



Technische
Universität
Braunschweig



Explaining Anomalies in Feature Models

Computer Science

-

Technical Report Nr. 2016-01

Matthias Kowal, Sofia Ananieva and Thomas Thüm

August 1, 2016

Institute of Software Engineering and Automotive Informatics
at
Technische Universität Carolo-Wilhelmina zu Braunschweig (Germany)

Acknowledgment

This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future — Managed Software Evolution and by the European Commission within the project HyVar (grant agreement H2020-644298).

Abstract

The development of variable software, in general, and feature models, in particular, is an error-prone and time-consuming task. It gets increasingly more challenging with industrial-size models containing hundreds or thousands of features and constraints. Each change may lead to anomalies in the feature model such as making some features impossible to select. While the detection of anomalies is well-researched, giving explanations is still a challenge. Explanations must be as accurate and understandable as possible to support the developer in repairing the source of an error. We propose a generic algorithm for explaining different anomalies in feature models. We achieve a benefit for the developer by computing short explanations and by emphasizing specific parts in explanations that are more likely to be responsible for the anomaly. We provide an open-source implementation in FeatureIDE and show its scalability for industrial-size feature models.

Contents

Contents	1
1. Introduction	3
2. Feature Models and Anomalies	5
3. Generic Anomaly Explanation	9
3.1. Logic Truth Maintenance System	9
3.2. Boolean Constraint Propagation	10
4. Explanations for Anomalies	13
4.1. Characteristics of BCP	15
4.2. Improvements	16
5. Evaluation	19
5.1. Implementation	19
5.2. Qualitative Evaluation	20
5.3. Quantitative Evaluation	21
6. Related Work	25
7. Conclusion	27
Bibliography	29

1 Introduction

Variability has become a key characteristic to many software systems, especially considering the steadily increasing demand for highly customizable products. Customers desire products that can be tailored to their individual requirements, e.g., as in the automotive domain. Indeed, variability enables systems to adapt to all kinds of environments or customer requirements, leading ultimately to a large variant space, which has to be managed. As a result, product lines have been introduced with variability management as a key concept several decades ago [7, 22]. A product line is a set of related systems that have common and differing features. For example, each operating system may have a GUI making it a core feature, but they can have an x86 or x64 architecture. Generating a specific variant of the system can efficiently be achieved by maximizing the reuse potential with the help of these core features. Hence, product lines enable a reduced time-to-market, are more cost-efficient, and simplify the maintenance compared to classical development methods [9, 30].

Since not all combinations of features are useful, feature models are often used to express the intended variability [5, 22]. Developing and maintaining feature models gets increasingly more difficult with regard to large product lines containing thousands of features and constraints such as the Linux kernel. Uncontrolled evolution due to constantly changing requirements and dependencies between features poses a major threat to the validity of the feature model and is a tedious task for developers [28].

Evolution can introduce redundancy and inconsistencies as anomalies in a feature model [40]. A feature model contains redundancy, if semantic information is modeled in multiple ways, which is in general not preferable. Inconsistencies appear in a feature model, if semantic information is modeled contradictory resulting in a feature that can never be selected, also called dead feature. To detect those and further anomalies, adequate tool support is needed for developers when evolving feature models [5, 4]. In addition to the detection, the user needs information in terms of why an anomaly occurs in a feature model. These explanations support the correction of anomalies [5]. An explanation is a subset of all dependencies that lead to an anomaly. Explanations are most helpful if they are short and easy to understand. Hence, developers can focus on the relevant part of the feature model and quickly comprehend the cause leading to an anomaly [4].

Existing work to compute explanations often includes only a subset of all anomaly types, gives explanations only in form of logic formulas or does not scale [20, 31, 3]. Hence, we propose an approach avoiding these limitations without putting additional workload on the developer or introducing new language concepts to feature models. In summary, we make the following contributions:

1. We propose a generic algorithm for the explanation of anomalies that is easy to extend.
2. We aim to compute short explanations in natural language and emphasize most relevant parts in explanations.
3. Based on our open-source tool support in FeatureIDE, we evaluate the scalability of our generic algorithm.

2 Feature Models and Anomalies

A feature model consists of a hierarchically arranged set of features and has typically a tree-like graphical representation such as depicted in Fig. 2.1. Relationships between parent and child features are expressed using the following notations and their semantics (see legend in Fig. 2.1 for graphical notation) [22, 9]:

- *Mandatory* – feature must be selected, if the parent is,
- *Optional* – feature is optional,
- *Or* – one or more subfeatures can be selected,
- *Alternative* – only one subfeature can be selected.

Fig. 2.1 shows a significantly simplified feature model of a car. A car must have a *Carbody* and a *Gearbox*. One can choose between a *Manual* or an *Automatic* gearbox. In addition, we can select a *Radio* with different subfeatures, namely *Ports*, *Navigation*, and *Bluetooth*, further extending the functionality. *Navigation* automatically includes a *GPS* system and may include maps for either *Europe* or *USA*. Music can be played via *USB*, *CD*, or both options. Relationships between features that are not directly related in the hierarchy of the feature tree are expressed using cross-tree constraints. The cross-tree constraints are usually written in textual notation with propositional logic of which six can be found in Fig. 2.1 below the feature tree. A feature model is typically translated to a satisfiability (SAT) problem to detect anomalies or to find valid product variants [3]. The feature tree itself does not produce any anomalies or inconsistencies, since it is simply not possible with the semantic at hand. However, the introduction of cross-tree constraints often leads to anomalies. Some examples are already visible in Fig. 2.1, which we now discuss in more detail.

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

Table 2.1.: Anomaly detection with a SAT solver. FM = feature model, f = feature of interest, f_{opt} = optional feature, p = parent of feature f_{opt} , c = cross-tree constraint, $FM = FM' \wedge c$, and x = propositional formula.

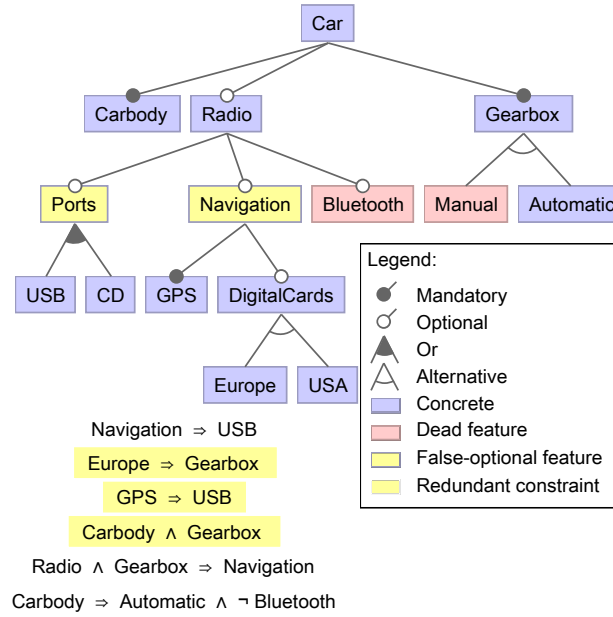


Figure 2.1.: Anomalies in a car feature model

Void Feature Models. The first anomaly that we want to tackle is probably the most severe. If the developer encounters a void feature model, it is not possible to derive any variant of the product line. For example in Fig. 2.1 adding the constraint $Carbody \wedge \neg Gearbox$ would result in a void feature model. A void model is detected by translating the feature model into a propositional formula and using a SAT solver to check for a contradiction as illustrated in Table 2.1.

Dead Features. Features are regarded as dead, if they can never be selected in any variant of the product line [40]. Hence, the feature has no effect at all. Table 2.1 shows the respective call of the SAT solver. For example, in Fig. 2.1, *Bluetooth* and *Manual* are dead features. This anomaly is problematic as software artifacts could be developed but never used.

False-Optional Features. A feature is defined as false-optional, if the selection of its parent makes the feature itself selected as well, although it is defined as optional and not mandatory. Fig. 2.1 presents two false-optional features, namely *Ports* and *Navigation*. It can be hard to determine the necessary repairs in a feature model, especially if the features may not even occur in the cross-tree constraint as the feature *Ports*. Again, the respective SAT call is presented in Table 2.1.

Redundant Constraints. A cross-tree constraint is redundant if its removal does not change the validity of configurations. For example, constraints that imply the selection of a core feature, which is always selected. This redundancy is quite easy to detect and solve, since the developer only has to remove the specific constraint. The process gets far more difficult if the redundancy is caused by the concatenation of numerous cross-tree constraints and it is not obvious, which constraint should be removed. A SAT solver checks if identical solutions are satisfiable for two feature models. One feature model contains the redundant constraint and a second model does not contain it, while they are identical in all other aspects (see Table 2.1).

Problem Statement. Feature models and their underlying propositional logic already exist for several decades. Hence, it is not surprising that an extensive amount of research is available, especially considering the analysis of feature models. Indeed, the identification of anomalies is often avail-

able in feature modeling tools such as FeatureIDE [23] or pure::variants¹. The explanation of these anomalies is often neglected or not sufficient in terms of explanation length, scalability and comprehensibility [5, 3, 25, 19, 38]. At this point, we leave it to the reader to find explanations for the anomalies shown in Fig. 2.1. It will most likely take quite some effort even for such a small feature model. In feature models with less than 50 features and even fewer cross-tree constraints, developers may be able to see the reason for redundant cross-tree constraints or anomalies in general [3]. This point gets increasingly more difficult with larger feature models. The next section tackles the theoretical part of our explanation algorithm. The anomalies in Fig. 2.1 are explained in Section 5 using our implementation in FeatureIDE.

¹<https://www.pure-systems.com/>

3 Generic Anomaly Explanation

It is our motivation to support the developer in a way that a solid decision process is possible regardless of the anomaly type leading to the following three criteria for our algorithm:

1. *Generic*: The algorithm should be *generic* to cover all anomalies mentioned in the previous section.
2. *Efficient*: The algorithm should scale to large feature models with thousands of features and constraints.
3. *Informative*: Explanations should be as short as possible and more important parts should be highlighted.

The foundations of the explanation algorithm are based on previous work by Batory and can be found in a tool called GUIDSL which is part of the AHEAD tool suite [3]. It successfully adapts basic ideas of the Boolean Constraint Propagation (BCP) algorithm to provide justifications for selected and deselected features. In GUIDSL, a user can select a specific feature and gets feedback why other features are not available anymore. However, explanations are just given in terms of propositional formulas and GUIDSL does only support the explanation of dead features. It provides the user with explanations considering the configuration possibilities of a product line. BCP on its part is a sound algorithm and used as an inference engine for a Logic Truth Maintenance System (LTMS). In the following, we explain the LTMS and BCP in more detail as foundation of our work. The necessary adaptations to use BCP for the explanation of anomalies are described in Section 4.

3.1. Logic Truth Maintenance System

Truth maintenance is a wide-spread approach in the area of artificial intelligence for implementing inference systems. The core of a truth maintenance system (TMS) is an inference engine which derives assumptions about variable values and maintains the reason for its belief. A TMS can be used to perform a range of activities such as explanation capabilities, reasoning, and propositional deduction [13]. Different types of TMSs exist in literature [32, 11, 27]. They differ in their fundamental structure, implementation, and functions. A powerful TMS is the logic truth maintenance system (LTMS), which is implemented in GUIDSL [3]. An LTMS is a boolean constraint propagation-based approach which can be used to infer an explanation. The basic principle of LTMS is depicted in Fig. 3.1. As the LTMS is based on logical constructions, we need a logical specification with (a) a set of boolean variables, (b) a propositional formula which consists of clauses constraining these variables, (c) premises which are permanent assignments of truth values to variables, and (d) a set of assumptions which are assignments of truth values to variables that may be revoked later.

We focus on the LTMS ability to compute inferences which follow assumptions and derive truth value assignments to variables. Whenever an inference is made, the LTMS stores the reason for the inference. The LTMS is able to find contradictions in the propositional formula depending on

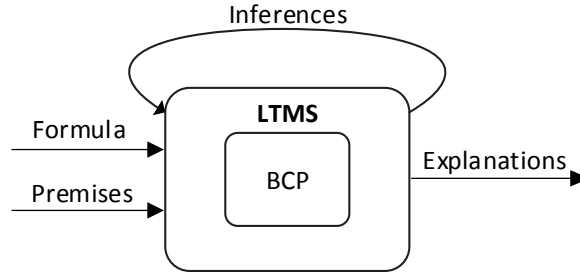


Figure 3.1.: LTMS

the given premises, e.g., the truth value of a dead feature must be *false* and by setting its value to *true*, LTMS computes a contradiction at some point. After the occurrence of this contradiction, the stored reasons can be used to generate explanations.

An automated analysis of the feature model in its logical representation as a propositional formula gives us the information, if anomalies occur. This information serves as input to the LTMS: The set of boolean variables represents features from the feature model. A propositional formula is derived from the feature tree and cross-tree constraints. Premises are based on the anomaly that we want to explain.

3.2. Boolean Constraint Propagation

As mentioned before, the core of an LTMS is its inference engine namely boolean constraint propagation (BCP) [3]. Boolean constraints are represented by means of boolean formulas and are a special case of constraint satisfaction problems [1]. They use typical connectives such as AND, OR, and NOT to combine variables. To reason about boolean constraints, rules can be applied to propagate known values for boolean variables. Two simple example rules are shown below:

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

BCP is also known as *Unit Resolution* and makes use of such rules to conclude inferences [10]. This type of inference forms the basis to increase the efficiency of a boolean constraint solver [8].

Input to a BCP is usually specified as a set of variables defined by a three-value logic (true, false, unknown) and a formula in conjunctive normal form (CNF). A CNF is a conjunction of clauses. A clause consists of a set of literals. A literal for its part is a variable or its negation. A formula is *satisfied*, if at least one literal in every clause is true. Consequently, a truth value assignment that satisfies the formula represents a product in the SPL [3].

The basic idea of a BCP is to assign a type to every clause:

- *Satisfied*: at least one literal is true
- *Violated*: all literals are false
- *Unit-Open*: one literal is unknown while the remaining literals are false
- *Non Unit-Open*: more than one literal is unknown, the rest is false

Hence, we can observe that a unit-open clause can be satisfied by setting its unknown literal to true and a violated clause is equivalent to a contradiction.

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

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

A general overview of BCP algorithm is presented in Fig. 3.2. BCP is invoked on initial assignments which propagate the consequences of premises. The selection of truth values for the premises are a crucial step and form the core of our generic algorithm, since they are responsible for the computation of a contradiction in the CNF. In its first iteration, the algorithm pushes every unit-open clause it encounters in the CNF to a stack. Iteratively, unit-open clauses are removed from the stack and BCP infers the truth value of the unknown literal. Afterwards, the CNF is searched for new unit-open clauses. Whenever the BCP algorithm detects a violation during constraint propagation, it reports the contradiction and terminates.

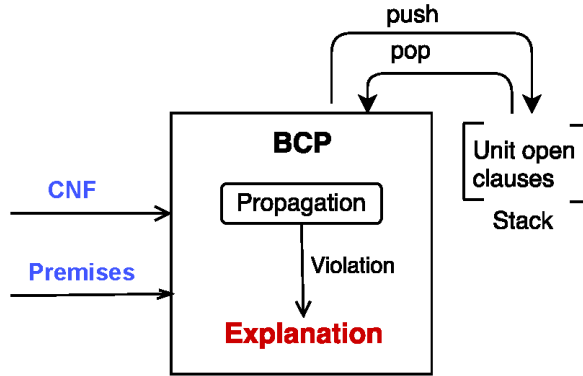


Figure 3.2.: Overview of BCP

The information storage of a BCP algorithm consists of a 3-tuple with

$$\{conclusion, reason, \{antecedents\}\}$$

for every derived value assignment. *Conclusion* represents a value assignment to a variable. *Reason* is the predicate or unit-open clause that lead to the derived value. *Antecedents* are the remaining variables in the unit-open clause whose values were referenced and for which the algorithm also maintains a 3-tuple.

Example. Consider the formula of a feature model: $(A \Rightarrow B) \wedge (B \Rightarrow \neg A)$, which is transformed to a CNF: $(\neg A \vee B) \wedge (\neg B \vee \neg A)$. As presented in Table 3.1, BCP sets $A = true$ and maintains its reason as *premise*. A premise does not have *antecedents*. BCP pushes respective unit-open clauses from the CNF to a stack. After examining $(\neg B \vee \neg A)$, BCP infers $B = false$, records its unit-open clause as reason and refers to variable A as its value was referenced. The BCP algorithm discovers the violated clause $(\neg A \vee B)$ and reports the contradiction which we can use to generate explanations.

LTMS and BCP are able to support the configuration process of a product line by giving explanations in propositional logic why a specific feature cannot be selected anymore [3]. To the best of

ID	Con.	Reason	AC	Stack
#1	A=1	premise		$(\neg A \vee B), (\neg B \vee \neg A)$
#2	B=0	$(\neg B \vee \neg A)$	#1	$(\neg A \vee B)$
#3		$(\neg A \vee B)$	#2	violated clause

Table 3.1.: BCP process. AC = *Antecedents*, Con. = *Conclusion*

our knowledge, LTMS and its internal inference engine BCP were never used to explain all of the considered anomalies in feature models or to give feedback in natural language. We adapted the BCP algorithm to do both.

4 Explanations for Anomalies

Given the original algorithm, the adapted BCP is straightforward to understand and it can explain all anomalies by varying the input parameters resulting in a generic approach. Minimal use cases for every anomaly serve as examples to demonstrate its applicability and are depicted in Fig. 4.1.

Dead Features. The most severe anomaly is a void feature model. The explanation of a void feature model is identical to the explanation of a dead feature, since a void model means that even the root feature is dead. We omit a separate section for void models at this point and focus on the explanation of dead features. BCP is used to compute a contradiction in the CNF and it generates an explanation based on the reasons for this contradiction. A contradiction in the CNF occurs, if the truth value assignment of the dead feature in the CNF is *true*. During the constraint propagation, BCP will encounter a violated clause and generate explanations.

An exemplary application to explain a dead feature is demonstrated with the feature model illustrated in Fig. 4.1a. An alternative feature *E* is implied by a core feature *B* which results in *E* to be dead. First, we build the CNF of the feature model as in Table 4.1. During the creation of a CNF, we additionally store information about the tracing of each literal to the feature model, since every literal belongs to a clause which either originates from the feature tree topology or from a cross-tree constraint. Then, the premise $D = \text{true}$ is propagated. As *D* is initially bound, it does not have any antecedents and all other variables are set to unknown. In a next step, all unit-open clauses from the CNF are collected in a stack. If all unit-open clauses have been pushed to the stack, the last occurred clause ($\neg D \vee \neg E$) is removed from the stack and examined. Since *D* is bound as *true* and the clause has to be satisfied, variable *E* is concluded to be *false*. The algorithm continues until a value is set resulting in a violated clause of the CNF, which is the case for the clause (*A*). *A* is the root feature and set to *false* in the fourth iteration of BCP resulting in a contradiction. The stack in (#5) is omitted at this point, but instead we present the violated clause of the CNF. Since the information of the tracing between all used clauses and the feature model is available, we can reuse it to create a comprehensible explanation. An explanation can be generated by reporting the reasons: first, take the violated clause into account and, second, traverse the reasons for conclusions backwards to the premises. To shorten the explanation, initial value assumptions do not need to be reported. For the dead feature *D*, the explanation is shown in natural language in Table 4.1.

False-Optional Features. False-optional features are modeled as optional, but are always present if their parent is as well. The adaption of the BCP algorithm is intuitive: By setting the truth value of an false-optional feature to *false* and the direct parent feature to *true*, a contradiction will appear. The CNF must contain the constraint which leads to a false-optional feature, since no violation would occur otherwise. An example for a false-optional feature is shown in Fig. 4.1b. Analogue to the process for dead features, we create the CNF and store the information about the origin of the clauses.

Table 4.2 demonstrates the constraint propagation of the BCP algorithm. Setting the false-optional

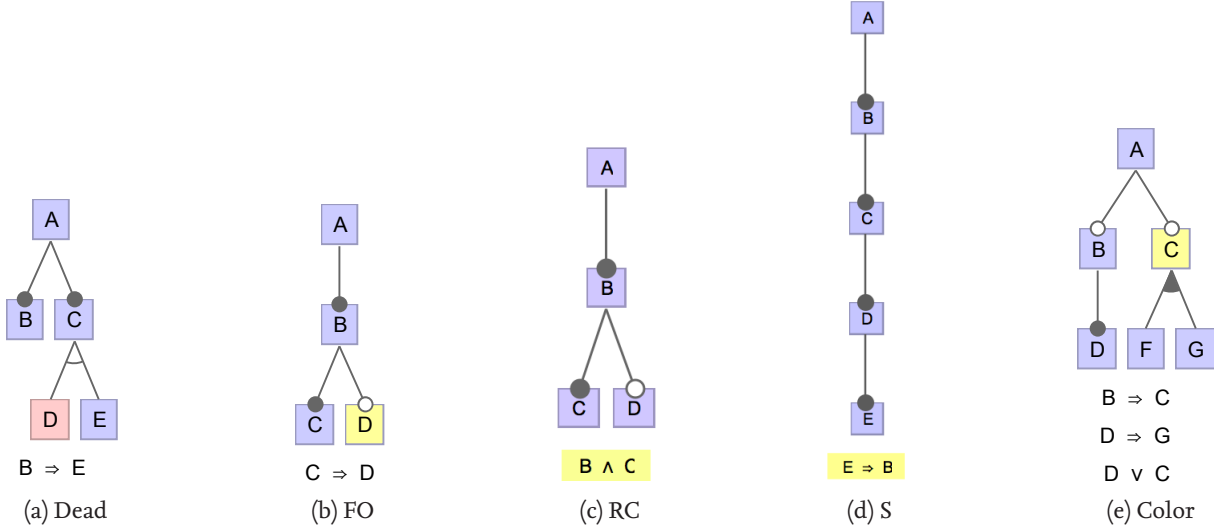


Figure 4.1.: Minimal examples of anomalies. FO = *false-optional*, RC = *redundant constraint*, S = *shorter explanation*.

feature D to *false*, and its parent B to *true*, infers features C to be *false*. In the third iteration, C has to be *false* to satisfy the constraint $(\neg C \vee D)$. This is a contradiction to the clause $(\neg B \vee C)$.

Redundant Constraints. A cross-tree constraint is only redundant, if and only if the relationship among its features is already modeled in some other way in the feature model. This relationship can be used to explain the redundant constraint. Therefore, the generated CNF from the feature model without the redundant constraint is needed. Setting the premises is a bit more challenging in this case. The truth values of the redundant constraint must result in a non-satisfiable constraint and therefore a contradiction, because information from the redundant constraint is still comprised in the CNF. We can have multiple assignments leading to a non-satisfiable constraint. It is necessary to analyze all of these combinations, since individual explanations may be incomplete. Each combination may result in a different explanation and only their union gives us a fully fledged explanation. Duplicate parts are ignored to keep it short.

For instance, we consider the feature model in Fig. 4.1c. The constraint $B \wedge C$ is redundant since B and C will always appear even without the constraint. The CNF of the feature model without redundancy is shown in Table 4.3. Creating a truth table for the redundant constraint $B \wedge C$ results in three different assignments that lead to an invalid formula. Table 4.3 presents the results computed by BCP. First, variables B and C are both bound to *false*. The latest unit-open clause $(\neg D \vee B)$ is removed from the stack and examined. Since B is bound *false* and the clause has to be satisfied, variable D is concluded to be *false*. The algorithm continues until a value is set resulting in a violated clause of the CNF, which is again the case for the clause (A) . In the second iteration, variable B is set to *false* and C is set to *true*. A violation in the CNF clause $(\neg C \vee B)$ appears since all terms are *false*. The third iteration sets variable B to *true* and C to *false*. A violation in the CNF clause $(\neg B \vee C)$ appears since all terms are *false*. Gathering the reasons for the three iterations results in the following set of clauses:

$$\{(A), (\neg A \vee B), (\neg C \vee B), (\neg B \vee C)\}$$

CNF: $(A) \wedge (\neg A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee A) \wedge (\neg C \vee A) \wedge (\neg C \vee D \vee E) \wedge (\neg D \vee C) \wedge (\neg E \vee C) \wedge (\neg D \vee \neg E) \wedge (\neg B \vee E)$				
ID	Con.	Reason	AC	Stack
#1	D=1	premise		$(\neg D \vee C), (\neg D \vee \neg E)$
#2	E=0	$(\neg D \vee \neg E)$	#1	$(\neg D \vee C), (\neg B \vee E)$
#3	B=0	$(\neg B \vee E)$	#2	$(\neg D \vee C), (\neg A \vee B)$
#4	A=0	$(\neg A \vee B)$	#3	$(\neg D \vee C)$
#5		(A)	#4	violated clause
Explanation: Feature D is dead, because: A is the root (#5), B is a mandatory child of A (#4), $B \Rightarrow E$ is a constraint (#3), E and D are alternative children of C (#2).				

Table 4.1.: Explaining the dead feature D .

CNF: $(A) \wedge (\neg A \vee B) \wedge (\neg B \vee A) \wedge (\neg B \vee C) \wedge (\neg C \vee B) \wedge (\neg D \vee B) \wedge (\neg C \vee D)$				
ID	Con.	Reason	AC	Stack
#1	D=0	premise		
#2	B=1	premise		$(\neg B \vee A), (\neg B \vee C), (\neg C \vee D)$
#3	C=0	$(\neg C \vee D)$	#1	$(\neg B \vee A), (\neg B \vee C)$
#4		$(\neg B \vee C)$	#3	violated clause
Explanation: Feature D is false-optional, because: C is a mandatory child of B (#4) and $C \Rightarrow D$ is a constraint (#3).				

Table 4.2.: Explaining the false-optional feature D .

Although the clause $(\neg D \vee B)$ is existent in the reasons, it is skipped since the explanations are generated backwards. The conclusion for variable A references variable C while ignoring all conclusions in between. Table 4.3 shows the final explanation.

4.1. Characteristics of BCP

Overall, the BCP algorithm meets the previously defined requirements. It is *generic*, since BCP works on the basis of logical constructions. Every propositional formula can be transformed into a CNF. Since a feature model can be mapped to a propositional formula, the BCP can process every kind of feature model regardless of their number of features, tree length, constraints, or anomalies. BCP is normally used to *efficiently* implement artificial intelligence and was already invented several decades ago [16]. BCP only maintains reasons leading to inferences. Clauses, which do not contribute to a violation during the propagation, are not used for the explanation. The additional tracing of CNF clauses to their feature model origin provides us with the information to create natural language explanations. A combination of both aspects leads to *informative* explanations for the

CNF: $(A) \wedge (\neg A \vee B) \wedge (\neg B \vee A) \wedge (\neg B \vee C) \wedge (\neg C \vee B) \wedge (\neg D \vee B)$				
ID	Con.	Reason	AC	Stack
#1.1	B=0	premise		$(\neg A \vee B), (\neg D \vee B)$ $(\neg A \vee B)$ violated clause
#1.2	C=0	premise		
#1.3	D=0	$(\neg D \vee B)$	#1.1	
#1.4	A=0	$(\neg A \vee B)$	#1.1	
#1.5		(A)	#1.4	
#2.1	B=0	premise		violated clause
#2.2	C=1	premise		
#2.3		$(\neg C \vee B)$	#2.1	
#3.1	B=1	premise		violated clause
#3.2	C=0	premise		
#3.3		$(\neg B \vee C)$	#3.1	
Explanation: Constraint $B \wedge C$ is redundant, because: A is the root (#1.5), B is a mandatory child of A (#1.4) and C is a mandatory child of B (#2.3, #3.3).				

Table 4.3.: Explaining the redundant constraint $B \wedge C$.

developer.

A minor drawback in BCP is that it cannot infer all truth values when it should. Consider two clauses: $A \Rightarrow B$, $B \Rightarrow \neg A$. The first constraint selects B if A is selected, while the second constraints deselects A . Therefore, selecting A implies its deselection. BCP is not able to infer that A cannot be present in a product. It only reveals the contradiction as soon as A is selected [20]. However, this characteristic is not restrictive for the generic explanation algorithm, since the detection of anomalies is performed by the feature modeling tool and not part of our contribution.

4.2. Improvements

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

Finding Short Explanations. BCP is also *order-sensitive*. Hence, the explanation depends on the order of clauses in the CNF. For example, processing a CNF from left to right may lead to a different explanation than from right to left. In particular, the BCP algorithm does not always find an explanation with a minimal length. A short explanation on its part offers the advantage of an improved comprehensibility for the developer, considering a feature model with thousands of features and corresponding large explanations for anomalies. Decoupling the algorithm from its *order-sensitivity* results in examining the clauses of the CNF in every possible order to find a minimal explanation. A first approach lies in the permutation of all CNF clauses. Running the algorithm on every possible clause order ensures finding a minimal explanation. However, this approach is not feasible in

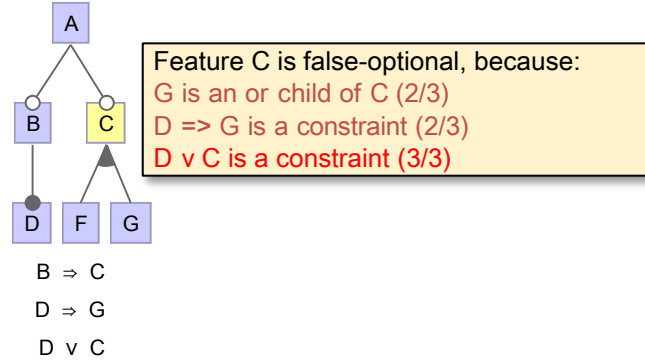


Figure 4.2.: Example of emphasized explanation parts.

terms of efficiency.

We propose a heuristic that takes advantage of the stack maintaining unit-open clauses. However, it does not guarantee to find a minimal explanation. The naive BCP algorithm reports a contradiction as soon as a violation occurs and terminates, while the stack might still contain unit-open clauses. Running the algorithm with those clauses again, BCP can generate further explanations. Consider Fig. 4.1d which contains the redundant cross-tree constraint $E \Rightarrow B$. The naive algorithm described previously generates an explanation in form of: *Constraint $E \Rightarrow B$ is redundant, because: E is a mandatory child of D, D is a mandatory child of C and C is a mandatory child of B.* Running the algorithm with the remaining clauses in the stack, a shorter explanation can be generated. For this example it is: *Constraint $E \Rightarrow B$ is redundant, because: A is the root, B is a mandatory child of A.* Both explanations comprise only relevant information to explain the constraint. However, the second explanation with a shorter length simply reports that feature B is mandatory instead of reporting the transitive chain which arises from the redundant constraint.

Emphasizing Core Parts in Explanations. As previously explained, several explanations can be generated for different anomalies. Besides finding a shorter explanation, this information can be further processed. Since every explanation consists of explanation parts which either represent a child-parent relationship or a constraint, same explanation parts might occur in multiple explanations for an anomaly. Such common explanation parts are more likely to represent the cause of an anomaly. Hence, editing respective parts of the feature topology or cross-tree constraints increases the probability to repair the anomaly.

For instance, changing a cross-tree constraint that is not available in all explanations cannot fix the anomaly, since at least one explanation containing a different cause remains for this anomaly. Fig. 4.1e demonstrates a feature model including three cross-tree constraints resulting in a false-optional feature C. The adapted BCP algorithm generates the following three explanations for this anomaly:

1. G is an or child of C, $D \Rightarrow G$ is a constraint and $D \vee C$ is a constraint.
2. $D \vee C$ is a constraint, D is a mandatory child of B and $B \Rightarrow C$ is a constraint.
3. $D \vee C$ is a constraint, $D \Rightarrow G$ is a constraint and

G is an or child of C.

Comparing the explanations above leads to the observation that $D \vee C$ is a constraint occurs most often in the explanations. Removing this constraint will repair the anomaly. Such information can be visually highlighted within an explanation in order to point out an explanation part which is more likely to cause the faulty relationship. Fig. 4.2 is a teaser for the next section and shows how the emphasis of core parts in explanations works in our implementation. By simply hovering the cursor on-top of the false-optional feature C, the developer gets the explanation on the right side of Fig. 4.2. One full explanation is shown. Behind every explanation part, numbers indicate how often a part is present in all generated explanations. This is also indicated by a color-intensity, which ranges from black to red. Black explanation parts are only present in one explanation at all, while red parts occur in all explanations.

5 Evaluation

Next, we give a description of our prototypical implementation in FeatureIDE, explain anomalies of the *Car* feature model from Section 2, and explore the adapted BCP algorithm in practice with feature models of different sizes in terms of features and constraints. For the evaluation, we investigate the typical size of a short explanation (cf. Section 4.2) as well as the time it takes to compute it. In detail, we investigate the following research questions:

RQ1: *Do explanations contain the necessary constraints to understand the anomaly?*

RQ2: *What is the performance impact of the algorithm?*

RQ3: *What is the average length of a short explanation?*

5.1. Implementation

We provide a prototypical implementation in the open-source framework FeatureIDE and it will be part of the next major release, namely FeatureIDE 3.1.0¹. The implementation of the adapted BCP uses the CNF generated from a feature model to infer reasons for an anomaly. An explanation is built by the combination of several reasons. However, such an explanation only consists of pure CNF clauses leading to the contradiction in the BCP algorithm (cf. Section 3). The information about CNF clauses is hardly beneficial for the developer, since their relation to the feature model is not obvious. We focus on the challenging task to provide the developer with useful feedback which is why we trace the relations between the feature model and its CNF clauses.

In a feature model, every feature comprises structural information. A feature occurs in a tree topology and may additionally occur in cross-tree constraints. Regarding the tree topology, a feature can take up different roles, i.e. *child* or *parent* and be additionally either *mandatory*, *optional* or in *alternative* and *or* groups. In order to generate explanations in natural language, the reasons can be presented using the feature model structure (cf. Section 4). The tracing of a clause to the feature model comes with some difficulties:

- Transforming the feature model into a CNF results in a one-to-many relationship between a feature and literals which represent a feature in the formula. Therefore, every literal has to carry its structural information whether it is child or parent in the tree topology or is contained inside a cross-tree constraint.
- This annotation of a literal has to be as efficient as possible, since FeatureIDE operates with literals in many different processes.

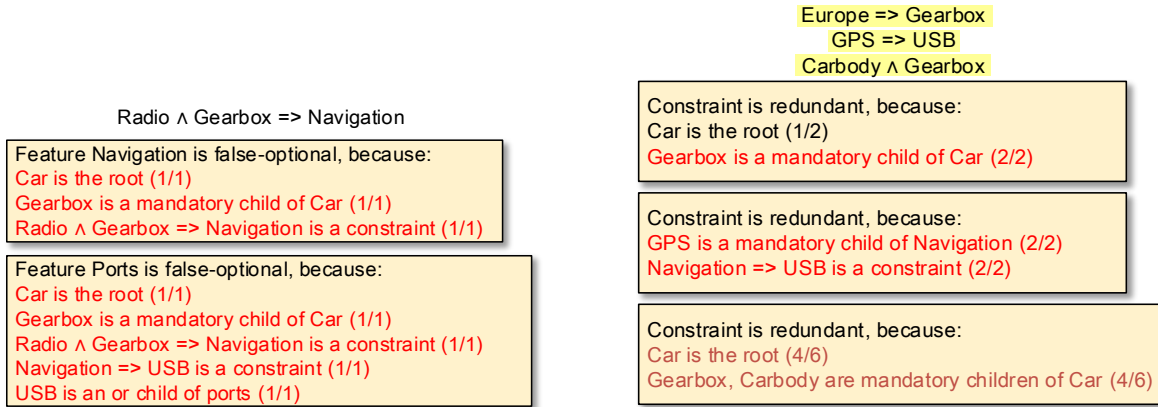
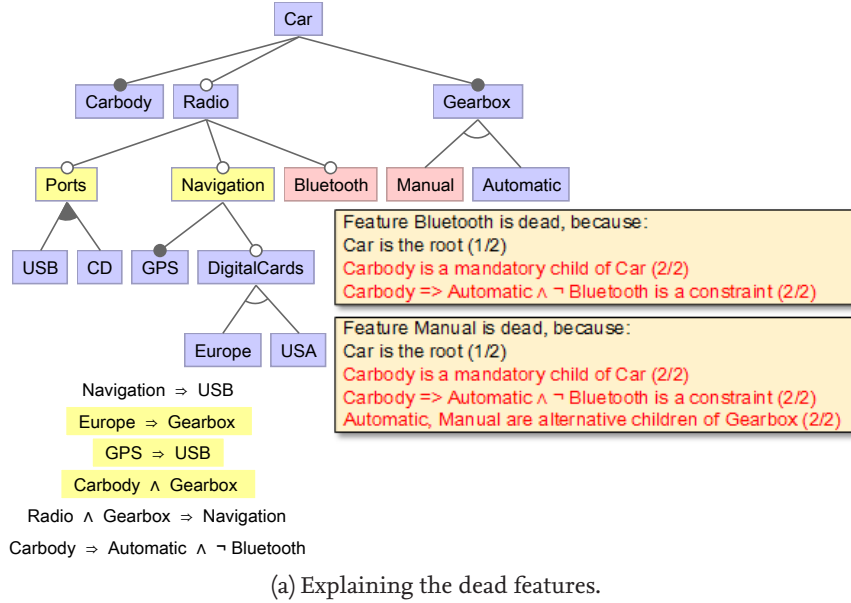
The annotation of every literal with the structural information takes place during the creation of the propositional formula. FeatureIDE itself already provides means to retrieve the structural information of a feature which is reused in this process. The source code is available on GitHub².

¹http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

²<https://github.com/FeatureIDE/FeatureIDE/tree/explanations/>

5.2. Qualitative Evaluation

Given our implementation, we can explain the anomalies presented in Section 2 and answer RQ1. Each anomaly of the *Car* feature model depicted in Fig. 2.1 is explained by a screenshot showing its explanation in FeatureIDE.



(b) Explaining the false-optional features.

(c) Explaining the redundant constraints.

Figure 5.1.: Explanations for the *Car* feature model.

Dead Features. The *Car* feature model has two dead features, namely *Bluetooth* and *Manual*. For convenience, Fig. 5.1a shows not only the explanations for both dead features, but also the feature model once more. Due to the core feature *Carbody*, we must select an *Automatic* transmission. *Automatic* itself is part of an alternative group and therefore excludes all other features in this group, here only *Manual*. *Bluetooth* is dead, since its negation is implied by a core feature. The algorithm generates two explanations for both.

False-Optional Features. Fig. 5.1b shows the cross-tree constraint responsible for the two false-optional features *Ports* and *Navigation*. *Navigation* must always be selected based on this constraint.

Model	# F	# C	# RC	# D	# FO
PPU	52	15	6	0	0
200-Model	200	20	8	106	13
500-Model	500	50	14	262	56
1000-Model	1,000	100	44	628	138
2000-Model	2,000	200	87	1,236	254
Automotive	2,513	2,833	563	192	12

Table 5.1.: Overview of evaluated feature models.

In contrast, *Ports* is only false-optional by taking a second cross-tree constraint into account with $Navigation \Rightarrow USB$. The explanation reveals this connection. Only one explanation was generated for these anomalies.

Redundant Constraints. Overall, there are three cross-tree constraints marked as redundant in the feature model in Fig. 5.1c. $Carbody \wedge Gearbox$ is redundant, because both features are already core features. The algorithm generated six explanations for this anomaly. Among all explanations, we present the improved one concerning length and coloring. The cross-tree constraint $Europe \Rightarrow Gearbox$ is redundant as implying a core feature is meaningless. In case of the last example $GPS \Rightarrow USB$ the redundancy is caused by the concatenation of two cross-tree constraints. By selecting the *Navigation*, it is already implied that the car has a *USB* port. Hence, it is not necessary to imply *USB* again with the *GPS* feature, which is automatically selected with *Navigation* because it is mandatory. Two explanations with identical parts were generated for this anomaly.

As a result, we observed that the explanations contain the required information to fix the anomalies. For the *Car* feature model, we have successfully shown the applicability of the presented algorithm as well as the satisfaction of RQ1. We are aware that this example does not cover a complete qualitative analysis of our approach with external developers. However, we postpone a detailed user study to further prove its benefits to future work.

5.3. Quantitative Evaluation

As next step, we focus on performance measurements and evaluations concerning the length of the short explanations to answer RQ2 and RQ3. Table 5.1 presents the evaluated feature models along with information about their number of features, constraints, and anomalies.

The first feature model, namely the *Pick and Place Unit* (PPU), originates from the *Institute of Automation and Information Systems* of the Technical University Munich. It is a real-world automation system [12]. Furthermore, additional generated feature models consisting of 200, 500, 1,000, and 2,000 features with a growing number of constraints and anomalies were used for the performance measurements and are freely available at the FeatureIDE website. A feature model from the automotive industry represents the biggest feature model with 2,513 features and 2,833 constraints and is our second real-world example. Void feature models were prohibited in the automatic generation process. The evaluation includes all detected anomalies in the feature models. The computation time has been measured using an Intel(R) Core(TM) i7-4800MQ CPU with 2.7 GHz and 16-GB RAM.

Computation Time. Table 5.2 shows the computation times for all performance measurements. First, each feature model was analyzed in FeatureIDE without the generation of any explanation

Model	No Expl. (s)	Tracing (s)	1. Expl. (s)	Shorter Expl. (s)	Colored Expl. (s)
PPU	0.02	0.02	0.05	0.05	0.06
200-Model	0.37	0.40	0.87	1.21	1.28
500-Model	5.08	5.53	9.67	11.42	11.65
1000-Model	43.42	46.41	88.77	116.77	116.96
2000-Model	352.61	372.89	567.27	831.72	832.11
Automotive	6,453.30	7,421.96	16,473.90	16,540.66	16,546.44

Table 5.2.: Computation times for all feature models in seconds (s).

(2. column). FeatureIDE only detects anomalies here. Second, we measured the computation time for the detection including our additional annotations for the tracing process. Third, we recorded the time to generate explanations. This part is divided into three individual steps (columns 4-6 in Table 5.2). The given values always include the anomaly detection and tracing time as well. The fourth column shows the time to calculate a first explanation for all anomalies in the feature model. Next, the algorithm tries to find shorter explanations and in a last step the explanation parts are colored based on their occurrence. Colored explanations force the generation of a first and possibly shorter (improved) one. All measurements have been repeated ten times for all models to reduce computation bias and we present the average in Table 5.2. We can observe that the computation time is roughly doubled in case of generating first explanations. In most cases, finding shorter explanations takes an additional 30% time over finding first ones, while the coloring process is almost instantly finished. Overall, the results look promising even for larger models.

Improved Explanation Length. The length of an explanation is measured in its number of parts. Fig. 5.2 illustrates the explanation length for all improved explanations found in the feature models. Given the rather small product line of the PPU, we observe that each explanation has at most seven parts (cf. Fig. 5.2a). Although the next model is about four times larger, we only see a slight increase in the explanation length in Fig. 5.2b. This trend continues for the larger models as well. Even for the Automotive feature model 50% of the explanations are positioned within 4 and 25 parts.

While reasoning on the length of improved explanations, we can determine the frequency of a first explanation already being the best possible based on our algorithm. In Table 5.3, the number of explanations for redundant constraints per model is illustrated as well as a how often an improved explanation has been found in later processing steps. Additionally, a relative shortening concerning the explanation length is presented. Given these results, we were able to make the following

Model	# Expl.	# Improved Expl.	Shorter (%)
PPU	6	1	44,4
200-Model	8	2	50,0
500-Model	14	2	25,1
1000-Model	44	11	39,7
2000-Model	87	21	48,4
Automotive	563	56	29,8

Table 5.3.: Finding improved explanations.

observations:

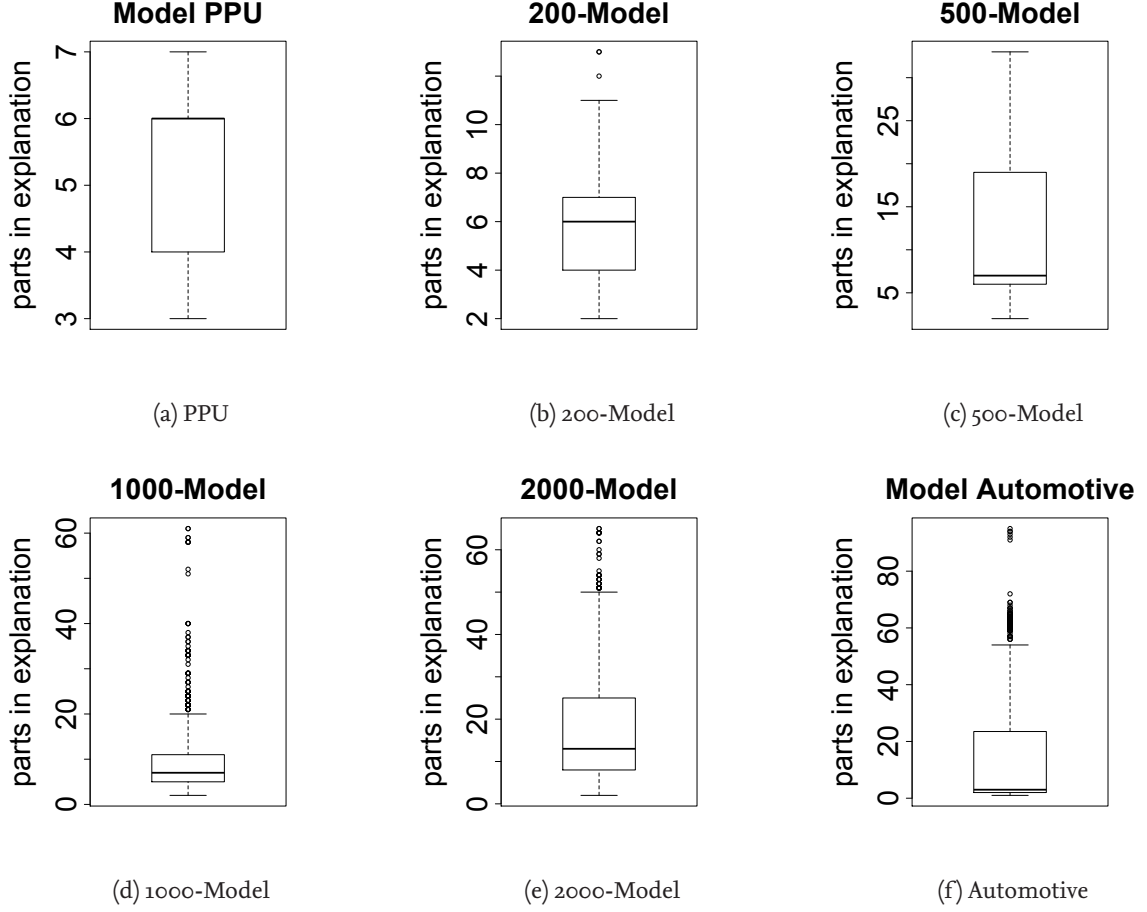


Figure 5.2.: Improved explanation lengths.

- The average explanation length increases only slightly compared to the number of features and constraints.
- In most cases, the adapted BCP already finds a short explanation in the first step.
- Searching for shorter explanations is worth the additional computation time as they are 25-50% smaller.

We conclude that the performance impact of the adapted BCP is acceptable with a doubling of the computation time, thus answering RQ2. Considering the length of shorter explanations (RQ3), we conclude that even for large feature models a significant number of explanations stay below 20 parts, which is still comprehensible for developers.

6 Related Work

There has been done a considerable amount of work on feature modeling and the automated analysis of feature models. FeatureIDE, TVL, FAMILIAR, SPLConqueror, Clafer, and pure::variants are some results of this research [6, 29, 33, 2]. In addition, many approaches including the previously mentioned ones already provide support for the detection of anomalies [5, 4, 15]. Hemakumar uses the SPIN model checker to detect anomalies with a BCP implementation in the background [20]. The approach is limited due to the difficulty of model checking large feature models. Another approach focuses mainly on requirements for product lines, but can also detect anomalies in feature models. However, no support for explanations is mentioned [40].

Considering the explanation process, we are most closely related to previous work by Batory [3]. The first attempt to connect propositional formulas to product lines was performed by Mannion [26]. Batory built upon this work and introduced LTMS and BCP to support developers in the configuration of a variant based on a feature model [3]. The implementation is available in GUIDSL and provides feedback in terms of why a certain feature cannot be selected. Hence, GUIDSL can explain dead features. However, the explanations are only presented in form of propositional formulas that can be quite difficult to understand. In addition, the remaining anomalies are not explained at all. We used this approach as foundation and improved it by explaining all anomalies, emphasizing important parts, provide natural language explanations, show its scalability and rather than focusing on the configuration process, we already support the developer during development of the feature model.

Another approach translates the feature model into a constraint satisfaction problem and uses the QuickXPlain algorithm to detect and explain anomalies [25, 21]. However, the constraint solver has to be changed directly, while we provide an approach that can be used independently of any solver. Their tool is implemented in SWI-Prolog, but the source code is not available. Since our implementation is based on FeatureIDE, it uses Java and is freely available on GitHub. An approach to detect and explain dead and false-optional features is also presented in [31]. The explanations are given in natural language and seem to be rivaling our explanation length. Zaid et al. also focus on the detection and explanation of dead features [42]. All three approaches are missing the explanation of redundant constraints and a large evaluation. Felfernig et al. provide an approach that is able to explain all anomaly types using the FastDiag and FMCore algorithms [19, 18, 17]. The explanations are given in form of constraint sets and the feature models are written in the SXFM format of S.P.L.O.T.. Similar to our approach, they are independent from underlying reasoning techniques. Trinidad did a significant amount of work in this domain. In an early work, the focus lies on the explanation of dead features [39]. This work is extended to all anomaly types and implemented in FAMA [38, 37]. The evaluation of both approaches only includes small feature models. Emphasizing important parts in explanations is not available. Kramer et al. add attributes to the feature model and generate explanations during runtime. The approach is only viable for small or medium sized feature models [24]. Osman et al. extend feature models by so called variation points, before an

explanation is possible [14]. It was our goal to refrain from such extensions or to put additional workload on the developer.

The problem of anomalies is also related to *ifdefs* in the C preprocessor. The most famous example is the Linux kernel with 10000 features predominantly implemented with *ifdefs*. Tartler et al. propose techniques to detect anomalies for this large-scale software system [34, 36, 35]. Even fixing aspects in the Linux kernel is possible [41]. An application of our work to explain dead code in the Linux kernel is imaginable.

7 Conclusion

We have presented an algorithm for explaining anomalies in feature models. It is able to explain all types of anomalies by varying the input parameters while using the same generic process. An additional tracing of clauses to structural information of the feature model provides us with the means to give the developer natural language explanations. The scalability is shown by analyzing several large-scale feature models including an industrial one. In all cases, the explanation length stays acceptable at the cost of a doubled computation time for the complete process. We implemented our approach as part of the open-source framework FeatureIDE.


Regarding future work, the approach can be extended in several directions. The tooltip is only a first step to support the developer. We plan to highlight the path in the actual feature model leading to an anomaly. This visualization improves the comprehensibility of our explanations even further. It is our goal to do a user study with the developers of the PPU to extend the results of our qualitative analysis. We are currently working on an extension to multi software product lines to explain not only anomalies but transitive constraints between the different feature models as well.

Bibliography

- [1] K. R. Apt. “Some Remarks on Boolean Constraint Propagation”. In: *New Trends in Constraints*. Springer, 1999, pp. 91–107.
- [2] K. Bak, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wasowski. “Clafer: Unifying Class and Feature Modeling”. In: *Software & Systems Modeling* (2014), pp. 1–35.
- [3] D. Batory. “Feature Models, Grammars, and Propositional Formulas”. In: *Proc. Int’l Software Product Line Conf. (SPLC)*. Berlin, Heidelberg: Springer, 2005, pp. 7–20.
- [4] D. Benavides, A. Felfernig, J. A. Galindo, and F. Reinfrank. “Automated Analysis in Feature Modelling and Product Configuration”. In: ed. by J. Favaro and M. Morisio. *Proc. Int’l Conf. Software Reuse (ICSR)*. Berlin, Heidelberg: Springer, 2013.
- [5] D. Benavides, S. Segura, and A. Ruiz-Cortés. “Automated Analysis of Feature Models 20 Years Later: A Literature Review”. In: *Information Systems* 35.6 (2010), pp. 615–708.
- [6] A. Classen, Q. Boucher, and P. Heymans. “A Text-Based Approach to Feature Modelling: Syntax and Semantics of {TVL} ”. In: *Science of Computer Programming* 76.12 (2011), pp. 1130–1143.
- [7] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley, 2001.
- [8] P. Codognet and D. Diaz. “A Simple and Efficient Boolean Solver for Constraint Logic Programming”. In: *Journal of Automated Reasoning* 17.1 (1996), pp. 97–128.
- [9] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. New York, NY, USA: ACM/Addison-Wesley, 2000.
- [10] M. Davis and H. Putnam. “A Computing Procedure for Quantification Theory”. In: *Journal of the ACM (JACM)* 7.3 (1960), pp. 201–215.
- [11] J. De Kleer. “An Assumption-Based TMS”. In: *Artificial intelligence* 28.2 (1986), pp. 127–162.
- [12] A. Dolgui, J. Sasiadek, M. Zaremba, S. Feldmann, C. Legat, and B. Vogel-Heuser. “Engineering Support in the Machine Manufacturing Domain through Interdisciplinary Product Lines: An Applicability Analysis”. In: *IFAC-PapersOnLine* 48.3 (2015), pp. 211–218.
- [13] J. Doyle. “A Truth Maintenance System”. In: *Artificial Intelligence* 12.3 (1979), pp. 231–272.
- [14] A. O. Elfaki, S. Phon-Amnuaisuk, and C. K. Ho. “Knowledge Based Method to Validate Feature Models”. In: *Proc. Int’l Software Product Line Conf. (SPLC)*. 2008.
- [15] A. O. Elfaki, S. Phon-Amnuaisuk, and C. K. Ho. “Using First Order Logic to Validate Feature Model.” In: *Proc. Int’l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*. 2009, pp. 169–172.
- [16] B. Faltings and P. Struss. *Recent Advances in Qualitative Physics*. MIT Press, 1992.

- [17] A. Felfernig, M. Schubert, and C. Zehentner. “An Efficient Diagnosis Algorithm for Inconsistent Constraint Sets”. In: *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 26 (01 02/2012), pp. 53–62.
- [18] A. Felfernig and M. Schubert. “Fastdiag: A Diagnosis Algorithm for Inconsistent Constraint Sets”. In: *Proceedings of the Workshop on the Principles of Diagnosis (DX 2010)*, Portland, OR, USA. 2010, pp. 31–38.
- [19] A. Felfernig, D. Benavides, J. Galindo, and F. Reinfrank. “Towards Anomaly Explanation in Feature Models”. In: *Proceedings of the Workshop on Configuration, Vienna, Austria*. Citeseer. 2013, pp. 117–124.
- [20] A. Hemakumar. “Finding Contradictions in Feature Models.” In: *Proc. Int’l Software Product Line Conf. (SPLC)*. 2008, pp. 183–190.
- [21] U. Junker. “Preferred Explanations and Relaxations for Over-Constrained Problems”. In: *AAAI-2004*. 2004.
- [22] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-21. SE Institute, 1990.
- [23] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. “FeatureIDE: A Tool Framework for Feature-Oriented Software Development”. In: *Proc. Int’l Conf. Software Engineering (ICSE)*. Formal demonstration paper. Vancouver, Canada: IEEE, 05/2009, pp. 611–614.
- [24] D. Kramer, C. S. Sauer, and T. Roth-Berghofer. “Towards Explanation Generation using Feature Models in Software Product Lines.” In: *KESE*. 2013.
- [25] U. Lesta, I. Schaefer, and T. Winkelmann. “Detecting and Explaining Conflicts in Attributed Feature Models”. In: *Proc. Int’l Workshop Formal Methods and Analysis in Software Product Line Engineering (FMSPLE)*. 2015, pp. 31–43.
- [26] M. Mannion. “Using First-Order Logic for Product Line Model Validation”. In: *Proc. Int’l Software Product Line Conf. (SPLC)*. Berlin, Heidelberg: Springer, 2002, pp. 176–187.
- [27] J. P. Martins and S. C. Shapiro. “Reasoning in Multiple Belief Spaces”. PhD thesis. State University of New York at Buffalo, 1983.
- [28] T. Mens and S. Demeyer, eds. *Software Evolution*. Springer-Verlag, Berlin Heidelberg, 2008.
- [29] M. Mernik, B. R. Bryant, G. Cabri, M. Ganzha, M. Acher, P. Collet, P. Lahire, and R. B. France. “FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models”. In: *Science of Computer Programming* 78.6 (2013), pp. 657–681.
- [30] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Berlin, Heidelberg: Springer, 09/2005.
- [31] L. Rincón, G. L. Giraldo, R. Mazo, and C. Salinesi. “An Ontological Rule-Based Approach for Analyzing Dead and False Optional Features in Feature Models”. In: *Electronic Notes in Theoretical Computer Science* 302 (2014), pp. 111–132.
- [32] V. Rutenburg. “Propositional Truth Maintenance Systems: Classification and Complexity Analysis”. In: *Annals of Mathematics and Artificial Intelligence* 10.3 (1994), pp. 207–231.

- [33] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake. “SPL Conqueror: Toward Optimization of Non-Functional Properties in Software Product Lines”. In: *Software Quality Journal* 20.3 (2012), pp. 487–517.
- [34] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. “Configuration Coverage in the Analysis of Large-Scale System Software”. In: *Proceedings of the Workshop on Programming Languages and Operating Systems*. ACM. 2011, p. 2.
- [35] R. Tartler, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. “Dead or Alive: Finding Zombie Features in the Linux Kernel”. In: *Proc. Int’l Workshop Feature-Oriented Software Development (FOSD)*. ACM. 2009, pp. 81–86.
- [36] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. “Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem”. In: *Proceedings of the Conference on Computer systems*. ACM. 2011, pp. 47–60.
- [37] P. Trinidad. “Automating the Analysis of Stateful Feature Models”. PhD thesis. University of Seville, 2012.
- [38] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. “Automated Error Analysis for the Agilization of Feature Modeling”. In: *Journal of Systems and Software* 81.6 (2008), pp. 883–896.
- [39] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, and M. Toro. “Explanations for Agile Feature Models”. In: *Proceedings of the Workshop on Agile Product Line Engineering (APLE)*. 2006, p. 44.
- [40] T. von der Maßen and H. Lichter. “Deficiencies in Feature Models”. In: *Proceedings of the Workshop on Software Variability Management for Product Derivation-Towards Tool Support*. 2004.
- [41] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki. “Generating Range Fixes for Software Configuration”. In: *Proc. Int’l Conf. Software Engineering (ICSE)*. 2012, pp. 58–68.
- [42] L. A. Zaid, F. Kleinermann, and O. De Troyer. “Applying Semantic Web Technology to Feature Modeling”. In: *ACM Symposium on Applied Computing, SAC ’09*. Honolulu, Hawaii: ACM, 2009, pp. 1252–1256.



Technische Universität Carolo-Wilhelmina zu Braunschweig (Germany)
Institute of Software Engineering and Automotive Informatics

Mühlenpfordtstr. 23
D-38106 Braunschweig

2012-03	J. Rang	An analysis of the Prothero–Robinson example for constructing new DIRK and ROW methods
2012-04	S. Kolatzki, M. Hagner, U. Goltz and A. Rausch	A Formal Definition for the Description of Distributed Concurrent Components - Extended Version
2012-05	M. Espig, W. Hackbusch, A. Litvinenko, H. G. Matthies and P. Wähnert	Efficient low-rank approximation of the stochastic Galerkin matrix in tensor formats
2012-06	S. Mennike	A Petri Net Semantics for the Join-Calculus
2012-07	S. Lity, R. Lachmann, M. Lochau, I. Schaefer	Delta-oriented Software Product Line Test Models - The Body Comfort System Case Study
2013-01	M. Lochau, S. Mennicke, J. Schroeter und T. Winkelmann	Extended Version of 'Automated Verification of Feature Model Configuration Processes based on Workflow Petri Nets'
2013-02	S. Lity, M. Lochau, U. Goltz	A Formal Operational Semantics of Sequential Function Tables for Model-based SPL Conformance Testing
2013-03	L. Giraldi, A. Litvinenko, D. Liu, H. G. Matthies, A. Nouy	To be or not to be intrusive? The solution of parametric and stochastic equations – the “plain vanilla” Galerkin case
2013-04	A. Litvinenko, H. G. Matthies	Inverse problems and uncertainty quantification
2013-05	J. Rang	Improved traditional Rosenbrock–Wanner methods for stiff ODEs and DAEs
2013-06	J. Kosłowski	Deterministic single-state 2PDAs are Turing-complete
2014-01	B. Rosić, J. Diekmann	Stochastic Description of Aircraft Simulation Models and Numerical Approaches
2014-02	M. Krosche, W. Heinze	A Robustness Analysis of a Preliminary Design of a CESTOL Aircraft
2014-03	J. Rang	Adaptive timestep control for fully implicit Runge–Kutta methods of higher order
2014-04	S. Mennicke, J.-W. Schicke-Uffmann, U. Goltz	Free-Choice Petri Nets and Step Branching Time
2014-05	A. Martens, C. Borchert, T. O. Geissler, O. Spinzcyk, D. Lohmann, R. Kapitza	Exploiting determinism for efficient protection against arbitrary state corruptions
2014-06	J. Rang	An analysis of the Prothero–Robinson example for constructing new adaptive ESDIRK methods of order 3 and 4
2014-07	J. Rang, R. Niekamp	A component framework for the parallel solution of the incompressible Navier–Stokes equations with Radau-IIA methods
2014-08	J. Rang	The Prothero and Robinson example: Convergence studies for Runge–Kutta and Rosenbrock–Wanner methods
2014-09	J. Wayetens, B. V. Rosic	Comparison of deterministic and probabilistic approaches to identify the dynamic moving load and damages of a reinforced concrete beam
2014-10	B. Rosic, J. Sykora, O. Pajonk, A. Kucerova, H. G. Matthies	Comparison of Numerical Approaches to Bayesian Updating
2016-01	M. Kowal, S. Ananieva, T. Thüm	Explaining Anomalies in Feature Models