An Efficient Algorithm for Feature-Model Slicing

Sebastian Krieter
University of Magdeburg
Magdeburg, Germany

Reimar Schröter
University of Magdeburg
Magdeburg, Germany

Thomas Thüm
TU Braunschweig
Brunswick, Germany

Gunter Saake
University of Magdeburg
Magdeburg, Germany

ABSTRACT

Feature models are a well-known concept to represent variability in software product lines. A feature model defines all features of a product line and their corresponding interdependencies. During software product line engineering, there arise situations that require the removal of certain features from a feature model such as feature-model evolution, information hiding, and feature-model analyses. However, crude deletion of features in a model typically has undesirable effects on interdependencies of the remaining features. Moreover, current algorithms for dependency-preserving feature removal (known as feature-model slicing) do not perform well when removing a high number of features from large feature models. Therefore, we propose an efficient algorithm for feature-model slicing based on logical resolution and CNF minimization.

Keywords

Feature-Model Evolution, Feature-Model Analyses, Software Product Lines

1. INTRODUCTION

Today, industrial software systems are often based on a high number of variable assets, called features. Feature models are commonly used to handle and describe existing dependencies between these features. As prominent examples like the Linux kernel and other case studies [5, 15] show, feature models can become very large with more than 10,000 features. Due to constant development, feature models evolve over time and their set of features and corresponding interdependencies change. Thus, it is not surprising that there are numerous applications that require the removal of one or more features from a feature model. For instance, during the evolution of feature models, features can become obsolete and have to be removed or are replaced by other features. Besides removing features under evolution, there are other applications, such as removing abstract features [17], the generation of feature-model interfaces [14], and decomposition of feature models [1]. However, when removing a feature, existing feature dependencies often need to be preserved. As example, consider the removal of feature B from the feature model given by the following propositional formula $(A \Rightarrow B) \wedge (B \Rightarrow (C \wedge D))$. In this case, the intended result is $A \Rightarrow (C \wedge D)$ as it maintains the dependencies between A, C, and D. However, a crude elimination of the variable from the formula leads to unwanted results. For example, a syntactical deletion of B from the formula leads to the result $(\neg A \wedge C \wedge D)$, which is obviously wrong.

To resolve this issue, there exist algorithms that remove a feature from a feature model without changing the dependencies between other features. This technique is also known as feature-model slicing [2]. However, when removing a large number of features, existing algorithms still require an insufficient amount of time. For instance, in our previous investigations of feature models, in which we remove more than 1,000 features, FeatureIDE's algorithm for removing abstract features did not scale well [14]. Therefore, we investigate existing algorithms and improve certain parts in order to increase the performance.

Similar to feature-model analyses that, for instance, can be used to identify feature-model inconsistency, the problem of feature-model slicing is NP-hard (e.g., using feature-model slicing, the void analysis is trivially solvable by removing all features from a feature model). However, previous investigations show that in the domain of feature models the analysis problem is nonetheless solvable in an adequate amount of time [12]. This motivates us to optimize feature-model slicing and we aim to find heuristics that enable a fast performance for real-world feature models. As result of our investigations, we propose an algorithm that is based on multiple satisfiability tests and logical resolution. In detail, we propose an improved algorithm for feature-model slicing based on

- an exchange of an existential quantification strategy to logical resolution, and
- a new heuristic to optimize the order of features that we want to remove.

The paper is structured as follows. In Section 2, we provide relevant background knowledge of feature models and their representations. In Section 3, we describe our algorithm using pseudo code. Afterwards, in Section 4, we discuss related work and present our conclusion with future work in Section 5.

2. FEATURE MODELS AND SLICING

Feature models define a set of features and specify dependencies between them [10]. We now briefly introduce two feature-model representations, feature diagrams and propositional formulas, which we use for our concept. For each representation, we use our running example, a graph product line, as illustration. In addition to our reflection on feature-model representations, we also consider the state-of-the-art procedure for feature-model slicing based on existential quantification.

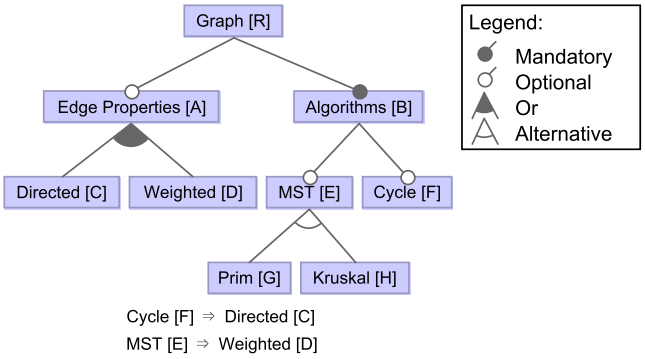

Figure 1: Feature model of a graph product line.

2.1 Feature Diagrams

Feature diagrams are graphical representations of feature models [10]. Compared to textual representations of feature models, feature diagrams are easier to read and to manipulate by developers. A feature diagram organizes features in a tree structure and thereby specifies their interdependencies. Features are represented by nodes and their dependencies are derived from the edges connected to them. Each feature implies its direct parent feature. Additionally, features can be mandatory, which means that they are required by their parent feature. Furthermore, features with the same parent can be organized in a group. Common groups are or-groups and alternative-groups. In an or-group a parent feature requires at least one of its children, whereas in an alternative group exactly one feature is required. Since dependencies between features in different subtrees cannot be represented by the tree structure alone, feature diagrams allow developers to add additional cross-tree constraints. One of the most common representations for cross-tree constraints are propositional formulas, in which the additional dependencies can be described using logical operations.

In Figure 1, we illustrate the representation as feature diagram using a graph product line. Depending on the algorithm that we want to support, a graph library needs to ensure special properties of edges. All graph libraries contain the root feature *Graph*, and the feature *Algorithms* (mandatory). Additionally, each graph library can provide some edge properties and certain algorithms on the graph structure (optional features). Possible properties for edges are *Directed* and *Weighted* that are located in an or group so that one of the properties needs to be selected if additional properties are desired. Besides additional properties of edges, each graph can provide different algorithms. In detail, it is possible to select the optional feature *MST* to identify minimal spanning trees or the feature *Cycle* to identify cycles in the graph. Since different algorithms exist to compute minimal spanning trees (MSTs), a user needs to choose a specific algorithm if this feature is selected. Therefore, we use an alternative group to force a decision between the algorithms of *Prim* and *Kruskal*. Depending on the selection of desired algorithms, different properties of edges are necessary. Therefore, we add cross-tree constraints to ensure a valid feature combination. For instance, the constraint $Cycle \rightarrow Directed$ ensures that all products, in which feature *Cycle* is included, also feature *Directed* exist.

Besides the described representation of feature diagrams, numerous extensions exist to enrich the expressiveness of feature diagrams (cf. the survey of Benavides et al. to get an overview [4]). One of these extensions are abstract features that are features without implementation artifacts and thus do not contribute to the final software product [17]. Thüm et al. proposed the concept of abstract features to enable a better organization within the tree structure and to distinguish already implemented from future features [17]. Since abstract features do not contribute to a product line's product, they need to be eliminated if the number of possible resulting products needs to be calculated. Therefore, abstract features are not only an extension of feature diagrams but also an application scenario in which feature-model slicing is needed.

Root	$R \wedge$
Child-Parent	$(A \Rightarrow R) \wedge (B \Rightarrow R) \wedge (C \Rightarrow A) \wedge$
	$(D \Rightarrow A) \wedge (E \Rightarrow B) \wedge (F \Rightarrow B) \wedge$
	$(G \Rightarrow E) \wedge (H \Rightarrow E) \wedge$
Mandatory	$(R \Rightarrow B) \wedge$
Or group	$(A \Rightarrow (C \vee D)) \wedge$
Alt. group	$(E \Rightarrow ((G \vee H) \wedge \neg(G \wedge H))) \wedge$
Constraints	$(F \Rightarrow C) \wedge (E \Rightarrow D)$

Figure 2: Propositional formula of the graph product line.

2.2 Propositional Formulas

Another useful representation of feature models are propositional formulas [3]. This representation is used in many analyses on product lines. For instance, to ensure a correct specification of feature dependencies, to present statistics on features models, or as base to ensure a correct implementation or behavior of product line's products [4, 16]. The representation of a feature model as a propositional formula is mainly used for analysis as it allows a reduction to the well-known satisfiability problem. In detail, features are represented by logical variables and their interdependencies are expressed using logical operators such as negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\Rightarrow$), and equivalence ($\Leftrightarrow$). Using our running example of the graph product line, we illustrate the representation as propositional formula based on the abbreviation characters of each feature in Figure 2. Therefore, we transform all existing dependencies into a logical representation.

For the formal description of our algorithm, we use the set notation of a propositional formula in *conjunctive normal form (CNF)*. A CNF consists of a conjunction of clauses, which consist of a disjunction of single literals. A literal is a variable in either its positive or negative form. In set notation, a feature model's CNF representation consists of a set of clauses $\mathcal{C} = \{c_1, c_2, ..., c_m\}$ where $m \in \mathbb{N}$ is the number of clauses. Each clause c_i is a subset of the set of literals $c_i \subseteq \mathcal{L} = \{l_1, ..., l_n, \neg l_1, ..., \neg l_n\}$ where $n \in \mathbb{N}$ is the number of features.

2.3 State-of-the-Art Feature-Model Slicing

A possible approach for feature-model slicing is existential quantification of a propositional variable [2, 17]. To illustrate the main idea, we use the small formula of our introduction section. In detail, we consider the formula $(A \Rightarrow B) \wedge (B \Rightarrow (C \wedge D))$ and want to remove the variable B. The idea of existential quantification is to replace all oc-

currences of the variable B in the formula with both possible assignments (*true (T)* and *false (F)*). Therefore, the formula is duplicated and combined with a logical or, whereas the variable is replaced with *true* on the one side and with *false* on the other side. Afterwards, it is possible to simplify the formula so that the performance of further variable removals can be improved. The following steps present an overview on how to remove variable B of the formula:

(1) Formula: $(A \Rightarrow B) \ \wedge \ (B \Rightarrow (C \wedge D))$
(2) CNF: $(\neg A \vee B) \wedge (\neg B \vee C) \wedge (\neg B \vee D)$
(3) Replace: $((\neg A \vee \mathbf{F}) \wedge (\neg \mathbf{F} \vee C) \wedge (\neg \mathbf{F} \vee D)) \vee ((\neg A \vee \mathbf{T}) \wedge (\neg \mathbf{T} \vee C) \wedge (\neg \mathbf{T} \vee D))$
(4) Simplify: $(\neg A) \vee (C \wedge D)$
(5) CNF: $(\neg A \vee C) \wedge (\neg A \vee D)$

3. FEATURE-MODEL SLICING

In this section, we propose our new algorithm for feature-model slicing based on logical resolution. Before we start to present details of the algorithm, we give an overview about the main idea of logical resolution and the effect on feature-model slicing. Afterwards, we present the base algorithm in pseudo code and illustrate its behavior using a small example. Based on this knowledge, we describe the algorithms details such as the heuristics for determining the feature order and the method for simplifying the resulting formula.

Another method of removing variables from propositional formulas is the application of logical resolution. The resolution rule derives a new clause c_{new} called *resolvent* from two other clauses $c_1, c_2 \in \mathcal{C}$ if there exists a literal l such that $l \in c_1$ and $\neg l \in c_2$. The resolvent is constructed by combining both clauses and removing l (i.e., $c_{new} = (c_1 \cup c_2) \setminus \{l, \neg l\}$). This resolvent represents a transitive dependency between c_1 and c_2. The application of resolution with respect to the variable that should be removed and a subsequent removal of all clauses that contain the variable lead to the desired result. In fact, resolution is a direct consequence from existential quantification and the subsequent transformation into CNF. After the replacement step, during a variable's removal through existential quantification, there exist two CNFs connected by a disjunction. All clauses that contain either *true* or *not false* are tautologies and, thus, removed from the respective CNF. Therefore, all clauses that previously contained the removed variable in its positive form are now present in one CNF, whereas all clauses that contained the variable's negative form are present in the other CNF. To reconstruct the overall CNF structure, the clauses from both CNFs are combined pairwise. Thus, logical resolution yields the same result as existential quantification, while additionally keeping the formula in CNF.

Again, we consider our formula example of the introduction section $(A \Rightarrow B) \ \wedge \ (B \Rightarrow (C \wedge D))$ to illustrate the mechanism of logical resolution. Here, we want to remove the variable B. Thus, when we apply resolution to the CNF of our formula (i.e., $(\neg A \vee B) \wedge (\neg B \vee C) \wedge (\neg B \vee D)$), we get the resolvents $(\neg A \vee C)$ and $(\neg A \vee D)$ that we add to the input formula. If we then delete all clauses used for the resolution we get the desired CNF. We exemplify the necessary steps as follows:

(1) Formula: $(A \Rightarrow B) \ \wedge \ (B \Rightarrow (C \wedge D))$
(2) CNF: $(\neg A \vee B) \wedge (\neg B \vee C) \wedge (\neg B \vee D)$
(3) Resolution: $(\neg A \vee B) \wedge (\neg B \vee C) \wedge (\neg B \vee D)$ $(\neg A \vee C) \wedge (\neg A \vee D)$
(4) Clause Deletion: $(\neg A \vee C) \wedge (\neg A \vee D)$

3.1 Slicing Algorithm Overview

Now, we take a look into the details of our algorithm. The algorithm takes a feature model and a set of features as input and returns a newly constructed formula in CNF representing the sliced feature model, which contains no variable from the given set. Therefore, we use an iterative process that removes a variable in two phases. First, the CNF is simplified to remove redundancies, e.g., clauses that are redundant because of the resolution of a previous iteration. Second, resolution is performed with respect to the variable. Of course, as necessary for feature-model slicing, the algorithm ensures to keep all dependencies between all other variables in the formula.

In Algorithm 1, we present the pseudo code of the approach. In our code, we use the following variable notation:

v	$\rightarrow$	variable name
$\mathcal{V}$	$\rightarrow$	set of variable names
l	$\rightarrow$	literal
c	$\rightarrow$	set of literals (clause)
$\mathcal{C}$	$\rightarrow$	set of clauses (CNF)
$\mathcal{FM}$	$\rightarrow$	feature model

Algorithm 1 Main algorithm - Iteratively removes all variables in $\mathcal{V}_{remove}$ from $\mathcal{FM}$

```
1: function removeVariables(FM, V_remove)
2:   C_cnf ← getCNF(FM)
3:   C_dirty ← ∅, C_clean ← ∅
4:   for all c1 ∈ C_cnf do
5:     classify(c1, V_remove, C_dirty, C_clean)
6:   end for
7:   while V_remove ≠ ∅ ∧ C_dirty ≠ ∅ do
8:     removeRedundant(C_clean)
9:     removeRedundant(C_dirty)
10:    v1 ← next(FM, V_remove)
11:    V_remove ← V_remove \ {v1}
12:    resolution(v1, V_remove, C_dirty, C_clean)
13:  end while
14:  return C_clean
15: end function
```

Our main algorithm has two input parameters, the feature model $\mathcal{FM}$ and a set of variables that should be removed ($\mathcal{V}_{remove}$). At first, $\mathcal{FM}$ is converted into CNF so that the feature model is represented by a set of clauses ($\mathcal{C}_{cnf}$). Then, all clauses of the given formula are divided into one of two sets, *dirty* or *clean* (cf. Line 3–6 and Algorithm 2). The dirty set contains all clauses that contain at least one variable from $\mathcal{V}_{remove}$. Consequently, the clean set contains all clauses in which no variable of $\mathcal{V}_{remove}$ exists. Next, the algorithm removes one variable at a time from the clauses in $\mathcal{C}_{dirty}$ by continuously processing all variables in the given variable set (cf. Line 7–13). When the algorithm is finished, the clean set contains all remaining clauses, whereas the dirty set is empty. Thus, the final formula is constructed by a conjunction of all clauses in the clean set.

Considering the details of each iteration of the main procedure (cf. Line 7–13), the algorithm simplifies the current

Algorithm 2 Tests whether the clause c_{new} contains a variable from $\mathcal{V}_{remove}$ and adds it to the corresponding set of clauses ($\mathcal{C}_{dirty}$ or $\mathcal{C}_{clean}$)

```
1: procedure CLASSIFY(c_new, V_remove, C_dirty, C_clean)
2:   dirty ← false
3:   for all l_1 ∈ c_new do
4:     if NAME(l_1) ∈ V_remove then
5:       dirty ← true
6:       break for
7:     end if
8:   end for
9:   if dirty then
10:    C_dirty ← C_dirty ∪ {c_new}
11:  else
12:    C_clean ← C_clean ∪ {c_new}
13:  end if
14: end procedure
```

CNF in $\mathcal{C}_{dirty}$ and $\mathcal{C}_{clean}$ by removing invalid and redundant clauses. The detection of redundancy depends on the specific strategy that is used. We present more insights to the different strategies in Section 3.3. After the CNF simplification, the `next` method returns a variable from $\mathcal{V}_{remove}$ (cf. Line 10) that is removed in this iteration (cf. Line 11). We describe the internal functionality of the `next` method in Section 3.2. For each variable, resolution is perform with respect to the clauses in set $\mathcal{C}_{dirty}$ (cf. Line 12).

In Algorithm 3, we show the pseudo code for resolution in more detail. The input parameters are the current variable v_1 that we want to remove, the set of all variables that we want to remove $\mathcal{V}_{remove}$, and the set of dirty $\mathcal{C}_{dirty}$ and clean clauses $\mathcal{C}_{clean}$. For each clause c_1 in the dirty set the algorithm checks if it contains the current variable v_1 (cf. Line 3). In this case, the algorithm removes c_1 from the dirty set and searches for clauses that contain the complement of v_1 in c_1 (cf. Lines 5–10). For each of these clauses c_2, the algorithm constructs a new combined clause (i.e., the resolvent) which is then again classified as clean or dirty dependent on its contained variables (cf. Lines 8, 9). After all clauses are processed, v_1 is no longer contained in any clause.

In the pseudo code, we did not specify the implementation for the functions `next` and `removeRedundant`. Therefore, our algorithm contains two major variation points. This allows us to use different strategies for both functionalities that influence the algorithm's performance. However, before we start to describe these details, we give a small example on how our algorithm proceeds with an input formula and a set of features that we want to remove.

The Graph Product Line Example

Using a smaller version of the graph product line (cf. Figure 3), we visualize the functionality of our algorithm. We execute our algorithm with the feature model $\mathcal{FM} = Graph$ and $\mathcal{V}_{remove} = \{B, D\}$ as input parameters. The algorithm starts with the transformation of the feature model into the conjunctive normal form $\mathcal{C}_{cnf} = \{\{R\}, \{\neg A, R\}, \{\neg B, R\}, \{\neg C, A\}, \{\neg D, A\}, \{\neg E, B\}, \{\neg F, B\}, \{\neg F, C\}, \{\neg E, D\}, \{\neg R, B\}, \{\neg A, C, D\}\}$. Using the CNF as additional information, we now show the intermediate results of the algorithm. Therefore, in Table 1, we depict the content of the sets of clean and dirty clauses. As first intermediate result

Algorithm 3 Performs resolution on clauses containing v_1

```
1: procedure RESOLUTION(v_1, V_remove, C_dirty, C_clean)
2:   for all c_1 ∈ C_dirty do
3:     l_1 ← LITERAL(c_1, v_1)
4:     if l_1 ≠ null then
5:       C_dirty ← C_dirty \ {c_1}
6:       for all c_2 ∈ C_dirty do
7:         if ¬l_1 ∈ c_2 then
8:           c_new ← (c_1 ∪ c_2) \ {l_1, ¬l_1}
9:           CLASSIFY(c_new, V_remove, C_dirty, C_clean)
10:        end if
11:      end for
12:    end if
13:  end for
14: end procedure
```

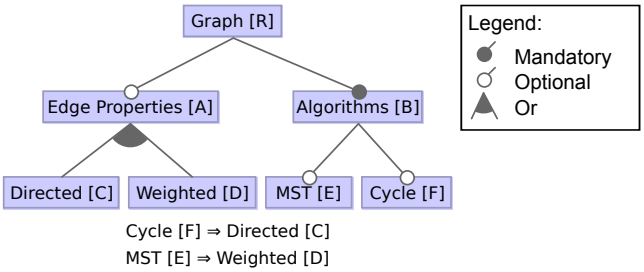

Figure 3: Small feature model of a graph product line.

(cf. #1), we depict the content of the dirty and clean set after their instantiation through the classification step of the algorithm. The resulting sets are: $\mathcal{C}_{dirty} = \{\{\neg\mathbf{B}, R\}, \{\neg\mathbf{D}, A\}, \{\neg E, \mathbf{B}\}, \{\neg F, \mathbf{B}\}, \{\neg E, \mathbf{D}\}, \{\neg R, \mathbf{B}\}, \{\neg A, C, \mathbf{D}\}\}$ and $\mathcal{C}_{clean} = \{\{R\}, \{\neg A, R\}, \{\neg C, A\}, \{\neg F, C\}\}$. Next, the algorithm tries to remove redundancies from both sets. It removes the clause $\{\neg B, R\}$ from the dirty set and the clause $\{\neg A, R\}$ from the clean set as they are subsumed from the clause $\{R\}$. The result is our second intermediate result (#2), in which the corresponding clauses are removed. Next, resolution is performed to all clauses in the dirty set that contain the current variable. In the first iteration, we assume that the variable D is chosen first. Thus resolution is

#	$\mathcal{C}_{dirty}$	$\mathcal{C}_{clean}$
1	$\{\{\neg\mathbf{B}, R\}, \{\neg\mathbf{D}, A\}, \{\neg E, \mathbf{B}\}, \{\neg F, \mathbf{B}\}, \{\neg E, \mathbf{D}\}, \{\neg R, \mathbf{B}\}, \{\neg A, C, \mathbf{D}\}\}$	$\{\{R\}, \{\neg A, R\}, \{\neg C, A\}, \{\neg F, C\}\}$
2	$\{\{\neg\mathbf{D}, A\}, \{\neg E, \mathbf{B}\}, \{\neg F, \mathbf{B}\}, \{\neg E, \mathbf{D}\}, \{\neg R, \mathbf{B}\}, \{\neg A, C, \mathbf{D}\}\}$	$\{\{R\}, \{\neg C, A\}, \{\neg F, C\}\}$
3	$\{\{\neg E, \mathbf{B}\}, \{\neg F, \mathbf{B}\}, \{\neg R, \mathbf{B}\}\}$	$\{\{R\}, \{\neg C, A\}, \{\neg F, C\}, \{\neg E, A\}, \{\neg A, A, C\}\}$
4	$\{\{\neg E, \mathbf{B}\}, \{\neg F, \mathbf{B}\}, \{\neg R, \mathbf{B}\}$	$\{\{R\}, \{\neg C, A\}, \{\neg F, C\}, \{\neg E, A\}\}$
5	$\emptyset$	$\{\{R\}, \{\neg C, A\}, \{\neg F, C\}, \{\neg E, A\}\}$

Table 1: Intermediate result of the dirty and clean set in the algorithm's example execution.

applied to the clauses $\{\neg D, A\}$, $\{\neg E, D\}$, and $\{\neg A, C, D\}$, which results in the resolvent $\{\neg E, A\}$ and $\{\neg A, A, C\}$. The original clauses are removed from the dirty set. The new clauses $\{\neg A, A, C\}$ and $\{\neg E, A\}$ are classified as clean as they contain no variable from $\mathcal{V}_{remove}$. The resulting set of clauses is represented in our third intermediate result, in which the variable D is completely removed (#3).

In the second iteration, the algorithm removes variable B. Again the algorithm tries to remove redundant clauses first and removes the clause $\{A, \neg A, C\}$ as it is a tautology. We depict the result as our fourth intermediate result (#4). Afterwards, the resolution with B finds no resolvent and, thus, the algorithm only removes the clauses $\{\neg E, B\}$, $\{\neg F, B\}$, and $\{\neg R, B\}$ that contain B from the dirty set (cf. #5). The dirty set is now empty, which means that all variables are removed. The remaining clauses in the clean set form the resulting CNF: $R \wedge (\neg C \vee A) \wedge (\neg F \vee C) \wedge (\neg E \vee A)$.

3.2 Feature Order

In Section 3.1, we stated that function `next` returns the next variable that should be removed. Finding a suitable order is crucial as it heavily influences the number of new clauses that are generated by the resolution. As there are $n!$ possible feature orders when removing n feature from a feature model, the computation of the optimal order is an expensive problem. Thus, the usage of a heuristic that at least specifies a good order is more feasible.

We propose the strategy *minimum clauses generation* that considers the number of new clauses that are generated in each resolution phase. Thus, the strategy directly aims to reduce the number of generated clauses during resolution. It is a greedy strategy that selects the best variable in each iteration. That is the variable, whose removal introduces the fewest new clauses to the CNF.

While the exact number of newly generated clauses is hard to compute beforehand (due to possible redundancy), we can easily determine an approximation by counting the number of clauses in which a given variable is contained. By multiplying the number of clauses that contain the variable in its positive form with the ones containing its negative form, we get the approximate number of clauses that would be generated by resolution. This estimated number is used as sorting criteria, whereas the variable with the lowest value will be removed next. Since new clauses are generated when a variable is removed, we have to update the comparative values before removing the next variable. Thus, this strategy is dynamically adapted in each iteration.

3.3 CNF Simplification

The main issue when using resolution to find the transitive dependencies in the CNF is that it introduces new clauses to the formula. As result, this can lead to an exponential growth of clauses in the dirty set and, thus, a bad overall performance when removing a large number of features. The function `removeRedundant` addresses this problem as most of the newly generated clauses contain no new information. Either because they can be derived from other clauses in the formula or because they are always *true*. For a high performance of the algorithm, it is important to minimize the number of clauses in the formula by remove all redundant clauses.

A first and straight-forward approach to remove clauses is to check whether a clause is a tautology, because it contains a variable and also its complement. More formally, if for a clause $c \in \mathcal{C}$ with $\{l, \neg l\} \subseteq c$ applies for some literal $l \in \mathcal{L}$ it is a tautology and thus, always evaluates to *true*. Therefore, the clause has no effect on complete formula and can be removed from $\mathcal{C}$. Since, it is possible to check this property for all clauses in linear time with respect to the number of clauses, the algorithm tests each new clause after every resolution phase and removes it if necessary.

Additionally, the function `removeRedundant` tries to detect redundancies between clauses and removes one or more of the responsible clauses. However, a complete check for redundancy is again an expensive task. Therefore, we consider three different levels of redundancy: A new clause c_{new} is redundant if

(a) $\exists c \in \mathcal{C} : c = c_{new}$ (Equivalence),

(b) $\exists c \in \mathcal{C} : c \subseteq c_{new}$ (Subsumption), or

(c) $\mathcal{C} \setminus \{c_{new}\} \vDash c_{new}$ (Derivation).

This classification arises from the computational effort required to check the property and number of detected redundant clauses. In general, checking whether there exists a clause equal to c_{new} can be done in less time, than checking whether c_{new} can be derived from other clauses. However, the check for equivalence finds less redundant clauses than the check for derivation, which is able to find all redundant clauses. In particular, all clauses that can be detected using a certain redundancy level can also be detected with any higher level. For example, given the clause set $\mathcal{C} = \{\{\neg A, B\}, \{\neg B, C\}\}$ and the new clause $c_{new_1} = \{A, \neg B, C\}$, c_{new_1} is not redundant with respect to (a), because there exists no equal clause in $\mathcal{C}$. However, c_{new_1} is subsumed by $\{\neg B, C\}$ and, thus, is redundant with respect to (b) and (c). Furthermore, the clause $c_{new_2} = \{\neg A, C\}$ has no equal clause in $\mathcal{C}$ and is not subsumed either, but can be derived from $\mathcal{C}$, which makes it redundant considering (c).

For the function `removeRedundant`, we use a combination of all mentioned redundancy levels in a certain order. By applying the more efficient checks first, we attempt to minimize the input for subsequent checks and, thus, decrease the overall time consumption. To specify the best order, we consider the complexity of the three checks in our implementation with regard to the number of clauses and the number of features within the clauses.

At first, the function checks whether there exist equivalent clauses in the respective set and removes all duplicates. Our implementation ensures that the literals within the clauses data structure are sorted. Thus, checking two clauses for equivalence has a linear complexity regarding the number of features in both clauses. Normally, comparing all clauses with each other requires quadratic time complexity with respect to the number of clauses. However, using data structures such as hash tables, equivalence for all clauses typically can be checked in linear time.

Afterwards, the function detects and removes all clauses that are subsumed by other clauses in the formula. Similar to the check for equivalence, the check whether a clause subsumes another clause has a linear time complexity with respect to the number of features in the clauses. Regarding the number of clauses, the check for subsumption has a quadratic complexity.

Finally, the function removes all clauses that can be derived from other clauses in the formula. This final check guarantees a formula that contains no redundancy among

all contained clauses. However, checking for derivation is again an NP-hard problem. This can be shown by reducing the satisfiability problem to the problem of derivation. If *false* can be derived from a set of clauses then the corresponding formula is unsatisfiable and satisfiable otherwise. Thus, the problem has an exponential complexity regarding the number of features in the formula and, since every clause must be tested, it has a linear complexity regarding the number clauses. Nevertheless, the benefit of removing as much clauses as possible outweighs the approach's large overhead when removing a high amount of features.

The consideration of different levels of redundancy aims to reduce the overhead of the derivation check by applying the checks for equivalence and subsumption first. The concept is that both previous checks are able to efficiently remove some obvious redundancies and consequently reduce the input size for the following derivation check.

4. RELATED WORK

Feature-model slicing and its application were originally introduced and discussed by Acher et al. [1, 2]. Their description of the algorithm uses existential quantification to remove variables in CNFs. As mentioned above, the tools FAMILIAR and FeatureIDE use implementations of existential quantification. In our work, we showed that logical resolution can also be used for feature-model slicing and is in fact a more direct way than existential quantification. Another approach, to remove features from a feature model is the usage of feature-model views [9, 13]. In contrast to feature-model slicing, feature-model views only hide information from certain users without deleting the feature and updating dependencies.

For our algorithm, we use multiple techniques to simplify a CNF and remove redundancies. However, there exist many other methods for CNF simplification, as this task is crucial for many CNF applications. For instance, tautology, subsumption, and blocked-clause elimination [7, 8]. Another useful technique is unit-clause propagation, which is used by the DPLL algorithm in modern satisfiability solvers [6]. In future work, we plan to exploit this mechanism to further improve the performance of our stated approach.

The concept of slicing was originally introduced for source code by Weiser, called program slicing, which removes unwanted source code fragments from a program [19]. Contrary to feature-model slicing, program slicing operates on the implementation level rather than on the abstract modeling level. Slicing is also implemented for other models such as UML model slicing [11]. In addition, Thüm et al. use slicing techniques to implement information hiding in source code specifications (i.e., method contracts) [18].

5. CONCLUSION AND FUTURE WORK

In this work, we proposed a new base algorithm for feature-model slicing to improve the algorithm's runtime based on logical resolution and CNF minimization. In detail, we propose a minimum-clauses heuristic to optimize the order in which we apply the removal of features and present an algorithm to simplify the CNF.

In future work, we plan to evaluate our algorithm through a comparison with the state-of-the-art feature-model slicing techniques. Furthermore, we plan to investigate further possibilities to optimize the algorithm. For instance, the usage of other CNF simplification methods could speed up the algorithm. If we are able to define a heuristic for removing enough redundancy in a CNF to avoid exponential clause growth, the algorithm would scale better for large feature models.

6. REFERENCES

[1] M. Acher, P. Collet, P. Lahire, and R. B. France. Decomposing Feature Models: Language, Environment, and Applications. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 600–603. IEEE, 2011.

[2] M. Acher, P. Collet, P. Lahire, and R. B. France. Slicing Feature Models. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 424–427. IEEE, 2011.

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

[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] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski. A Survey of Variability Modeling in Industrial Practice. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 7:1–7:8. ACM, 2013.

[6] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Commun. ACM*, 5(7):394–397, 1962.

[7] N. Eén and A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *Proc. Int'l Conf. Theory and Applications of Satisfiability Testing (SAT)*, pages 61–75. Springer, 2005.

[8] M. Heule, M. Järvisalo, and A. Biere. Clause elimination procedures for CNF formulas. In *Proc. Int'l Conf. Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 357–371. Springer, 2010.

[9] A. Hubaux, P. Heymans, P.-Y. Schobbens, D. Deridder, and E. K. Abbasi. Supporting Multiple Perspectives in Feature-Based Configuration. *Software and System Modeling*, 12(3):641–663, 2013.

[10] 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.

[11] K. Lano and S. Kolahdouz-Rahimi. Slicing of UML Models Using Model Transformations. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MODELS)*, pages 228–242. Springer, 2010.

[12] M. Mendonça, A. Wąsowski, and K. Czarnecki. SAT-Based Analysis of Feature Models is Easy. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 231–240. Software Engineering Institute, 2009.

[13] J. Schroeter, M. Lochau, and T. Winkelmann. Multi-Perspectives on Feature Models. In *Proc. Int'l*

Conf. Model Driven Engineering Languages and Systems (MODELS), pages 252–268. Springer, 2012.

[14] R. Schröter, S. Krieter, T. Thüm, F. Benduhn, and G. Saake. Feature-Model Interfaces for Compositional Analyses. Technical report, University of Magdeburg, 2015.

[15] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. Configuration Coverage in the Analysis of Large-Scale System Software. *ACM SIGOPS Operating Systems Review*, 45(3):10–14, 2012.

[16] 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, 2014.

[17] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund. Abstract Features in Feature Modeling. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 191–200. IEEE, 2011.

[18] T. Thüm, T. Winkelmann, R. Schröter, M. Hentschel, and S. Krüger. Variability Hiding in Contracts for Dependent Software Product Lines. In *Proc. Int'l Workshop Variability Modelling of Software-intensive Systems (VaMoS)*, pages 97–104. ACM, 2016.

[19] M. Weiser. Program Slicing. *IEEE Trans. Software Engineering (TSE)*, 10(4):352–357, 1984.