies this concept to satisfiability queries in the context of software product lines. For each clause of the found minimal unsatisfiable subset, the corresponding meaning in model space is remembered, e.g., that a feature is manually selected in a configuration or that a feature is a child of another feature.

However, the aforementioned explanation approaches use a variety of problem representations such as propositional formulas [Bat05, KAT16], constraint satisfiability problems [TBD+08], and satisfiability modulo theories [MNSY17]. Typically, minimal unsatisfiable subsets are extracted from propositional formulas, but minimal unsatisfiable subsets can also be extracted in these other cases. For example, Grégoire et al. [GMP08b] extract minimal unsatisfiable subsets from constraint satisfiability problems. Similarly, Guthmann et al. [GST16] extract minimal unsatisfiable subsets using several satisfiability modulo theory solvers. This means that the explanation approach presented in this thesis can be applied to the aforementioned approaches for finding explanations.

Liffiton et al. [LS08] propose CAMUS, an algorithm for computing all minimal unsatisfiable subsets of a formula rather than just one. This is done by first finding a minimal correction subset and then deducing the minimal unsatisfiable subsets from them. Finding all minimal unsatisfiable subsets is interesting because it enables identifying particularly critical clauses as well as using the smallest minimal

unsatisfiable subset for the explanation. The SAT solver library Sat4J [LBP10], which is used in the implementation of the approach presented in this thesis, provides an implementation of the algorithm of Liffiton et al. [LS08].a However, finding all minimal unsatisfiable subset takes significantly longer than finding any minimal unsatisfiable subset. This is because finding a minimal correction subset is an expensive operation, which is addressed in follow-up work by Liffiton et al. [LM13] with the algorithm MARCO. Due to the performance issues of extracting all minimal unsatisfiable subsets, the implementation of the approach contributed in this thesis uses only a single minimal unsatisfiable subset to create the explanation.

7. Conclusion

The first conclusion to draw from this thesis is that Ananieva's explanation approach based on Boolean constraint propagation [Ana16, KAT16] is incomplete. This means that it sometimes fails to produce any explanations for a defect. In this thesis, the incompleteness has been linked to what Forbus and de Kleer [FdK93] call refutation incompleteness and literal incompleteness. A quantitative analysis showed that these cases do in fact occur in feature models of real software projects, albeit only 1% of the feature model defects tested could not be explained using Boolean constraint propagation.

Hence, to be able to find explanations for any satisfiability query in a software product line context, another explanation algorithm using minimal unsatisfiable subsets was devised in this thesis. This algorithm uses SAT solvers to extract minimal unsatisfiable subsets from the propositional formulas expressing the circumstances to explain. Each resulting clause is traced back to model space for a more intuitive representation such as natural language or, for explanations for feature model defects, highlighted elements in the feature diagram. The algorithm is easily extensible and was applied to explain various circumstances in feature models, configurations, and code annotated with preprocessor directives. As far as the author is aware, this is the first time explanations are found automatically for code annotated with preprocessor directives.

As the evaluation showed, the algorithm presented in this thesis is both sound and complete and typically finds explanations of reasonable length. Unfortunately, compared to Boolean constraint propagation, it is slower by orders of magnitudes. To solve the performance issue, the two algorithms were combined into one. This combined algorithm attempts to find an explanation using Boolean constraint propagation first and only falls back to minimal unsatisfiable subset extractors if Boolean constraint propagation fails. As a result, the combined algorithm is as fast as Boolean constraint propagation if it finds an explanation. In the rare case that Boolean constraint propagation fails to find an explanation, the combined approach is still complete through the use of minimal unsatisfiable subset extractors.

8. Future Work

The explanation approach presented in this thesis entails several possibilities for future work. This chapter provides a few examples of them.

Using a Different Solver to Extract Minimal Unsatisfiable Subsets

Extracting minimal unsatisfiable subsets using the SAT solver from the Sat4J framework takes significantly longer than running Boolean constraint propagation. However, it is unclear whether this is an inherent issue of SAT solvers in general or only the implementation provided by Sat4J. Since it is possible that other SAT solvers are faster or provide smaller unsatisfiable subsets, another SAT solver implementation could be added to the SAT solver facade implemented as part of this thesis. Instead of SAT solvers, the minimal unsatisfiable subset extraction could also be done using other solver types such as those for satisfiability modulo theories. This could then be evaluated regarding criteria such as performance and the size of the returned minimal unsatisfiable subsets.

Extracting a Minimum Unsatisfiable Subset

To explain a circumstance, the approach presented in this thesis uses whichever minimal unsatisfiable subset of the satisfiability query is returned by the SAT solver. The minimal unsatisfiable subset extractor provided by Sat4J does not promise that the returned minimal unsatisfiable subset is actually the smallest one possible, i.e., the minimum unsatisfiable subset. Extracting the minimum unsatisfiable subset instead of just any minimal unsatisfiable subset would ensure that the explanation is as short as possible and therefore the most easily comprehensible. Given that extracting any minimal unsatisfiable subset is already a costly operation, this should include an evaluation with a strong focus on performance.

Extracting Multiple Minimal Unsatisfiable Subsets

Ananieva's explanation approach based on Boolean constraint propagation [KAT16] finds multiple explanations and returns the shortest one found. The other explanations are not simply discarded but each reason is used to give a confidence hint based

on how often it occurs across all found explanations. The idea behind this is that an explanation part of every explanation is more likely to be the root cause of the problem. It could be investigated whether such a confidence hint could also be given in a performant manner using the approach presented in this thesis by extracting multiple minimal unsatisfiable subsets instead of just one. For example, Liffiton et al. [LM13] propose the algorithm MARCO with the purpose of enumerating multiple minimal unsatisfiable subsets quickly.

Reusing the Internal Solver State

In this thesis, detecting a circumstance is separate from explaining it. This is a deliberate decision to increase the separation of concerns, to enable the independent optimization of the two tasks, and to make finding explanations optional, thereby typically avoiding having to find many explanations that end up being unused. However, for cases in which most or even all of the defects in a model need to be explained, merging the two tasks might bring a performance benefit. In particular, by reusing the state of the solver used to detect the defect, the explanation algorithm might run faster than if it did not.

Still, care must be taken not to simply reuse all of the solver's internal propagations without question. After all, to decide the satisfiability query, the solver might have branched incorrectly in trying to arrive to the conclusion. In such a case, the subset of the clauses deduced to have a certain truth value might not actually be a minimal unsatisfiable subset, which is critical to having a sensible explanation. Having to minimize the set might in turn nullify the performance advantages gained by reusing the solver state in the first place.

Visual Explanations for Configurations and Preprocessor Directives

Currently, the explanations for configurations and code with preprocessor directives are displayed as text in natural language inside tooltips. This can be made more intuitive using a visual representation of the explanation. These could look similar to the visual explanations for feature model defects. They could even reuse the visual explanations for feature model defects by for example opening the relevant parts of the feature diagram when an explanation is visualized.

Collapsing Parts of Configurations and Preprocessor Directives Irrelevant to Explanation

Parts of the feature diagram that are irrelevant to the visual explanations for feature model defects can be collapsed in a single operation. There could be an analogous operation for collapsing irrelevant parts of the configuration and for hiding irrelevant parts of the code annotated with preprocessor directives.

Quantifying the Usefulness of Explanations

A user study or experiment could be conducted to measure how useful explanations are in practice. For instance, it could be measured how fast test groups find solutions to defects with or without being given explanations.

Bibliography

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, October 2013. (cited on Page 1, 8, 9, 10, 11, and 12)

[AHKT+16] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. IncLing: Efficient Product-Line Testing Using Incremental Pairwise Sampling. In *Proceedings of the International Conference on Generative Programming: Concepts and Experiences (GPCE)*, pages 144–155, 2016. (cited on Page 14 and 68)

[AKTS16] Sofia Ananieva, Matthias Kowal, Thomas Thüm, and Ina Schaefer. Implicit Constraints in Partial Feature Models. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, pages 18–27, 2016. (cited on Page 42)

[ALS13] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Improving Glucose for Incremental SAT Solving with Assumptions: Application to MUS Extraction. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 309–317, 2013. (cited on Page 7 and 27)

[Ana16] Sofia Ananieva. Explaining Defects and Identifying Dependencies in Interrelated Feature Models. Master's thesis, Technische Universität Braunschweig, September 2016. (cited on Page 2, 3, 16, 23, 24, 27, 29, 37, 38, 54, 55, 62, 66, 67, 79, and 85)

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Lines Conference (SPLC)*, pages 7–20, 2005. (cited on Page 1, 2, 9, 11, 81, and 82)

[BCH15] Jan Bosch, Rafael Capilla, and Rich Hilliard. Trends in Systems and Software Variability. *IEEE Software*, 32(3):44–51, 2015. (cited on Page 1, 14, and 17)

[BFBW92] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint Hierarchies. *LISP and Symbolic Computation*, 5(3):223–270, 1992. (cited on Page 80)

[BFT17] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, University of Iowa, 2017. (cited on Page 58)

[Bie10] Matthias Biehl. Literature Study on Model Transformations. Technical Report ISRN/KTH/MMK/R-10/07-SE, Royal Institute of Technology, July 2010. (cited on Page 54)

[BLMS12] Anton Belov, Inês Lynce, and Joao P. Marques-Silva. Towards Efficient MUS Extraction. *AI Communications*, 25(2):97–116, 2012. (cited on Page 7, 27, 58, and 82)

[BRN+13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A Survey of Variability Modeling in Industrial Practice. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, 2013. (cited on Page 1 and 9)

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–636, 2010. (cited on Page 1, 10, 11, 14, and 79)

[CDS08] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, 2008. (cited on Page 14)

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Longman Publishing Co., Inc., 2000. (cited on Page 8, 9, and 12)

[CESS08] Koen Claessen, Niklas Eén, Mary Sheeran, and Niklas Sörensson. SAT-Solving in Practice. In *Proceedings of the International Workshop on Discrete Event Systems (WODES)*, pages 61–67, 2008. (cited on Page 6, 7, and 79)

[CGR+12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 173–182, 2012. (cited on Page 1 and 9)

[CKMRM03] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks*, 41(1):115–141, 2003. (cited on Page 14 and 68)

[Coo71] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC)*, pages 151–158, 1971. (cited on Page 6)

[CW07] Krzysztof Czarnecki and Andrzej Wąsowski. Feature Diagrams and Logics: There and Back Again. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 22–31, 2007. (cited on Page 27 and 54)

[dlBSW03] Maria Garcia de la Banda, Peter J. Stuckey, and Jeremy Wazny. Finding All Minimal Unsatisfiable Subsets. In *Proceedings of the International Conference on Principles and Practice of Declaritive Programming (PPDP)*, pages 32–43, 2003. (cited on Page 7, 66, and 82)

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5(7):394–397, 1962. (cited on Page 7 and 26)

[DP60] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960. (cited on Page 7 and 26)

[EMS07] Niklas Eén, Alan Mishchenko, and Niklas Sörensson. Applying Logic Synthesis for Speeding Up SAT. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 7, pages 272–286, 2007. (cited on Page 6 and 57)

[End01] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 2001. (cited on Page 5)

[ER11] Emelie Engström and Per Runeson. Software Product Line Testing — A Systematic Mapping Study. *Information and Software Technology*, 53(1):2–13, 2011. (cited on Page 1 and 14)

[ES03] Niklas Eén and Niklas Sörensson. Temporal Induction by Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003. (cited on Page 7, 27, 45, and 57)

[ES04] Niklas Eén and Niklas Sörensson. An Extensible SAT-Solver. *Lecture Notes in Computer Science*, 2919:502–518, 2004. (cited on Page 7)

[FdK93] Kenneth D. Forbus and Johan de Kleer. *Building Problem Solvers*. MIT Press, 1993. (cited on Page 23, 24, and 85)

[FLVH15] 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 IFAC Symposium on Information Control Problems in Manufacturing (INCOME)*, volume 48, pages 211–218, 2015. (cited on Page 67)

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995. (cited on Page 36, 40, 43, 48, 52, 55, 58, and 78)

[GM96] James Gosling and Henry McGilton. The Java Language Environment, May 1996. (cited on Page 36)

[GMP08a] Éric Grégoire, Bertrand Mazure, and Cédric Piette. On Approaches to Explaining Infeasibility of Sets of Boolean Clauses. In *Proceedings of the International Conference on Tools with Artificial Intelligence (ICTAI)*, volume 1, pages 74–83, 2008. (cited on Page 7, 23, and 82)

[GMP08b] Éric Grégoire, Bertrand Mazure, and Cédric Piette. On Finding Minimally Unsatisfiable Cores of CSPs. *International Journal on Artificial Intelligence Tools*, 17(4):745–763, 2008. (cited on Page 82)

[GST16] Ofer Guthmann, Ofer Strichman, and Anna Trostanetski. Minimal Unsatisfiable Core Extraction for SMT. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 57–64, 2016. (cited on Page 82)

[Gü16] Timo Günther. Visual Explanations of Defects in Feature Models. Project work, Technische Universität Braunschweig, December 2016. (cited on Page 38, 60, and 62)

[Hem08] Adithya Hemakumar. Finding Contradictions in Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 183–190, 2008. (cited on Page 79)

[HSJ+04] Tarik Hadzic, Sathiamoorthy Subbarayan, Rune M. Jensen, Henrik R. Andersen, Jesper Møller, and Henrik Hulgaard. Fast Backtrack-Free Product Configuration Using a Precompiled Solution Space Representation. In *Proceedings of the International Conference on Economic, Technical and Organizational Aspects of Product Configuration Systems (PETO)*, pages 131–138, 2004. (cited on Page 18 and 81)

[Jan08] Mikoláš Janota. Do SAT Solvers Make Good Configurators? In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 191–195, 2008. (cited on Page 16 and 81)

[JHF12] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 46–55, 2012. (cited on Page 14)

[JLBRS12] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The International SAT Solver Competitions. *AI Magazine*, 33(1):89–92, 2012. (cited on Page 6)

[Kan16] Frederik Kanning. Presence Condition Reasoning with Feature Model Interfaces. Master's thesis, Technische Universität Braunschweig, December 2016. (cited on Page 42)

[KAT16] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. Explaining Anomalies in Feature Models. In *Proceedings of the International*

Conference on Generative Programming: Concepts and Experiences (GPCE), pages 132–143, 2016. (cited on Page 1, 2, 14, 15, 22, 23, 42, 60, 65, 66, 69, 77, 79, 82, 85, and 87)

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, DTIC, November 1990. (cited on Page 1, 9, and 10)

[KLL+14] Matthias Kowal, Christoph Legat, David Lorefice, Christian Prehofer, Ina Schaefer, and Birgit Vogel-Heuser. Delta Modeling for Variant-Rich and Evolving Manufacturing Systems. In *Proceedings of the International Workshop on Modern Software Engineering Methods for Industrial Automation (MoSEMInA)*, pages 32–41, May 2014. (cited on Page 67)

[KSTS16] Sebastian Krieter, Reimar Schröter, Thomas Thüm, and Gunter Saake. An Efficient Algorithm for Feature-Model Slicing. Technical Report FIN-001-2016, University of Magdeburg, 2016. (cited on Page 42)

[KTM+17] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. Is There a Mismatch Between Real-World Feature Models and Product-Line Research? In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESECFSE)*, pages 291–302, 2017. (cited on Page 68)

[KTS+17] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. Propagating Configuration Decisions with Modal implication Graphs. 2017. (cited on Page 17, 18, and 81)

[KWG04] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, 2004. (cited on Page 14 and 68)

[LAL+10] Jorg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, volume 1, pages 105–114, 2010. (cited on Page 1 and 13)

[LBP10] Daniel Le Berre and Anne Parrain. The Sat4j Library, Release 2.2: System Description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(0):59–64, July 2010. (cited on Page 7, 55, 58, and 83)

[LM13] Mark H. Liffiton and Ammar Malik. Enumerating Infeasibility: Finding Multiple MUSes Quickly. In *Proceedings of the International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming (CPAIOR)*, pages 160–175, 2013. (cited on Page 83 and 88)

[LMS04] Inês Lynce and João P. Marques-Silva. On Computing Minimum Unsatisfiable Cores. *Lecture Notes in Computer Science*, 3542:305–310, 2004. (cited on Page 7)

[LS08] Mark H. Liffiton and Karem A. Sakallah. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008. (cited on Page 7, 82, and 83)

[LvRK+13] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable Analysis of Variable Software. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESECFSE)*, pages 81–91, 2013. (cited on Page 18 and 82)

[Man02] Mike Mannion. Using First-Order Logic for Product Line Model Validation. *Proceedings of the International Software Product Lines Conference (SPLC)*, pages 149–202, 2002. (cited on Page 1, 9, 11, and 79)

[McA90] David A. McAllester. Truth Maintenance. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, volume 90, pages 1109–1116, 1990. (cited on Page 6 and 79)

[Men09] Marcílio Mendonça. *Efficient Reasoning Techniques for Large Scale Feature Models*. PhD thesis, University of Waterloo, 2009. (cited on Page 1, 9, and 79)

[ML97] Marc H. Meyer and Alvin P. Lehnerd. *The Power of Product Platforms*. Free Press, 1997. (cited on Page 8)

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the Design Automation Conference (DAC)*, pages 530–535, 2001. (cited on Page 7)

[MNSY16] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. Context aware reconfiguration in software product lines. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 41–48, 2016. (cited on Page 80)

[MNSY17] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. Anomaly Detection and Explanation in Context-Aware Software Product Lines. In *Proceedings of the International Systems and Software Product Line Conference (SPLC)*, 2017. (cited on Page 2, 80, and 82)

[MTS+16] Jens Meinicke, Thomas Thüm, Reimar Schröter, Sebastian Krieter, Fabian Benduhn, Gunter Saake, and Thomas Leich. FeatureIDE:

Taming the Preprocessor Wilderness. In *Proceedings of the International Conference on Software Engineering Companion (ICSE-C)*, pages 629–632, 2016. (cited on Page 13 and 36)

[MTS+17] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017. (cited on Page 3, 14, 35, 36, and 81)

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL (T). *Journal of the ACM (JACM)*, 53(6):937–977, 2006. (cited on Page 7 and 27)

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005. (cited on Page 1, 7, 8, 9, and 14)

[PKM+16] Juliana Alves Pereira, Sebastian Krieter, Jens Meinicke, Reimar Schröter, Gunter Saake, and Thomas Leich. FeatureIDE: Scalable Product Configuration of Variable Systems. In *Proceedings of the International Conference on Software Reuse (ICSR)*, pages 397–401, 2016. (cited on Page 18 and 36)

[PMK+16] Juliana Alves Pereira, Pawel Matuszyk, Sebastian Krieter, Myra Spiliopoulou, and Gunter Saake. A Feature-Based Personalized Recommender System for Product-Line Configuration. In *Proceedings of the International Conference on Generative Programming: Concepts and Experiences (GPCE)*, pages 120–131, 2016. (cited on Page 68)

[SKT+16] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *IEEE Transactions on Software Engineering*, pages 667–678, 2016. (cited on Page 1, 9, 14, and 79)

[SRG11] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. A Comparison of Decision Modeling Approaches in Product Lines. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 119–126, 2011. (cited on Page 9)

[SSSPS07] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Is The Linux Kernel a Software Product Line? In *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL)*, 2007. (cited on Page 8)

[TAK+14] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys (CSUR)*, 47(1):6:1–6:45, 2014. (cited on Page 1, 2, 8, 14, 16, 18, 79, 81, and 82)

[TBD+08] 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*, 81:883–896, 2008. (cited on Page 2, 81, and 82)

[TKB+14] 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*, 79(0):70–85, January 2014. (cited on Page 3, 10, 35, 36, and 81)

[TKES11] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. Abstract Features in Feature Modeling. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 191–200, 2011. (cited on Page 11)

[TLD+11] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. Configuration Coverage in the Analysis of Large-Scale System Software. In *Proceedings of the International Workshop on Programming Languages and Operating Systems (PLOS)*, 2011. (cited on Page 14 and 82)

[TLSSP11] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proceedings of the International EuroSys Conference (EuroSys)*, pages 47–60, 2011. (cited on Page 8, 18, 68, and 82)

[Tse68] Grigorij Samuilovich Tseitin. On the Complexity of Derivation in Propositional Calculus. *Studies in Constrained Mathematics and Mathematical Logic*, 1968. (cited on Page 6)

[TSSPL09] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Dead or Alive: Finding Zombie Features in the Linux Kernel. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, pages 81–86, 2009. (cited on Page 82)

[vdLSR07] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007. (cited on Page 1, 7, and 8)

[vdML04] Thomas von der Maßen and Horst Lichter. Deficiencies in Feature Models. In *Workshop on Software Variability Management for Product Derivation-Towards Tool Support*, 2004. (cited on Page 14 and 79)

[WXH+14] Bo Wang, Ying-Fei Xiong, Zhen-Jiang Hu, Hai-Yan Zhao, Wei Zhang, and Hong Mei. Interactive Inconsistency Fixing in Feature Modeling. *Journal of Computer Science and Technology*, 29(4):724–736, 2014. (cited on Page 80)

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

Braunschweig, den 2017-10-17