Reasoning about Edits to Feature Models

Thomas Thüm
School of Computer Science
University of Magdeburg
tthuem@st.ovgu.de

Don Batory
Dept. of Computer Science
University of Texas at Austin
batory@cs.utexas.edu

Christian Kästner
School of Computer Science
University of Magdeburg
ckaestne@ovgu.de

Abstract

Features express the variabilities and commonalities among programs in a software product line (SPL). A feature model defines the valid combinations of features, where each combination corresponds to a program in an SPL. SPLs and their feature models evolve over time. We classify the evolution of a feature model via modifications as refactorings, specializations, generalizations, or arbitrary edits. We present an algorithm to reason about feature model edits to help designers determine how the program membership of an SPL has changed. Our algorithm takes two feature models as input (before and after edit versions), where the set of features in both models are not necessarily the same, and it automatically computes the change classification. Our algorithm is able to give examples of added or deleted products and efficiently classifies edits to even large models that have thousands of features.

1 Introduction

Software product line (SPL) engineering is a paradigm for systematic reuse [2, 32, 21]. From common assets, different programs of a domain can be assembled. Programs of an SPL are distinguished by *features*, which are domain abstractions relevant to stakeholders and are typically increments in program functionality. Every program of an SPL is represented by a unique combination of features, and an SPL could have millions of distinct programs. A *feature model* compactly defines all features in an SPL and their valid combinations; it is basically an AND-OR graph with constraints.

SPLs and their feature models evolve over time. Even small changes to a feature model, like moving a feature from one branch to another, can unintentionally change the set of legal feature combinations. For example, edits may preclude the creation of programs that were previously built or enlarge the set of programs that can be built. Understanding the impact of feature model edits is known to be impractical to determine manually. Tool support to provide feedback to engineers to explain how changes to a feature model have altered the program membership of an SPL is a fundamental problem.

In prior work, Czarnecki et al. defined operations to specialize feature models [13, 14, 25], where a feature model X is a *specialization* of a feature model Y if the set of products in X is a subset of that in Y. Alves et al. discussed operations called *refactorings* that maintain the set of products or add new products to an SPL [1]. We call this latter case (when new products are added) a *generalization* of a feature model. In both approaches, a set of sound operations is used to edit feature models to determine whether a model is a specialization or refactoring or generalization of another. To determine the effects of feature model edits using these results requires that all edits be expressed as a sequence of predefined operations (refactorings, generalizations, or specializations). However, a tool that allows only sound operations to be performed would have the feel of a hard-to-use structure editor, e.g., how one changes a feature model X into a target feature model Y using a sequence of sound operations is not obvious. A path planner [17] would be needed to automatically compute a sequence (if it exists). If a required operation is missing, designers would be out of luck, as the desired change to a feature model would be precluded by the editor. This approach seems too difficult.

In other work, Sun et al. used first-order logic to determine if two feature models are equivalent, i.e., both models have the same set of products [37]. Janota and Kiniry used higher-order logic to determine whether a feature model is a specialization of another [20]. Both approaches work if both models have the same set of features; their results are not applicable to model edits that add or delete features. Further, we are unaware of performance studies that demonstrate first-order and higher-order logic approaches scale to large feature models efficiently.

In this paper, we present and evaluate an algorithm to determine the relationship between two feature models (i.e., specialization/refactoring/generalization/none of these) using satisfiability solvers. We overcome the limitations of

ICSE'09, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE

previous solutions: (a) arbitrary edits on feature models are supported, (b) the models to be compared need not have the same set of features, and (c) we empirically show that our algorithm can efficiently compare models with thousands of features.

2 Feature Model Background

A *feature model* defines the valid combinations of features in a domain. It is organized hierarchically and is graphically depicted as a *feature diagram* [21]. As an example, we show a feature model of an embedded database product line called FAME-DBMS [33] in Figure 1. Every feature has one parent except the *root feature*. Connections between a feature and its group of children are distinguished as *And*- (no arc), *Or*- (filled arc) and *Alternative*-groups (unfilled arc) [19, 12, 4, 8, 16]. The children of *And*-groups can be either mandatory (filled circle) or optional (unfilled circle).

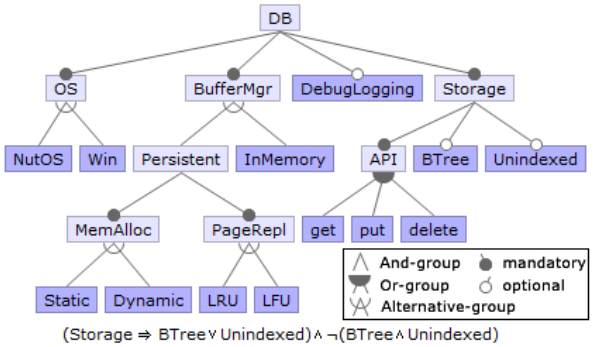


Figure 1. Feature Model of FAME-DBMS

Feature models have the following semantics: if a feature is selected, so too is its parent. Further, if the parent is selected, all mandatory children (features) of an *And*-group are selected; in *Or*-groups, at least one child must be selected, and in *Alternative*-groups, exactly one child is selected.

A feature model may also have constraints, called *cross-tree constraints*, that cannot be easily expressed hierarchically [12, 3, 7]. Cross-tree constraints can be arbitrary propositional formulas and may be written below a feature diagram as in Figure 1. Our notation is based on *guidsl* [3].

As in [3], a *terminal* or *concrete* feature is a leaf and a *non-terminal* or *compound* or *abstract* feature (we do not distinguish between these terms) is an interior node of a feature diagram. A unique characteristic of [3] is that each non-terminal feature represents a composition of features that are its descendants. This is not a standard interpretation in the feature modeling literature. (Stated differently, [3] assumes no assets are associated with a non-terminal feature; assets are bound only to concrete features). Although we develop our work using *guidsl* assumptions, simple tree rewrites can transform non-terminal features with assets into terminal features. Figure 2 illustrates such a transformation.

Group Type	Propositional Formula
And	$(P \Rightarrow \bigwedge_{i \in M} C_i) \wedge (\bigvee_{1 \le i \le n} C_i \Rightarrow P)$
Or	$P \Leftrightarrow \bigvee_{1 \le i \le n} C_i$
Alternative	$(P \Leftrightarrow \bigvee_{1 \le i \le n} C_i) \wedge \text{atmost1}(C_1, \ldots, C_n)$

Table 1. Propositional Formulas

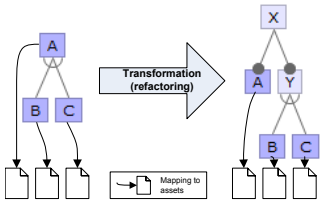


Figure 2. Refactoring for Non-Terminal Features with Assets

Our choice of *guidsl* assumptions is not critical to our results, although it does simplify their development as clean distinctions among different types of features are made. Different notations or extensions of feature models, e.g, [21, 19, 12, 37, 1, 16, 34], can be translated into *guidsl* representations [39].[1] This includes non-terminal features that have assets, which can be handled appropriately and transparently by translations.

A *configuration* of a feature model is a set of concrete features. A configuration is *valid* if the selection of all features contained in the configuration and the deselection of all other concrete features is allowed by the feature model. For example, a valid configuration of the feature model in Figure 1 is {*Win, InMemory, get, BTree*}. A *software product line* $L(f)$ is the set of all configurations that are valid for the feature model f.

Propositional Formulas. Every feature model can be translated into a propositional formula with a variable for each feature using the rules given in Table 1 [3]. P denotes a compound feature and $C_1, \ldots, C_n$ are its children. If the relationship among children is of type *And*, then $M \subseteq \{1, \ldots, n\}$ identifies the mandatory features by their index. The term $\text{atmost1}(C_1, \ldots, C_n)$ is equivalent to $\bigwedge_{i<j}(\neg C_i \vee \neg C_j)$ [17].

The propositional formula of a feature model is constructed by (a) conjoining the propositional formula for each

[1] An exception are feature models with local attributes, i.e., with attributes whose values are defined by configurations and not by the feature model itself [13].

group in the feature model, (b) conjoining all cross-tree constraints and (c) selecting the root feature. For example, the feature model f in Figure 1 has the formula $P(f)$:

$$
\begin{aligned}
&DB \wedge \\
&\textit{//tree constraints:} \\
&(DB \Rightarrow OS \wedge BufferMgr \wedge Storage) \wedge \\
&(OS \vee BufferMgr \vee DebugLogging \vee Storage \Rightarrow DB) \wedge \\
&(OS \Leftrightarrow NutOS \vee Win) \wedge \text{atmost1}(NutOS, Win) \ \wedge \ldots \ \wedge \\
&\textit{//cross-tree constraints:} \\
&(Storage \Rightarrow BTree \vee Unindexed) \wedge \neg(BTree \wedge Unindexed)
\end{aligned}
$$

Using a propositional formula representation of a feature model, it possible to automatically determine whether a given configuration is valid. For every feature that is selected, its variable is assigned `true`, otherwise `false`. The configuration is valid if and only if the formula evaluates to `true` [3].

3 Reasoning

Editing a feature model produces a new feature model. We want to know how the product line of the original feature model changes. An edit is a *refactoring*, i.e., no new products are added and no existing products are removed, a *specialization* meaning that some existing products are removed and no new products are added, a *generalization* when new products are added and no existing products removed, or an *arbitrary edit* otherwise. This classification is summarized in Table 2, where shaded regions are after-sets and unshaded regions are before-sets.

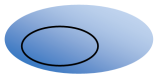
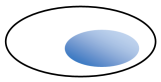
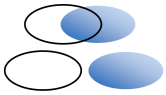
	No Products Added	Products Added
No Products Deleted	 Refactoring	 Generalization
Products Deleted	 Specialization	 Arbitrary Edit

Table 2. Classification of Feature Model Edits

A refactoring is useful in restructuring a feature model without changing its set of products; specializations are useful during derivation processes when products are eliminated; and generalizations arise when an SPL is extended. Designers should avoid arbitrary edits and restructure them in terms of a sequence of specializations, generalizations, and refactorings, to understand the evolution of a feature model.

Reasoning about feature model edits is far from trivial. For humans, even small edits can be difficult to classify, especially if cross-tree constraints are present. For example, consider Figure 1 once more. The cross-tree constraints in this model assert that *BTree* and *Unindexed* are mutually exclusive. This same relationship could be expressed by making *BTree* and *Unindexed* alternative features without cross-tree constraints. This change is a refactoring, as no products are added or removed. Although this is a simple example, it does take some time to verify manually that the change is indeed a refactoring. As modifications become more complicated, and as feature models grow in size, it can be extraordinarily difficult to understand the effect of feature model edits manually.

In this section, we show how *satisfiability (SAT) solvers* can be used to classify feature model edits. Our algorithm is described in four steps. First, we present an intuitive algorithm known in the literature which does not scale. Second, we present a technique that we call *simplified reasoning* to decrease computational complexity to a level that makes it practical to reason about edits to large feature models where both models have the same set of concrete features. Third, we provide a solution for cases when both feature models are not defined over the same set of concrete features. Finally, we give a procedure to handle abstract features.

3.1 Reasoning with Formulas

Our classification of edits is based on set relations. Let f and g be feature models and $L(f)$ and $L(g)$ denote their respective set of products. If edits transform f into g, these edits are (a) a generalization if $L(f) \subset L(g)$, (b) a specialization if $L(g) \subset L(f)$, (c) a refactoring if $L(f) = L(g)$, and otherwise (d) arbitrary. Let $P(f)$ denote the propositional formula for feature model f. The connection of subset relations to propositional formulas is defined by [20]:

$$(L(f) \subseteq L(g)) \equiv (P(f) \Rightarrow P(g)) \tag{1}$$

That is, all solutions to $P(f)$ must be solutions to $P(g)$. The intuition is that $L(f) \subseteq L(g)$ when $P(f) \Rightarrow P(g)$ for *all* possible configurations. Hence we are only interested in whether $P(f) \Rightarrow P(g)$ is a tautology. SAT solvers can verify this. A propositional formula X is a tautology if its negation $\neg X$ is not satisfiable. The following equation computes $\neg X$ with $X = (P(f) \Rightarrow P(g))$.

$$
\begin{aligned}
\neg(P(f) \Rightarrow P(g)) &\equiv \neg(\neg P(f) \vee P(g)) \\
&\equiv P(f) \wedge \neg P(g)
\end{aligned} \tag{2}
$$

For computation, SAT solvers require a formula to be in *conjunctive normal form (CNF)*. It is easy to convert a propositional formula of a feature model into CNF. In fact, it is straightforward to show that if there are n features in

a feature model, a crude upper bound on the worst case number of clauses is $O(n^2)$ and the number of disjuncts in a clause is $O(n)$ [39]. In practice, the number of clauses and the average number of disjuncts per clause is much lower than these bounds.

We implemented this strategy and tested it with several feature models (see Section 4). For very small feature models this algorithm can classify edits quickly (changes to feature models with 25 features can be classified in less than a second). However, already with feature models with as little as 50 features, the calculation time exceeds the mark of 10-20 seconds considered acceptable by user interface guidelines [31, 35]. For feature models with hundreds of features often used in practice, this algorithm does not scale. Our results are consistent with experience by others [9, 40].

The source of these long calculation times lies in the negation of P(g) in Equation 2. Let c denote the number of CNF clauses in $P(g)$ and let d denote the number of disjuncts per clause. When $P(g)$ is negated and transformed into CNF, the number of clauses produced is $O(d^c)$ where each clause has $O(c)$ terms. Observe that an exponential explosion of clauses occurs, rendering a straightforward evaluation of Equation 2 for even small values of c and d impractical.[2] In the next section, we explain how to avoid explosions.

3.2 Simplified Reasoning

In the typical case when feature model g is derived from f, we would expect f and g to have many CNF clauses in common. We can rewrite $P(f)$ and $P(g)$ as:

$$P(f) = p_f \wedge c \tag{3}$$
$$P(g) = p_g \wedge c \tag{4}$$

where c denotes the conjunction of the common clauses of $P(f)$ and $P(g)$, and p_f is the conjuction of clauses that are contained in $P(f)$ but not in $P(g)$. The fact that $p_f \wedge c \wedge \neg c$ is a contradiction leads to the following equivalence:

$$P(f) \wedge \neg P(g) \equiv P(f) \wedge \neg p_g \tag{5}$$

This rewrite helps significantly when f and g are largely similar, but by itself does not avoid the exponential explosion. It simply reduces the number of clauses in $P(g)$ from n to a lower n'.

To eliminate the explosion, let p_g denote the CNF expression $R_1 \wedge R_2 \wedge \ldots \wedge R_{n'}$, where each R_i is a disjunction of terms, and a term is a feature variable or its negation. We now simplify Equation 5:

$$\begin{aligned} P(f) \wedge \neg p_g &\equiv P(f) \wedge \neg \bigwedge_{1 \le i \le n'} R_i \\ &\equiv P(f) \wedge \bigvee_{1 \le i \le n'} \neg R_i \\ &\equiv \bigvee_{1 \le i \le n'} (P(f) \wedge \neg R_i) \end{aligned} \tag{6}$$

Equation 6 tells us that we can now make n' distinct calls to a SAT solver to determine whether formula $(P(f) \wedge \neg R_i)$ is satisfiable, once for each value of i. If any formula is satisfiable, we know that $L(f)$ is not a subset of $L(g)$ because Equation 1 evaluates to false. The possibility to terminate early also helps to reason efficiently with large feature models. Moreover, each formula $(P(f) \wedge \neg R_i)$ is simple: R_i is a disjunction of d_i terms, and its negation is a conjunction of d_i terms. That is, small problems are submitted to a SAT solver, and this increases reasoning efficiency, too. In addition, the output of a SAT solver is useful: it provides an example that justifies our classification (e.g., a product configuration that is present in $P(f)$ but not in $P(g)$, or vice versa). This too is helpful in explaining to engineers how edits change a feature model.

In summary, it is not necessary to compute the CNF of a negated feature model. We can efficiently classify feature model edits using multiple calls to SAT solvers, posing simple questions each time. This makes it possible to classify changes for large feature models efficiently, as well as illustrating changes using counter examples. We call this technique *simplified reasoning*.

3.3 Adding/Removing Concrete Features

Simplified reasoning applies to feature models that have the same set of features. But what if edits add or remove concrete features? To motivate our solution, consider an example.

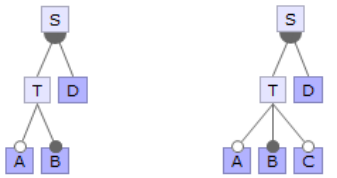
Example. Figure 3 shows a feature model f where an optional concrete feature C has been added to produce feature model g. We know that this edit is a generalization as g contains all products of f and more. Now let's see if $P(f) \Rightarrow P(g)$ holds by applying our technique:

$$P(f) = (S \wedge (S \Leftrightarrow T \vee D) \wedge (T \Rightarrow B) \wedge (A \vee B \Rightarrow T)) \tag{7}$$
$$P(g) = (S \wedge (S \Leftrightarrow T \vee D) \wedge (T \Rightarrow B) \wedge (A \vee B \vee C \Rightarrow T)) \tag{8}$$

$P(f) \Rightarrow P(g)$ holds if and only if the following formula is unsatisfiable:

$$P(f) \wedge \neg(A \vee B \vee C \Rightarrow T) \tag{9}$$

[2] As an optimization, we could use 'satisfiability equivalent' transformations of $\neg P(g)$ into CNF, instead of equivalent transformations [10]. This would limit the explosion of terms, but requires additional variables. Furthermore, it is not possible to calculate an example to why an edit is a generalization or to determine the number of added or deleted products. Therefore, we base our solution on equivalent transformations.



(a) Feature Model f (b) Feature Model g

Figure 3. Adding Concrete Feature C to f

Unfortunately, Equation 9 is satisfiable and therefore $P(f) \Rightarrow P(g)$ fails. Here is a counter-example: let S, D and C be `true` and T, A and B be `false`. For this assignment, $P(f)$ is `true`, and $P(g)$ is `false`. This example tells us that we have to generalize Equation 1 to take into account concrete feature additions and deletions. Futhermore, we cannot simply say that adding a feature creates new products. A user might add or remove a dead feature (that by some constraint can never be selected in any product), which is in fact a refactoring. □

The solution for this example, which we extrapolate into a general solution, is the following: when we change feature model f to feature model g by adding a feature, we automatically add this feature to f as well but deselect it there. Now our algorithm detects a refactoring if the new feature is always deselected in g (i.e., it is a dead feature) or a generalization (new products) in all other cases. Similarly, when we change feature model f to feature model g by removing a concrete feature, we automatically add this feature to g but deselect it. Our algorithm detects a refactoring when the removed feature was dead or a specialization (fewer products) in all other cases.

We operationalize the above in the following way: For every conrete feature A that is contained in g but not in f we conjoin the new clause $\neg A$ to $P(f)$ resulting in $P'(f)$. Thus, when feature A is deselected in g, its deselection in f will have no impact on f's product-line (i.e., $P'(f)$ reduces to $P(f)$). Analogously, we conjoin a new clause $\neg B$ to $P(g)$ for every feature B that is contained in f but no longer in g resulting in $P'(g)$. Doing all this generalizes Equation 1:

$$(L(f) \subseteq L(g)) \Leftrightarrow (P'(f) \Rightarrow P'(g)) \qquad (10)$$

By using Equation 10, we can apply our simplified reasoning algorithm to classify edits when concrete features are added or removed.

In our above example, when feature C is selected, $P'(f)$ is `false` for all remaining variable assignments. For all assignments where $P'(f)$ is `true`, C is deselected and $P'(g)$ is also `true`, which is what we wanted to prove.

3.4 Handling Abstract Features

Our last topic considers the handling of abstract features. Again, we start with an example to motivate why abstract features need special treatment.

Example. Figure 4(a) shows a feature model f, where A and B are *Or*-connected. Feature model g is derived from f by adding a new abstract feature T which does not change the product line membership of f (i.e., $L(f) = L(g)$).

(a) Feature Model f (b) Feature Model g

Figure 4. Adding Abstract Feature T to f

Using our translation of Section 3.2, the propositional formulas of f and g are as follows:

$$P'(f) = (S \wedge (S \Leftrightarrow A \vee B)) \qquad (11)$$
$$P'(g) = (S \wedge (S \Leftrightarrow T) \wedge (T \Leftrightarrow A \vee B)) \qquad (12)$$

As before, we give a counter-example: setting T `false` and all other variables to `true` shows $P'(f) \not\Rightarrow P'(g)$. □

The fundamental difference between concrete features and abstract features is that an abstract feature does not appear in a configuration; its value is computed from the values of its children. (Remember that in other feature modeling approaches, there may not be a clear distinction between non-terminal features that can be selected, and abstract features whose value is computed. As mentioned earlier, by applying rewrites such as that in Figure 2, we model the selection of a non-terminal feature as that of selecting a concrete subfeature of a non-terminal feature that is abstract). In principle, abstract features need a different treatment than concrete features because they represent a semantically different concept.

The solution for this example, which we extrapolate into a general solution, is based on the observation that an abstract feature is selected if and only if at least one of its children is selected. Hence, the propositional formula $\bigvee_{1 \le i \le n} C_i$ computes the value of the abstract feature P, where C_i are the children of P. We call this formula the *definition* of P.

Given definitions, our solution is to replace every occurrence of an abstract feature by its definition. Since a definition might contain an abstract feature itself, we iterate this process until the resulting propositional formula only contains concrete features.

3.5 Recap

With the algorithms and ideas that we have outlined, it is possible to classify edits that transform one feature model into another as being a refactoring, a specialization, a generalization, or an arbitrary edit. Our approach avoids an exponential explosion of CNF clauses that has has been a stumbling block in prior work; we use several simplifications and break reasoning into smaller steps. Furthermore, we developed extensions to support the addition and removal of concrete features and the substitution of abstract features to cover all possible edits on feature models.

Because we have reduced the problem of reasoning about feature model edits to the satisfiability problem, the algorithm could still have an exponential run-time in the worst case. Therefore, in the next section, we analyze the run-time of our algorithm empirically and show that it scales well for common and even for very large feature models.

4 Evaluation

We implemented the algorithm presented in the previous sectionas part of the graphical feature model editor of FeatureIDE[3] [24]. In the editor, the user can freely modify a feature diagram via drag and drop or using built-in operations. After each change, the tool shows whether all edits performed since the feature model was last saved constitute a refactoring, a specialization, a generalization, or an arbitrary edit. It illustrates this result by providing an according example of added and removed products. For solving propositional formulas, we used the SAT solver Sat4J[4].

For an initial evaluation, we took existing feature models and manually modified them. We used several small and medium-sized feature models that were publicly available, e.g., FAME-DBMS [33] of Figure 1 with 21 features, decomposed Berkeley DB [23] with 38 features, decomposed Violet UML editor[5] with 101 features, e-Shop [29] with 287 features, different versions of a graph product line [27] with 25 to 64 features, and several AHEAD tools [5] with 24 to 52 features. Additionally, we checked all feature models depicted in the proceedings of the *International Conference of Software Product Lines (SPLC)* 2007. Our tool worked as expected, and in every case, for every edit, we found that the response time for computing relationships between before and after-edited models was essentially instantaneous. This challenged us to evaluate how scaling the size of the models would affect performance.

Even though there are reports from industry of feature models with hundreds or thousands of features [26, 36], authors typically publish only a small excerpt of their feature models. We did not find large feature models for a thorough evaluation. Therefore, we decided to perform an experiment using randomly generated feature models with different characteristics.

[3]FeatureIDE is open source and available at: `http://www.fosd.de/featureide/`

[4]`http://www.sat4j.org`

[5]`http://sf.net/projects/violet/`

4.1 Experimental Setup

We randomly generated feature models and performed random edits on them. In this way, we can parametrically control the size of feature models and the number and kind of edits which allows a thorough run-time evaluation.

Independent variables in our experiment are (a) the number of features in a feature model, (b) number of edits, and (c) kind of edits. As a dependent variable we measured the time needed to determine the classification. To reduce fluctuations in the dependent variable caused by random generation, we performed 200 iterations for each configuration of independent variables, i.e., we generated 200 random feature models with the same parameters and each performed the same number of random edits of the same kind. Extraneous variables like hardware are kept constant by performing all measurements on the same Windows XP lab PC with 2.4 GHz and 2 GB RAM.

In the following, we describe the algorithms for generating feature models and edits on them.

Feature model generation. Our algorithm to randomly generate feature models of size n works as follows: Starting with a single root node, it runs several iterations. In each iteration, an existing node without children is randomly selected, and one to ten (random amount) of child nodes are added. Those child nodes are connected either by *And*- (50 % probability), *Or*- (25 % probability) or *Alternative*-group (25 % probability). Children in an *And*-group are optional by a 50 % chance. This iteration is continued until the feature model has n features. All features with children are considered abstract.

Additionally to the tree structure, we also generate cross-tree constraints. For every 10 features, we create one cross-tree constraint using the following algorithm: A constraint is generated consisting of two to five (random amount) variables or negated variables, connected randomly by $\neg, \wedge, \vee, \Rightarrow$, or $\Leftrightarrow$. Finally, we discard all feature models that do not have a single valid configuration (by unfortunate choice of cross-tree constraints) and repeat the entire process until the appropriate number of models are generated.

The parameters used in this generation algorithm (maximum number of children = 10; type of child group = (50 %, 25 %, 25 %); optional child = 50 %; number of cross-tree constraints = $0.1 * n$; variables in cross-tree constraints = 2–5) are fixed for in the entire experiment. They are backed up by a survey of all feature models we had available (see

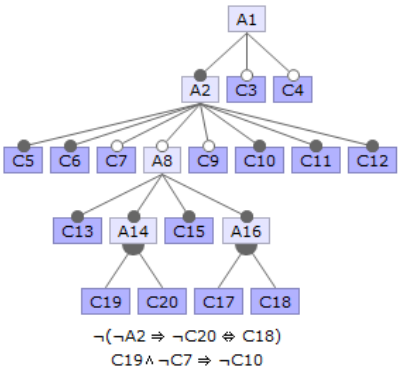


Figure 5. Random Generated Feature Model

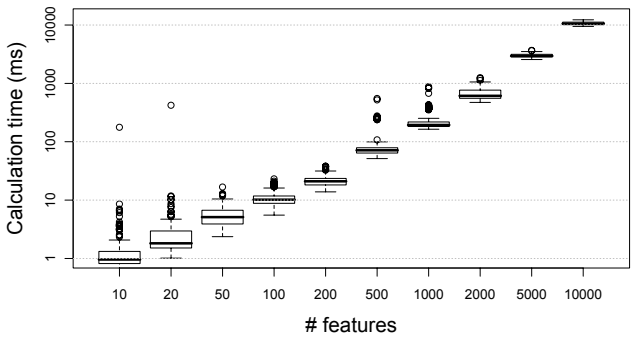


Figure 6. Calculation Time in Milliseconds for 10 Random Edits

above). The surveyed models were very different (e.g., some use almost only *And*-groups while others use almost only *Alternative*-groups), but these percentages represent a rough average. In this way, our generated feature models are statistically similar to the ones that we have found. All 2000 generated feature models that we used in our experiment are available on the FeatureIDE website to enable comparative studies. In Figure 5, we show one generated feature model for $n = 20$.

Feature model edit generation. Edits to a feature model are also generated randomly. Our generator takes a feature model and the number of edits as input. Furthermore, the edits can be limited to known refactorings or generalizations, which is useful to evaluate whether the performance depends on the kind of edits.

As refactorings and generalizations we implemented the 16 operations presented by Alves et al. [1], e.g., 'pull up node', 'add optional node', or 'collapse optional and or'. For arbitrary edits we implemented the following operations: (a) create or delete a feature without children; (b) change the type of a group, i.e., to *And*-, *Or*, or *Alternative*-group that is different from the type before; (c) make a mandatory feature optional or vice versa; (d) create a new constraint with the above defined parameters or delete a constraint. Finally, (e) move one concrete feature to a new parent.

These edits cover both, a sound sets of operations and operations a user could make in our editor (drag 'n' drop, double click, etc). When an edit is applied to a feature model, one of the available operations is randomly selected.

4.2 Experimental Results

Number of Features. First, we measured how calculation time scales as feature models increase in size. Therefore, we varied the size of the generated feature model between 10 and 10 000 features (which is significantly larger than all reported feature models in practice we came across). For each feature model, we performed 10 random edits (not restricted to refactorings or generalizations). As described above, we performed 200 repetitions for each model size, as there is a fluctuation of calculation times depending on the randomly generated models (with the same size) and the randomly performed edits. In Figure 6, we show the result of this measurements as boxplots with a logarithmic scale.[6]

For small feature models ($<$ 200 features) the average calculation time is less than 0.1 second (standard deviation less than 0.03 seconds, worst measured case was 0.4 seconds). For large feature models (1000 features) the average calculation time is 0.2 seconds (standard deviation 0.1 second, worst measured case 0.9 seconds). Only for very large feature models with up to 10 000 features, the calculation requires several seconds (mean 11 seconds, standard deviation 0.7 seconds, worst measured case 12 seconds).

Kind of Edits. Next, we performed the same measurement varying the feature model size from 10 to 10 000, but now we distinguished between different possible edits. This should determine whether changes that make arbitrary edits are more difficult to detect than those selected from Alves' set of sound operations. We distinguish between three possible kinds of edits: refactorings from the catalog of sound operations, generalizations from the same catalog, and arbitrary edits as described above. Specializations are not considered as they are the inverse of generalizations. Again we applied 10 random edits of this classification (i.e., 10 refactorings, or 10 generalizations, or 10 arbitrary edits) to each feature model.

In Figure 7, we show the results of our measurement (mean value of 200 repetitions for each combination of feature model size and type of edits). There is no significant

[6] A box plot is a common form to depict groups of numerical data and their dispersion. It plots the median as thick line and the quartiles as thin line, so that 50 % of all measurements are inside the box. Values that strongly diviate from the median are outlies and drawn as separate dots.

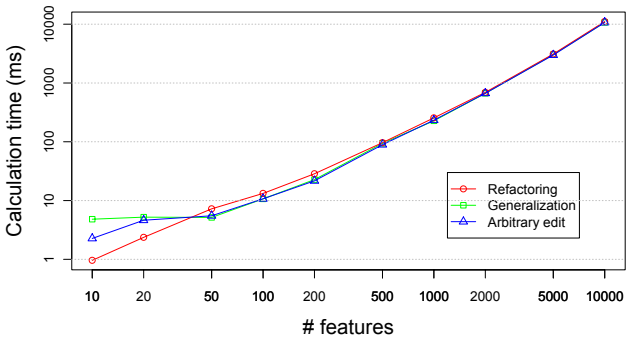


Figure 7. Calculation Time in Milliseconds for 10 Random Edits per Kind of Edit

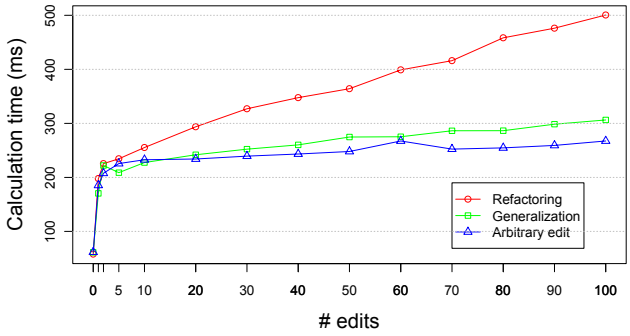


Figure 8. Calculation Time vs. Number of Edits (Feature Model Size = 1000)

difference between the three classifications refactoring, generalization, and arbitrary edits. The reason is that for 10 edits the time for initializing the solver often takes more time than verifying particular edits. The predicate that we supply to each invocation of a SAT solver is largely the size of $P(f)$, the original model. So as the size of the feature model increases, we would expect to see a (small) increase in computation size. As we will see in the next experiment, 10 edits turns out not to enough to distinguish computation times for refactorings, generalizations, or arbitrary edits.

Number of Edits. Finally, we measured the calculation time while varying the number of edits for a fixed feature model size of 1000 features. In this way, we can determine whether more edits require longer calculation times. We again distinguish between the three classifications refactoring, generalization (which includes specialization), and arbitrary edits.

The results (mean value of 200 iterations each) are shown in Figure 8. Additionally, we measured the mean calculation time for one edit (0.18 seconds), two edits (0.22 seconds), and five edits (0.23 seconds). For any number of edits, arbitrary edits and generalizations are calculated in approximately 0.3 seconds, almost independent of the number of edits. Only refactorings require longer, but still less than 0.5 seconds even for 100 edits in a feature model with 1000 features.

Although these almost constant times appear counter-intuitive at first, there is a simple explanation: The algorithm aborts as soon as an added and a deleted product is found (cf. Sec. 3.2). Since almost all arbitrary edits add or delete products, the algorithm terminates fastest for these inputs. Detecting refactorings is more time-consuming because all changed clauses need to be verified and the algorithm cannot stop earlier. But still it increases roughly linearly with the number of edits. Generalizations are in between as an added product can be found very fast and the nature of generalization edits usually simplifies the remaining formula c to check that no product was removed.

As special cases, we also measured identical feature models (0 edits) and compared completely independently generated feature models with the same feature names (which is roughly similar to an infinite number of edits and can be regarded as upper bound). Identical feature models with 1000 features are detected instantaneously, on average in 0.06 seconds. Completely independent feature models are classified in 0.7 seconds.

4.3 Discussion & Threats to Validity

Experiments show that our algorithm works efficiently even for large feature models. Mostly independent from the kind and number of edits, the algorithm can determine the classification of an edit in less than 0.1 seconds for small models (< 200 features), in less than 0.5 seconds for large models (< 1000 features) and even for very large feature models (up to 10 000 features) in few seconds.

This comfortably allows an implementation in our graphical editor that shows the classification on the fly for every change. It is calculated in a background thread, whose result is shown almost instantly. For very large feature models, a calculation time of 11 seconds (worst case 12 in our experiment) seems acceptable, given that such feature models are probably so complex that automatic reasoning for edits would be appreciated and a few seconds wait (for which we can even offer a progress bar) is tolerable.

Threats to internal validity are influences that can affect the calculation time that have not been considered. To avoid influence of computing power, all calculations have been performed on the same PC. We cannot guarantee that computation time depends on certain shapes of a feature model, or certain kinds of edits. However, to avoid misleading effects of specially-shaped feature models, we generated feature models automatically that statistically resemble known fea-

ture models, and repeated each measurement 200 times with freshly generated models. To avoid effects of certain edits, we randomly selected from a pool of edits that have been suggested in prior work.

Threats to external validity are conditions that limit our ability to generalize the results of our experiment to industrial practice. Our prior concern is whether generated feature models and edits are representative of industrial usage. First, we generated feature models with the described algorithm and parameters, and confirmed that they align well with those feature models we acquired from our own research, and publications in the software product line community. However, as we mentioned earlier, these models are typically small, and published feature models usually reveal excerpts from a larger model. Additionally, we tested our algorithm with feature models generated by others [29] and found no significant difference in computation times. Second, we applied edits from the catalog of feature model refactorings of Alves et al. [1] and added some edits that are possible using our editor (e.g., drag 'n' drop to move features or double click to change group type). We cannot guarantee that these edits are complete and typical in practice, however all reasonable edits we wanted to perform in our manual experiments could be performed with one or a sequence of these edits. Additionally, we also measured that comparing two feature models that were generated independently (which can be considered as a result of a worst case sequence of edits) performs within reasonable range.

5 Related Work

Operations on feature models have been classified in recent research on SPLs. Specializations were introduced by Czarnecki et al. for the process of deriving configurations of a feature model [13, 14, 25]. Specializations on cardinality-based feature models were shown, that result in feature models where some products are deleted. Specializations are defined formally by Janota and Kiniry [20], but both feature models have to be defined on the same set of features. We allow feature models also to be defined on different sets of features. For instance, we can remove an optional feature to achieve a specialization or remove a dead feature which is a refactoring.

Alves et al. proposed refactorings and generalizations [1]. They discussed several operations that maintain the set of products which they call *bi-refactorings* and operations that add new products which they call *refactorings*. In this paper, we decided to use the terms refactoring and generalization instead of bi-refactoring and refactoring, as a generalization is the inverse of a specialization, and a refactoring (as defined by Fowler [18]) should improve the feature model without modifying its 'behavior', i.e., the possible set of products. The main disadvantages of using sound operations is that designers have limited possibilities to alter feature models and that all edits that convert one model into another one must be known. For example, two feature models designed by different domain engineers could be extraordinarily difficult to compare.

While our paper is about alteration of feature models, automated analysis of feature models focuses on properties of a feature model, e.g., if a feature model contains at least one product or how many valid configurations it describes [28, 9, 15, 7, 6]. We see our work as yet another extension of this line of research, namely computing properties of (or relationships among) feature models. Benavides et al. analyzed the performance of CSP, SAT and BDD solvers in finding valid configuration given a feature model [8]. Their main result is that BDDs are faster than CSP or SAT solvers, but with a ten times higher memory usage. In particular, they state that BDDs are much faster in counting all solutions. Computation speed in our work is not a paramount problem. It might be when feature models become extremely large, and BDDs, rather than SAT solvers, may be a better underlying tool.

We performed reasoning purely on feature models. Some researchers also investigated relationships to other models or code artifacts. For example, Thaker et al. presented the safe composition of product lines [38]. The challenge is to ensure that every product (program) that is defined by a feature model is type safe. A similar form of type-checking an entire SPL on the background of a feature model has been formalized by Kästner and Apel [22]. Furthermore, Metzger et al. formalized *orthogonal variability models* as an instrument for product management and feature models to express software variability [30]. They proposed the use of SAT solvers for reasoning, e.g., to check whether the all product line members of the variability model are realizable according to the feature model.

It recently occurred to us that the problem of reasoning about feature model edits may be recast as a specific subproblem of language containment, whether one language contains the sentences of another [11]. This problem is computationally difficult, belonging to PSPACE.

6 Conclusion

Feature models are a fundamental representation of software product lines. They compactly define the set of legal combinations of features, where each combination corresponds to a product in a product line. As the number of optional features increases linearly, the number of possible products in a product line can increase exponentially. This means that small changes to a feature model can have a significant impact on the membership of a product line. Prior to our work, there was little or no tool support for helping product line designers gauge the impact of their changes to a

feature model. As a consequence, inadvertently adding and deleting products was always a possibility, and discovering errors was fortuitous. This no longer has to be the case.

We presented an efficient algorithm to determine the relationship between two feature models, where an edit of a feature model is classified as a specialization, refactoring, generalization, or none of these. Our algorithm overcomes four limitations of earlier attempts. First, it supports arbitrary edits on feature models, even arbitrary changes to cross-tree constraints. Second, it supports addition or deletion of features. Third, the algorithm provides examples to explain computed classifications. Fourth, due to several optimizations it scales even to very large feature models with thousands of features.

Future work could generalize our analysis to other variability models, e.g., feature models with attributes [6]. Another possibility is to make finer distinctions than our refactoring-generalization-specialization-arbitrary classifications. For example, suppose a feature X is split into features X1 and X2 (where the use of X1 implies the use of X2, and vice versa). Clearly, the set of products in a product line have not been changed by this edit, but we would classify the change as an arbitrary edit. A compact representation of all added and removed products would be an extension to our proposed reasoning that seems desirable for tool support.

We believe our work is a practical step forward in improving tools for managing the evolution of feature models.

Acknowledgment. The authors thank Daniel Le Berre for his help with SAT4J, and for fine-tuning it for our purpose. We also thank David Benavides for his comments on an earlier draft of our paper. Thüm's work was supported by DFG project #SA 465/32-1 and the Metop GmbH. Batory's work was supported by NSF's Science of Design Project #CCF-0438786 and #CCF-0724979.

References

[1] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. J. P. de Lucena. Refactoring Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering*, pages 201–210. ACM, 2006.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[3] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Software Product Line Conference*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.

[4] D. Batory, R. E. Lopez-Herrejon, and J.-P. Martin. Generating Product-Lines of Product-Families. In *Proc. Int'l Conf. Automated Software Engineering*, pages 81–92. IEEE Computer Society, 2002.

[5] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Softw. Eng.*, 30(6):355–371, 2004.

[6] D. Benavides. *On the Automated Analysis of Software Product Lines Using Feature Models - A Framework for Developing Automated Tool Support*. PhD thesis, University of Seville, Spain, 2007.

[7] D. Benavides, A. Ruiz-Corts, P. Trinidad, and S. Segura. A Survey on the Automated Analyses of Feture Models. *Jornadas de Ingeniera del Software y Bases de Datos*, pages 367–376, 2006.

[8] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. A First Step Towards a Framework for the Automated Analysis of Feature Models. In *Proc. Managing Variability for Software Product Lines: Working With Variability Mechanisms*, pages 39–47, 2006.

[9] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Using Constraint Programming to Reason on Feature Models. In *Proc. Int'l Conf. on Software Engineering and Knowledge Engineering*, pages 677–682, 2005.

[10] H. K. Buning and T. Letterman. *Propositional Logic: Deduction and Algorithms*. Cambridge University Press, 1999.

[11] E. M. Clarke, I. A. Draghicescu, and R. P. Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. In *Information Processing Letters, Vol. 46*, pages 301–308, 1993.

[12] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley, 2000.

[13] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing Cardinality-Based Feature Models and their Specialization. *Software Process: Improvement and Practice*, 10:7–29, 2005.

[14] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged Configuration through Specialization and Multi-Level Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[15] K. Czarnecki and C. H. P. Kim. Cardinality-Based Feature Modeling and Constraints: A Progress Report. In *Proc. of Int'l OOPSLA Workshop on Software Factories*, pages 16–20, 2005.

[16] K. Czarnecki and A. Wasowski. Feature Diagrams and Logics: There and Back Again. In *Proc. Int'l Software Product Line Conference*, pages 23–34. IEEE Computer Society, 2007.

[17] K. D. Forbus and J. de Kleer. *Building problem solvers*. MIT Press, 1993.

[18] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[19] M. L. Griss, J. Favaro, and M. d' Alessandro. Integrating Feature Modeling with the RSEB. In *Proc. Int'l Conf. on Software Reuse*, pages 76–85, 1998.

[20] M. Janota and J. Kiniry. Reasoning about Feature Models in Higher-Order Logic. In *Proc. Int'l Software Product Line Conference*, pages 13–22. IEEE Computer Society, 2007.

[21] 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, Software Engineering Institute, 1990.

[22] C. Kästner and S. Apel. Type-checking Software Product Lines - A Formal Approach. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*, pages 258–267. IEEE Computer Society, 2008.

[23] C. Kästner, S. Apel, and D. Batory. A case study implementing features using AspectJ. In *Proc. Int'l Software Product Line Conference*, pages 223–232. IEEE Computer Society, 2007.

[24] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. FeatureIDE: Tool Framework for Feature-Oriented Software Development. In *Proc. Int'l Conf. on Software Engineering*. IEEE, 2009. Formal Demonstration.

[25] C. H. P. Kim and K. Czarnecki. Synchronizing Cardinality-Based Feature Models and Their Specializations. In *Proc. European Conf. on Model Driven Architecture - Foundations and Applications*, volume 3748 of *LNCS*, pages 331–348. Springer, 2005.

[26] F. Loesch and E. Ploedereder. Optimization of Variability in Software Product Lines. In *Proc. Int'l Software Product Line Conference*, pages 161–170. IEEE Computer Society, 2007.

[27] R. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In *Proc. Int'l Conf. Generative and Component-Based Software Engineering*, pages 10–24. Springer, 2001.

[28] M. Mannion. Using First-Order Logic for Product Line Model Validation. In *Proc. Int'l Software Product Line Conference*, volume 2379 of *LNCS*, pages 176–187. Springer, 2002.

[29] M. Mendonca, A. Wasowski, K. Czarnecki, and D. Cowan. Efficient Compilation Techniques for Large Scale Feature Models. In *Proc. Int'l Conf. Generative Programming and Component Engineering*, pages 13–22. ACM, 2008.

[30] A. Metzger, K. Pohl, P. Heymans, P.-Y. Schobbens, and G. Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *Proc. Int'l Conf. Requirements Engineering*, pages 243–253. IEEE Computer Society, 2007.

[31] J. Nielsen. *Usability Engineering*. Morgan Kaufmann, 1994.

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

[33] M. Rosenmüller et al. FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems. In *Proc. EDBT Workshop on Software Engineering for Tailor-made Data Management*, pages 1–6. ACM, 2008.

[34] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic Semantics of Feature Diagrams. *Computer Networks*, 51(2):456–479, 2007.

[35] J. Spolsky. *User Interface Design for Programmers*. Springer, 2001.

[36] M. Steger, C. Tischer, B. Boss, A. Müller, O. Pertler, W. Stolz, and S. Ferber. Introducing PLA at Bosch Gasoline Systems: Experiences and Practices. In *Proc. Int'l Software Product Line Conference*, volume 3154 of *LNCS*, pages 34–50. Springer, 2004.

[37] J. Sun, H. Zhang, Y. F. Li, and H. Wang. Formal Semantics and Verification for Feature Modeling. In *Proc. Int'l Conf. on Engineering of Complex Computer Systems*, pages 303–312. IEEE Computer Society, 2005.

[38] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering*, pages 95–104. ACM, 2007.

[39] T. Thüm. Reasoning about Feature Model Edits. Bachelor's thesis, University of Magdeburg, Germany, 2008.

[40] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated Diagnosis of Product-line Configuration Errors in Feature Models. In *Proc. Int'l Software Product Line Conference*. IEEE Computer Society, 2008.