Explaining Anomalies in Feature Models

Matthias Kowal
TU Braunschweig, Germany
m.kowal@tu-braunschweig.de

Sofia Ananieva
FZI Research Center for Information Technology, Germany
ananieva@fzi.de

Thomas Thüm
TU Braunschweig, Germany
t.thuem@tu-braunschweig.de

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 an efficient and generic algorithm for explaining different anomalies in feature models. Additionally, we achieve a benefit for the developer by computing short explanations expressed in a user-friendly manner and by emphasizing specific parts in explanations that are more likely to be the cause of an anomaly. We provide an open-source implementation in FeatureIDE and show its scalability for industrial-size feature models.

Categories and Subject Descriptors D.2.9 [*Software Engineering*]: Management—Software configuration management; D.2.13 [*Software Engineering*]: Reusable Software

Keywords Software Product Lines, Feature Models, Explanations, Anomalies

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. 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 [8, 22]. A product line is a set of related systems that have common and differing features. For example, operating systems can have an x86 or an x64 architecture. Generating a specific variant of the system can efficiently be achieved by maximizing the reuse potential of such features. Hence, product lines enable a reduced time-to-market, are more cost-efficient, and simplify the maintenance compared to classical development methods [9, 31].

Since not all combinations of features are useful, feature models are often used to express the intended variability [4, 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 [29].

We define an anomaly as redundancy or inconsistency caused by the evolution of a feature model. A feature model contains redundancy, if semantic information is modeled in multiple ways, which is in general not preferable [42]. Inconsistencies comprise contradictions within a feature model, e.g., a feature that cannot be selected in any configuration. To detect such anomalies, adequate tool support is needed for developers when evolving feature models [4, 5]. In addition to the detection, the user needs information in terms of why an anomaly occurs in a feature model. Such explanations support the correction of anomalies [4]. We define an explanation as the subset of relationships between features 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 [5].

Existing work to compute explanations often includes only a subset of all anomaly types, gives explanations which are hard to understand, e.g., in form of logic formulas, or the generation of explanations does not scale [3, 20, 32]. Hence, we propose an approach avoiding these limitations without putting additional workload on the developer or introducing

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

GPCE'16, October 31 – November 1, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4446-3/16/10...
http://dx.doi.org/10.1145/2993236.2993248

new language concepts to feature models. In summary, we make the following contributions:

1. We propose an efficient and generic algorithm for the explanation of all kinds of anomalies that is easy to extend.
2. We compute short explanations expressed in a user-friendly manner and emphasize most relevant parts in explanations.
3. Based on our open-source tool support in FeatureIDE, we evaluate the scalability of the 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. 1. Relationships between parent and child features are expressed using the following notations and their semantics (see legend in Fig. 1 for graphical notation) [9, 22]:

- *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. 1 shows a significantly simplified feature model of a car. A car must have a *Carbody* and a *Gearbox*. One can choose between *Manual* and *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. 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]. Some examples of anomalies are already visible in Fig. 1, which we now discuss. Specific reasons for anomalies will be explained in detail later in Section 5.2.

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

Table 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.

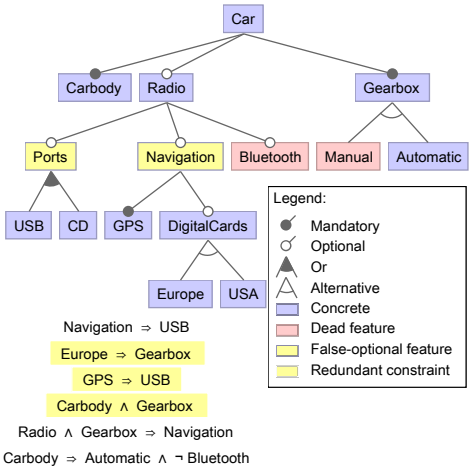


Figure 1: Anomalies in a car feature model

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. 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 1.

Dead Features. Features are regarded as dead, if they can never be selected in any variant of the product line [42]. Hence, the feature has no effect at all. Table 1 shows the respective call of the SAT solver. For example, in Fig. 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. 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 cross-tree constraints as the feature *Ports*. Again, the respective SAT call is presented in Table 1.

Redundant Constraints. A cross-tree constraint is redundant if its removal does not change the validity of configurations. For example, such a constraint implies the selection of a core feature, which is superfluous. 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 when it is not obvious which constraint should be removed. A SAT solver checks if identical solutions are satisfiable for two feature models. In particular, one feature model contains the redundant constraint whereas the other model does not, while they are identical in all other aspects (see Table 1).

Problem Statement. Feature models and their underlying propositional logic have existed 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 available in feature modeling tools such as FeatureIDE [23] or pure::variants [6]. The explanation of these anomalies is often neglected or not sufficient in terms of explanation length, scalability and comprehensibility [3, 4, 17, 25, 41]. At this point, we leave it to the reader to find explanations for the anomalies shown in Fig. 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. 1 are explained in Section 5 using our implementation in FeatureIDE.

3. Generic Anomaly Explanation

It is our motivation to support the developer such that a solid decision process is possible regardless of the anomaly type. This led 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 user-friendly and as short as possible. Explanation parts, which have a high probability to cause a defect, 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 [18]. In GUIDSL, a user can select a specific feature and obtains feedback why other features are not available anymore. However, explanations are just given in terms of propositional formulas and GUIDSL only supports the explanation of dead features. It provides the user with explanations concerning the configuration possibilities of a product line. BCP 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 [11, 28, 33]. 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. 2. 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.

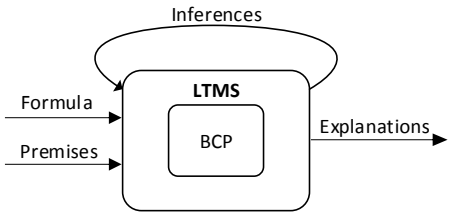


Figure 2: LTMS

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 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, whether 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 = *true*, then X and Y must be true.

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

BCP is also known as *Unit Resolution* and makes use of such rules to conclude inferences [10]. 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. 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. 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 the BCP algorithm is presented in Fig. 3. 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 the 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.

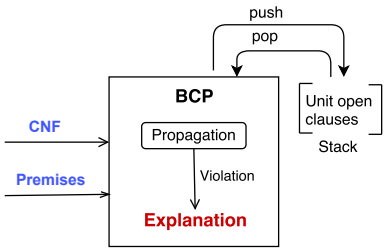


Figure 3: Overview of the BCP algorithm.

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 2, 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 a 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 use to generate an explanation.

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

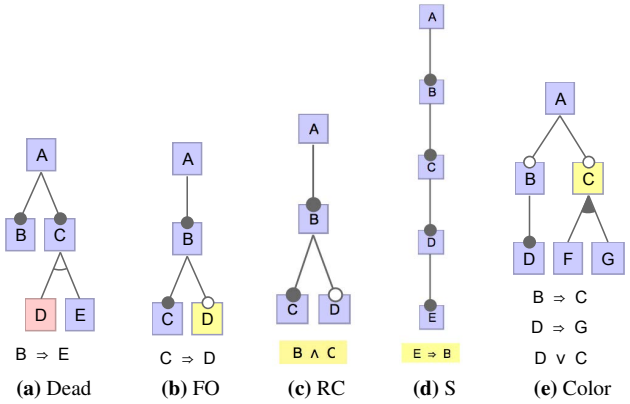
Table 2: BCP process. AC = *Antecedents*, Con. = *Conclusion*

LTMS and BCP are able to support the configuration process of a product line by giving explanations based on propositional logic why a specific feature cannot be selected anymore [3]. To the best of 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 a user-friendly manner. We adapted the BCP algorithm to do both.

4. Explanations for Anomalies

BCP is highly generic since it operates on a CNF and premises, i.e., a set of initial truth value assumptions for literals contained in the CNF. Therefore, BCP is able to explain every defect in a feature model that can be encoded in the described way. Minimal use cases for the presented anomalies serve as examples to demonstrate the applicability of BCP and are depicted in Fig. 4.

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. Hence, we omit a separate discussion of 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 an explanation.



(a) Dead **(b)** FO **(c)** RC **(d)** S **(e)** Color

Figure 4: Minimal examples of anomalies. FO = *false-optional*, RC = *redundant constraint*, S = *shorter explanation*.

An exemplary application to explain a dead feature is demonstrated with the feature model illustrated in Fig. 4a. An alternative feature E is implied by a core feature B which results in D to be dead. The respective BCP process including a resulting explanation is depicted in Table 3. First, we create a CNF for the feature model. During the creation of the 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 = *true* is propagated. As D is initially bound, it does not have any antecedents and all other variables are set to unknown. In the 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. Whereas initial value assumptions do not need to be reported. For the dead feature D, the explanation is shown in a user-friendly manner in Table 3.

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 a false-optional feature to *false* and the direct parent feature to *true*, a contradiction will occur.

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 3: Explaining the dead feature D.

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. 4b.

Table 4 demonstrates the constraint propagation of the BCP algorithm. Setting the false-optional feature D to *false*, and its parent B to *true*, feature C must be *false* to satisfy the constraint $(\neg C \vee D)$. However, this is a contradiction to the clause $(\neg B \vee C)$.

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

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: Explaining the false-optional feature D.

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 contained in the CNF. We can have multiple assignments leading to a non-satisfiable constraint. It is necessary to analyze all of these combinations as 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. 4c. 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 that constraint is shown in Table 5. Creating a truth table for the redundant constraint $B \wedge C$ results in three different assignments that lead to an invalid formula. Table 5 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 because all terms are *false*. The third iteration sets variable B to *true* and C to *false*. A violation in clause $(\neg B \vee C)$ appears as all terms are *false*. Gathering the reasons for the three iterations results in the following set of clauses:

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

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 5 shows the final explanation.

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		
#1.2	C=0	premise		$(\neg A \vee B), (\neg D \vee B)$
#1.3	D=0	$(\neg D \vee B)$	#1.1	$(\neg A \vee B)$
#1.4	A=0	$(\neg A \vee B)$	#1.1	
#1.5		(A)	#1.4	violated clause
#2.1	B=0	premise		
#2.2	C=1	premise		
#2.3		$(\neg C \vee B)$	#2.1	violated clause
#3.1	B=1	premise		
#3.2	C=0	premise		
#3.3		$(\neg B \vee C)$	#3.1	violated clause

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 5: Explaining the redundant constraint $B \wedge C$.

4.1 Characteristics of BCP

Overall, the BCP algorithm meets previously defined requirements. Since BCP works on the basis of predicate logic and every feature model can be mapped to a propositional formula, BCP is highly *generic*. It has been invented several decades ago and was initially used to *efficiently* implement artificial intelligence [18]. The BCP algorithm only maintains reasons leading to inferences and ignores clauses, which do not contribute to a violation during the propagation. The additional tracing of CNF clauses to their feature model origin provides us with the information to create user-friendly explanations. A combination of both aspects leads to *informative* explanations for the 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 no limitation for the generic explanation algorithm as efficient techniques for the detection of anomalies already exist [23].

4.2 Improvements

In order to explain an anomaly, the 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 *order-sensitive*, meaning that an explanation depends on the order of clauses in the CNF. For example, processing a CNF from left to right may lead to a longer explanation than processing the CNF from right to left. However, a short explanation typically 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. However, this approach is not feasible in terms of efficiency.

We propose a heuristic that takes advantage of the stack maintaining unit-open clauses. The basic 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. 4d which contains the redundant cross-tree constraint $E \Rightarrow B$. The basic algorithm described previously generates an explanation in the 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. *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 mentioned, several explanations can be generated a certain anomaly. This information can be further processed. Since

every explanation consists of explanation parts which either represent a child-parent relationship or a cross-tree constraint, some 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.

In particular, changing a cross-tree constraint that is not available in all explanations cannot fix the anomaly, as at least one explanation containing a different cause remains for this anomaly. Fig. 4e 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. 5 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. 5. 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, while red parts occur in all explanations.

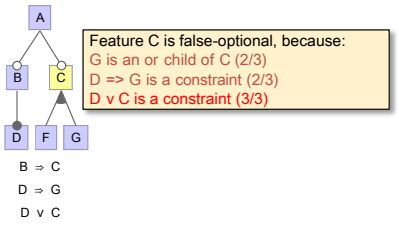


Figure 5: Example of emphasized explanation parts.

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 an explanation?*

5.1 Implementation

We provide a prototypical implementation in the open-source framework FeatureIDE, which was released in 3.1.0 [1]. 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, as 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 a user-friendly manner, 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 a 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.

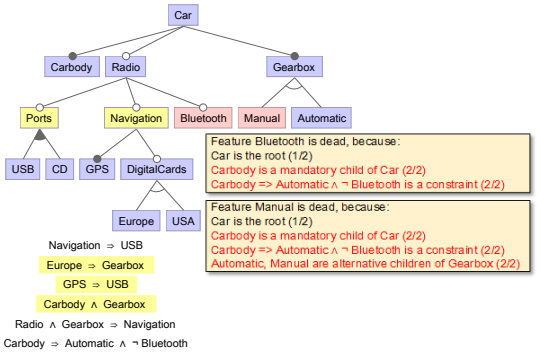
The annotation of every literal with the structural information takes place during the creation of the propositional formula. FeatureIDE itself already provides means to retrieve the structural information of a feature which is reused here.

5.2 Qualitative Evaluation

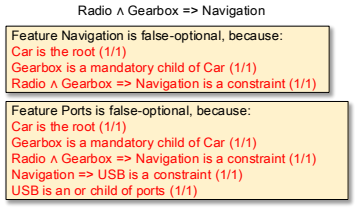
Given our implementation, we can explain the anomalies presented in Section 2 and answer RQ1. Each anomaly of the *Car* feature model depicted in Fig. 1 is explained by a screenshot showing its explanation in FeatureIDE.

Dead Features. The *Car* feature model has two dead features, namely *Bluetooth* and *Manual*. For convenience

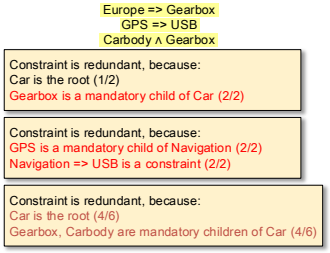
[1] https://github.com/FeatureIDE/FeatureIDE/tree/explanations/



(a) Explaining the dead features.



(b) Explaining the false-optional features.



(c) Explaining the redundant constraints.

Figure 6: Explanations for the *Car* feature model.

Fig. 6a 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. 6b shows the cross-tree constraint responsible for the two false-optional features *Ports* and *Navigation*. *Navigation* must always be selected based on this constraint. In contrast, *Ports* is only false-optional by taking a second cross-tree constraint $Navigation \Rightarrow USB$ into account. 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. 6c. $Carbody \wedge Gearbox$ 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.

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 6: Overview of evaluated feature models.

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.

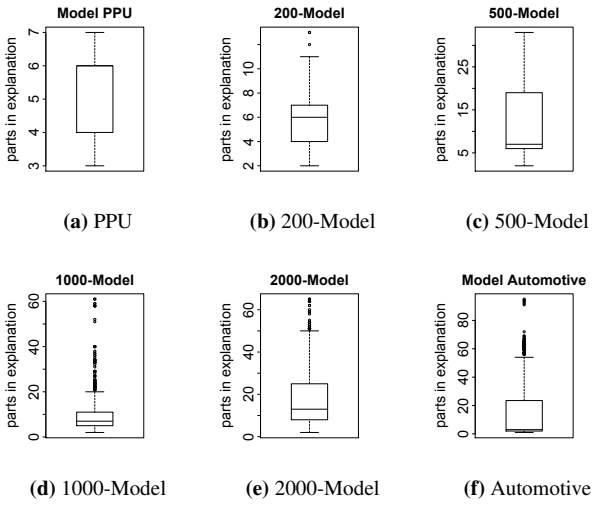
As a result, we observed that the explanations contain the required information to fix the anomalies. For the *Car* feature model, we have successfully shown the applicability of the presented algorithm as well as the satisfaction of RQ1. We are aware that this example does not cover a complete qualitative 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 the next step, we focus on performance measurements and evaluations concerning the length of the short explanations to answer RQ2 and RQ3. Table 6 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 an automation system [12]. Furthermore, additional generated feature models consisting of 200, 500, 1,000, and 2,000 features and constraints, respectively, were taken into account for the performance measurements and have already been used in prior evaluations [23]. 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 7 shows the computation times for all performance measurements. First, each feature model was analyzed in FeatureIDE without the generation of any explanation (column 2). FeatureIDE only *detects* anomalies here. Second, we measured the computation time for the detection including our additional annotations for the trac-



(a) PPU **(b)** 200-Model **(c)** 500-Model

(d) 1000-Model **(e)** 2000-Model **(f)** Automotive

Figure 7: Improved explanation lengths.

ing process. Third, we recorded the time to generate explanations. This part is divided into three individual steps (columns 4-6 in Table 7). 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 a shorter explanation compared to the first one. Finally, the computation and visualization of relevant explanation parts takes place which 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 7. We can observe that the computation time is roughly doubled in the 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.

Improved Explanation Length. We measure the explanation length in its number of parts. Fig. 7 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.7a). Although the next model is about four times larger, we only see a slight increase in the explanation length in Fig. 7b. This trend continues for the larger models as well. Even for the Automotive feature model 50% of the explanations consist of 4 to 25 parts.

While reasoning on the length of improved explanations, we can determine the frequency of a first explanation already being the shortest possible based on our algorithm. In Table 8, the number of explanations for redundant constraints per model is illustrated as well as 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 observations:

Model	# Expl.	# Improved Expl.	Shorter (%)
PPU	6	1	44.4
200-Model	8	2	50
500-Model	14	2	25.1
1000-Model	44	11	39.7
2000-Model	87	21	48.4
Automotive	563	56	29.8

Table 8: Finding improved explanations.

- 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 (cf. RQ2). Considering the length of shorter explanations (cf. 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 a considerate amount of work on feature modeling and the automated analysis of feature models. In particular, many tools are available such as FeatureIDE, TVL, FAMILIAR, SPLConqueror, Clafer, and pure::variants [2, 7, 30, 34]. In addition, many approaches including the previously mentioned ones already provide support for the detection of anomalies [4, 5, 14]. Hemakumar uses the SPIN model checker to detect anomalies with a BCP implementation in the background [20]. However, approaches without explanations such as [20, 42] are rather useless for large feature models as it is almost impossible to comprehend why a certain defect occurred.

The first attempt to connect propositional formulas to product lines was performed by Mannion [27]. 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. In contrast to our work, GUIDSL can only explain dead features and focuses on the configuration process, while we already support the developer during development of the feature model. We improved Batory's work in different ways: First, by explaining all kinds of anomalies and also expressing explanations in a user-friendly language. Second, our explanations are closer to the feature model by referring to the structure and, hence, increasing the transparency of explanations. Third, we emphasize important explanation parts and show the scalability.

With dead and false-optional features many approaches only consider a subset of anomalies. Lesta et al. translate the feature model into a constraint satisfaction problem and adapt the QuickXPlain algorithm to detect and explain

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

anomalies in attributed feature models [21, 25]. However, there are many differences compared to our approach: while we show the scalability of BCP using differently sized feature models up to 2,513 features, Lesta et al. only use one feature model with less than 500 features to evaluate the efficiency of their approach. Additionally, the constraint solver has to be changed directly, while we provide an approach that can be used independently of any solver based on an arbitrary anomaly detection. Finally, we provide an open-source implementation. Trinidad et al. implemented the open-source tool FAMA, which provides explanations for dead and false-optional features [39–41]. However, explanations are given in an greatly abbreviated way, which may be at the expense of quickly recognizing the cause of an anomaly. Further approaches are presented in literature, but none of them explain redundant cross-tree constraints. All mentioned approaches [25, 32, 41, 43] miss an evaluation with large feature models containing thousands of features and constraints.

Felfernig et al. provide an approach that is able to explain all anomaly types using the FastDiag algorithm [15–17]. Explanations are given in the form of constraint sets which have to be adapted or deleted to repair the anomaly. Similar to our approach, FastDiag explains all types of anomalies and is independent from underlying reasoning techniques. Contrary to our contribution, the evaluation is limited to feature models containing only up to 172 features and the explanations are not directly related to the structural information of the feature model making it more difficult for the developer to comprehend the anomaly.

Similar to GUIDSL, Kramer et al. present an approach to generate explanations for configuration purposes. In particular, they add explanation fragments as attributes to features and relationships while the concatenation of such fragments forms a complete explanation for a configuration. Although explanations are also expressed in a user-friendly way, the approach is only viable for small or medium sized feature models [24]. In contrast, we give explanations during the modeling phase and strive to find short explanations.

The analysis of a product line to determine anomalies is not limited to feature models only. Actual source code can also be analyzed, e.g., to find dead code blocks or compiler errors [38]. Tartler et al. propose techniques to detect anomalies in the source code of the Linux kernel [35–37]. An application of our work to explain dead code or compiler errors could help developers in fixing those problems, especially as the actual cause could also be in the feature model.

Further related work concentrates on finding guaranteed minimal explanations [19, 26]. Given an unsatisfiable formula, a minimal explanation is determined by using a *satisfiability module theories* (SMT) solver to find a minimal unsatisfiable subset of clauses. Finding minimal explanations would obviously be an improvement over our approach. However, it is an open research question whether such algorithms scale to large feature models.

7. Conclusion

We have presented a generic and efficient algorithm for explaining anomalies in feature models based on predicate logic. It is able to explain all types of anomalies which can be encoded in a CNF and a set of initial truth value assumptions. An additional tracing of literals to structural information of the feature model provides us with the means to express explanations in a user-friendly manner. 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. For long explanations, we plan to highlight the path in the actual feature model leading to an anomaly. In addition, irrelevant parts of the model with respect to the explanation could be concealed in large feature models improving the comprehensibility of our explanations even further. Another aspect is the calculation of the shortest possible explanation. It is our goal to do a user study with the developers of the PPU to inspect whether the explanations are actually useful to practitioners. Finally, minor edits to feature models should not result in a complete recomputation of the feature model analysis and, hence, improve the performance impact to detect and explain anomalies during the evolution of a feature model.

Acknowledgments

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).

References

[1] K. R. Apt. Some Remarks on Boolean Constraint Propagation. In *New Trends in Constraints*, pages 91–107. Springer, 1999.

[2] K. Bak, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wasowski. Clafer: Unifying Class and Feature Modeling. *Software & Systems Modeling*, pages 1–35, 2014.

[3] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 7–20, Berlin, Heidelberg, 2005. Springer.

[4] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–708, 2010.

[5] D. Benavides, A. Felfernig, J. A. Galindo, and F. Reinfrank. *Automated Analysis in Feature Modelling and Product Configuration*. Proc. Int'l Conf. Software Reuse (ICSR). Springer, Berlin, Heidelberg, 2013.

[6] D. Beuche. Modeling and building software product lines with pure::variants. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 255–255, New York, NY, USA, 2012. ACM.

[7] A. Classen, Q. Boucher, and P. Heymans. A Text-Based Approach to Feature Modelling: Syntax and Semantics of {TVL} . *Science of Computer Programming*, 76(12):1130 – 1143, 2011.

[8] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA, 2001.

[9] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM/Addison-Wesley, New York, NY, USA, 2000.

[10] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[11] J. De Kleer. An Assumption-Based TMS. *Artificial intelligence*, 28(2):127–162, 1986.

[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. *IFAC-PapersOnLine*, 48(3):211 – 218, 2015.

[13] J. Doyle. A Truth Maintenance System. *Artificial Intelligence*, 12(3):231–272, 1979.

[14] 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)*, pages 169–172, 2009.

[15] 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*, pages 31–38, 2010.

[16] A. Felfernig, M. Schubert, and C. Zehentner. An Efficient Diagnosis Algorithm for Inconsistent Constraint Sets. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 26:53–62, 2 2012.

[17] A. Felfernig, D. Benavides, J. Galindo, and F. Reinfrank. Towards Anomaly Explanation in Feature Models. In *Proceedings of the Workshop on Configuration, Vienna, Austria*, pages 117–124. Citeseer, 2013.

[18] K. D. Forbus and J. D. Kleer. *Building Problem Solvers*. MIT Press, 1993.

[19] O. Guthmann, O. Strichmann, and A. Trostanetski. Minimal unsatisfiable core extraction for SMT. In *Proc. Int'l Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, 2016.

[20] A. Hemakumar. Finding Contradictions in Feature Models. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 183–190, 2008.

[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. Technical Report 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)*, pages 611–614, Washington, DC, USA, May 2009. IEEE. Formal demonstration paper.

[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)*, pages 31–43, 2015.

[26] M. H. Liffiton and K. A. Sakallah. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. *Journal of Automated Reasoning*, 40:1–33, 2008.

[27] M. Mannion. Using First-Order Logic for Product Line Model Validation. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 176–187, Berlin, Heidelberg, 2002. Springer.

[28] J. P. Martins and S. C. Shapiro. *Reasoning in Multiple Belief Spaces*. PhD thesis, State University of New York at Buffalo, 1983.

[29] T. Mens and S. Demeyer, editors. *Software Evolution*. Springer-Verlag, Berlin Heidelberg, 2008.

[30] 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. *Science of Computer Programming*, 78(6):657 – 681, 2013.

[31] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Berlin, Heidelberg, Sept. 2005.

[32] 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. *Electronic Notes in Theoretical Computer Science*, 302:111–132, 2014.

[33] V. Rutenburg. Propositional Truth Maintenance Systems: Classification and Complexity Analysis. *Annals of Mathematics and Artificial Intelligence*, 10(3):207–231, 1994.

[34] 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. *Software Quality Journal*, 20(3):487–517, 2012.

[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)*, pages 81–86. ACM, 2009.

[36] 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*, page 2. ACM, 2011.

[37] 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*, pages 47–60. ACM, 2011.

[38] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 47(1):6:1–6:45, June 2014. ISSN 0360-0300. doi: 10.1145/2580950.

[39] P. Trinidad. *Automating the Analysis of Stateful Feature Models*. PhD thesis, University of Seville, 2012.

[40] 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)*, page 44, 2006.

[41] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. Automated Error Analysis for the Agilization of Feature Modeling. *Journal of Systems and Software*, 81(6): 883–896, 2008.

[42] 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.

[43] L. A. Zaid, F. Kleinermann, and O. De Troyer. Applying Semantic Web Technology to Feature Modeling. In *ACM Symposium on Applied Computing*, SAC '09, pages 1252–1256, New York, NY, USA, 2009. ACM.