Comparing Algorithms for Efficient Feature-Model Slicing

Sebastian Krieter
University of Magdeburg
Magdeburg, Germany

Reimar Schröter
University of Magdeburg
Magdeburg, Germany

Thomas Thüm
TU Braunschweig
Brunswick, Germany

Wolfram Fenske
University of Magdeburg
Magdeburg, Germany

Gunter Saake
University of Magdeburg
Magdeburg, Germany

ABSTRACT

Feature models are a well-known concept to represent variability in software product lines by defining features and their dependencies. During feature-model evolution, for information hiding, and for feature-model analyses, it is often necessary to remove certain features from a model. As the crude deletion of features can have undesirable effects on their dependencies, dependency-preserving algorithms, known as feature-model slicing, have been proposed. However, current algorithms 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 the minimization of logical formulas. We empirically evaluate the scalability of our algorithm on a number of feature models and find that our algorithm generally outperforms existing algorithms.

Keywords

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

CCS Concepts

•**Software and its engineering** → *Software product lines; Abstraction, modeling and modularity;*

1. INTRODUCTION

Today, industrial software systems are often based on a high number of variable assets, called features. Feature models are commonly used to describe dependencies between these features [10]. As prominent examples such as the Linux kernel and other software product lines [5, 17] illustrate, feature models can become very large with more than 10,000 features. Numerous applications require the removal of one or more features from a feature model. For instance, when evolving a feature model, features can become obsolete and have to be removed or replaced by other features. Similarly, features are removed when eliminating abstract features [19], generating feature-model interfaces [16], or decomposing feature models [1]. All these applications require that existing feature dependencies are preserved. For instance, assume that feature B is removed from the feature model given by the propositional formula $(A \Rightarrow B) \wedge (B \Rightarrow (C \wedge D))$. A crude syntactical elimination of variable B would be incorrect, because the dependencies between the remaining variables, A, C, and D, would be violated. The intended result is $A \Rightarrow (C \wedge D)$ as it maintains these dependencies.

There are algorithms to remove a feature from a feature model, while keeping other dependencies intact. This technique is known as feature-model slicing [2]. However, when removing a large number of features, existing algorithms do not scale well. For instance, in our previous investigations of feature-model interfaces, in which we removed more than 1,000 features, FeatureIDE's algorithm took several hours to complete [16]. In this paper, we improve crucial aspects of existing algorithms in order to increase their performance.

Ultimately, feature-model slicing is NP-hard as the satisfiability problem can be reduced to the removal of all features. However, it has been shown that in the domain of feature models, satisfiability problems are often solvable in an adequate amount of time [14]. This motivated us to optimize feature-model slicing so that even the removal of thousands of features scales well. In detail, our contributions are:

- We propose an improved algorithm for feature-model slicing based on
  - the exchange of existential quantification for logical resolution, and
  - a new heuristic to optimize the order in which features are removed.
- We compare our approach to existing feature-model slicing strategies and discuss advantages and drawbacks.

2. PROBLEMS OF FEATURE REMOVAL

Feature diagrams are a common way to represent feature models [10]. However, to apply feature-model slicing, we represent feature models as propositional formulas. In this context, features are represented by logical variables and their dependencies are expressed using logical operators such as negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), and implication ($\Rightarrow$). Every feature diagram can be transformed into a propositional formula [4]. Using a *graph product line* as an example (cf. Figure 1), we illustrate its feature diagram and the respective propositional formula:

SPLC '16, September 16-23, 2016, Beijing, China

DOI: http://dx.doi.org/10.1145/2934466.2934477

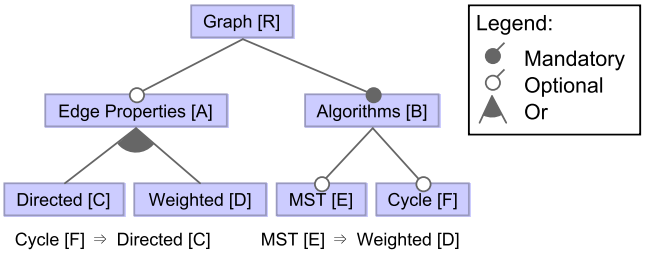

Figure 1: Feature model of a graph product line.

$$
\begin{array}{ll}
\textit{Root} & R \wedge \\
\textit{Child-Parent} & (A \Rightarrow R) \wedge (B \Rightarrow R) \wedge (C \Rightarrow A) \wedge \\
 & (D \Rightarrow A) \wedge (E \Rightarrow B) \wedge (F \Rightarrow B) \wedge \\
\textit{Mandatory} & (R \Rightarrow B) \wedge \\
\textit{Or group} & (A \Rightarrow (C \vee D)) \wedge \\
\textit{Constraints} & (F \Rightarrow C) \wedge (E \Rightarrow D)
\end{array}
$$

The sliced feature model should be easy to analyze and also allow for a reconstruction into a diagram representation. Therefore, we represent feature models in *conjunctive normal form (CNF)*. A propositional formula in CNF consists of a conjunction of clauses, which, in turn, consists of a disjunction of literals. A literal is a variable in positive or in negated form. In set notation, a CNF is 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 variables.

Based on the CNF representation, existential quantification of a variable is used as the state-of-the-art approach for feature-model slicing [2, 19]. The idea of existential quantification is to replace all occurrences of a variable in the formula with both possible assignments, *true (T)* and *false(F)*. In detail, all clauses in the formula that contain the variable are duplicated and combined with a logical *or*. The variable is replaced with *true* on the one side and with *false* on the other side. Afterwards, the formula is simplified and transformed back into CNF. To illustrate this approach, we use the relevant parts of the formula of the graph product line (i.e., $(D \Rightarrow A) \wedge (A \Rightarrow (C \vee D)) \wedge (E \Rightarrow D)$) and remove variable D:

$$
\begin{array}{ll}
\text{(1) CNF:} & (\neg D \vee A) \wedge (\neg A \vee C \vee D) \wedge (\neg E \vee D) \\
\text{(2) Replace:} & ((\neg \mathbf{F} \vee A) \wedge (\neg A \vee C \vee \mathbf{F}) \wedge (\neg E \vee \mathbf{F})) \vee \\
 & ((\neg \mathbf{T} \vee A) \wedge (\neg A \vee C \vee \mathbf{T}) \wedge (\neg E \vee \mathbf{T})) \\
\text{(3) Simplify:} & ((\neg A \vee C) \wedge \neg E) \vee A \\
\text{(4) CNF:} & \neg E \vee A
\end{array}
$$

Based on our experience with existing feature-model slicing algorithms, we identified two points of improvement. The first point concerns the use of existential quantification and the second point concerns the order in which the features are removed. In detail, the concept of existential quantification can be simplified as it contains unnecessary steps that degrade the slicing process' overall performance. Furthermore, currently used heuristics to determine the order of feature removal are based on the feature-diagrams' tree structure (e.g., *preorder* and *reverse level order*), but not on the structure of the logical formula that is modified. Moreover, the removal order is fixed for the duration of the slicing process. In the following section, we discuss how we improve these points and, thus, make feature-model slicing more efficient.

3. FEATURE-MODEL SLICING

In this section, we propose our new algorithm for feature-model slicing based on logical resolution. First, we provide an overview about the main idea of logical resolution and the effect on feature-model slicing. Second, we present the base algorithm in pseudo code and illustrate its behavior using the graph product line example. Finally, we describe crucial details of the algorithm such as feature-order determination and CNF simplification.

3.1 Logical Resolution

In our algorithm we apply logical resolution to remove variables from a CNF. 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., $c_{new} = (c_1 \cup c_2) \setminus \{l, \neg l\}$).

Again, we consider the removal of variable D from the formula of our graph product line to illustrate the mechanism of logical resolution. If we apply resolution with respect to D to the relevant clauses of the formula (i.e., $(\neg D \vee A) \wedge (\neg A \vee C \vee D) \wedge (\neg E \vee D)$), we get the resolvents $(\neg A \vee A \vee C)$ and $(\neg E \vee A)$. The first resolvent $(\neg A \vee A \vee C)$ is a tautology (i.e., always *true*) and, thus, can be neglected in the following step. By adding the remaining resolvent $(\neg E \vee A)$ to the formula of the graph product line and removing the original clauses that contained D, we get the desired result.

The application of logical resolution is a direct consequence from existential quantification and the subsequent transformation into CNF. During a variable's removal via 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 in a pairwise manner. Thus, logical resolution yields the same result as existential quantification, while additionally keeping the formula in CNF.

3.2 Slicing Algorithm Overview

Our algorithm takes a CNF formula and a set of variables as input and returns another formula in CNF representing the sliced feature model, which contains no variable from the given set. We use an iterative process that removes one variable at a time in two distinct phases. First, redundant clauses are removed from the formula. Second, resolution is performed with respect to the currently processed variable.

In Algorithm 1, we present our algorithm in pseudo code. We use the following variable notation:

$$
\begin{array}{llll}
v & \rightarrow \text{ variable name} & \mathcal{V} & \rightarrow \text{ set of variable names} \\
l & \rightarrow \text{ literal} & c & \rightarrow \text{ clause (set of literals)} \\
\mathcal{C} & \rightarrow \text{ set of clauses} & &
\end{array}
$$

Our algorithm has two input parameters, a feature model's CNF representation $\mathcal{C}_{cnf}$ and a set of variables that should be removed $\mathcal{V}_{remove}$. First, all clauses of the given formula are divided into one of two sets, $\mathcal{C}_{dirty}$ or $\mathcal{C}_{clean}$ (cf. Line 2–5). The dirty set contains all clauses that contain at least one variable from $\mathcal{V}_{remove}$. Consequently, the clean set contains

Algorithm 1 Slicing algorithm – Iteratively removes all variables in $\mathcal{V}_{remove}$ from $\mathcal{C}_{cnf}$

```
1: function removeVariables(C_cnf, V_remove)
2:     C_dirty ← ∅, C_clean ← ∅
3:     for all c_1 ∈ C_cnf do
4:         classify(c_1, V_remove, C_dirty, C_clean)
5:     end for
6:     while C_dirty ≠ ∅ do
7:         removeRedundant(C_clean, C_dirty)
8:         v_1 ← next(V_remove)
9:         V_remove ← V_remove \ {v_1}
10:        resolution(v_1, V_remove, C_dirty, C_clean)
11:    end while
12:    return C_clean
13: end function
```

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 6–11). When the algorithm is finished, the clean set contains all remaining clauses, whereas the dirty set is empty. Finally, the resulting formula is constructed as a conjunction of all clauses in the clean set.

Considering the details in each iteration of the main procedure (cf. Line 6–11), the algorithm simplifies the current CNF in $\mathcal{C}_{dirty}$ and $\mathcal{C}_{clean}$ by removing redundant clauses. After the CNF simplification, the method `next` (see Section 3.3) returns a variable from $\mathcal{V}_{remove}$ (cf. Line 8) that is removed in this iteration (cf. Line 9). For each variable, resolution is performed with respect to the clauses in set $\mathcal{C}_{dirty}$ (cf. Line 10).

The Graph Product Line Example. We now provide a small example of how our algorithm proceeds with an input formula and a set of features that we want to remove. Using the graph product line (cf. Figure 1), we visualize the functionality of our algorithm. We execute our algorithm with the formula given in Section 2 in CNF and $\mathcal{V}_{remove} = \{B, D\}$. The algorithm starts by classifying each clause as clean or dirty. Resulting in: $\mathcal{C}_{dirty} = \{\{\neg\mathbf{B}, R\}, \{\neg\mathbf{D}, A\}, \{\neg E, \mathbf{B}\}, \{\neg F, \mathbf{B}\}, \{\neg E, \mathbf{D}\}, \{\neg R, \mathbf{B}\}, \{\neg A, C, \mathbf{D}\}\}$ and $\mathcal{C}_{clean} = \{\{R\}, \{\neg A, R\}, \{\neg C, A\}, \{\neg F, C\}\}$.

For this example, we assume that variable D will be removed in the first iteration. First, the algorithm removes the redundant clauses $\{\neg B, R\}$ from the dirty set and $\{\neg A, R\}$ from the clean set since they are subsumed by the clause $\{R\}$. Now, the algorithm applies resolution to all clauses in the dirty set that contain D (i.e., $\{\neg D, A\}$, $\{\neg E, D\}$, and $\{\neg A, C, D\}$). Next, the original clauses are removed from the dirty set and the resulting resolvents $\{\neg E, A\}$ and $\{\neg A, A, C\}$ are classified as clean as they contain no variable from $\mathcal{V}_{remove}$.

In the second iteration, the algorithm removes variable B. Again the algorithm starts with removing redundant clauses (i.e., $\{A, \neg A, C\}$, which is a tautology). Afterwards, the algorithm applies resolution with respect to B, but finds no resolvent. Thus, the algorithm only removes the clauses $\{\neg E, B\}$, $\{\neg F, B\}$, and $\{\neg R, B\}$ from the dirty set. Since the dirty set is now empty, the algorithm is finished and the remaining clauses in the clean set form the resulting CNF (i.e., $R \wedge (\neg C \vee A) \wedge (\neg F \vee C) \wedge (\neg E \vee A)$).

3.3 Feature Order

In Section 3.2, we stated that method `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 features from a feature model, we aim to use a heuristic rather than to find the optimal order.

We propose the heuristic *min-clause* that considers the number of new clauses that are generated in each resolution phase. Thus, the min-clause heuristic directly aims to reduce the number of generated clauses during resolution. It is a greedy strategy that selects the best variable in each iteration. In this case, the best variable is the one 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 redundancies), 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 maximum number of clauses that could be generated by resolution. This estimated number is used as a sorting criterion, whereupon 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, the feature order is dynamically adapted in each iteration.

3.4 CNF Simplification

The main issue when using existential quantification or logical resolution for feature-model slicing is the possibly exponential increase of the number of clauses in the formula. A high amount of clauses results in a slow overall performance when removing a large number of features. The method `removeRedundant` addresses this problem by removing newly generated clauses that are redundant. Clauses can be redundant either because they can be derived from other clauses in the formula or because they are tautologies.

In order to efficiently detect redundant clauses, we apply two kinds of redundancy tests. First, we use tests that detect only a subset of all redundant clauses, such as the test for tautology and subsumption. Second, we test whether a clause is derived from other clauses in the formula to find all remaining redundant clauses. The idea is that the first kind of tests are able to efficiently remove some obvious redundancies, since they run in polynomial time regarding the number of clauses. Consequently, the input size for the following derivation test, which has an exponential time complexity in the number of features, is reduced.

First, we remove all clauses that are tautologies because they contain a literal and its complement (i.e., $\{c \in \mathcal{C} \mid \exists l \in \mathcal{L} : \{l, \neg l\} \subseteq c\}$). It is possible to test this property for all clauses in linear time with respect to the number of clauses.

Second, we remove all clauses that are subsumed by another clause (i.e., $\{c \in \mathcal{C} \mid \exists c' \in \mathcal{C} : c' \neq c,\ c' \subseteq c\}$). By using a sorted list to store the literals of a clause, the test whether two clauses are equivalent has a linear complexity regarding the number of features in both clauses. Regarding the number of clauses, the complete subsumption test for all clauses has a quadratic complexity.

Finally, we remove all clauses that can be derived from other clauses in the formula (i.e., $\{c \in \mathcal{C} \mid \mathcal{C} \setminus \{c\} \vDash c\}$). This

final test guarantees a formula that contains no redundancy among all contained clauses. Since every clause must be tested, this test has a linear complexity regarding the number of clauses. However, it has an exponential complexity in the number of features in the formula. Nevertheless, the benefit of removing many clauses outweighs the approach's large overhead when slicing a high amount of features.

4. EVALUATION

In our evaluation, we compare different variants of our feature-model slicing algorithm. For this, we mainly focus on different feature-order heuristics. In addition to our proposed *min-clause* heuristic, we consider the three tree-traversal heuristics, *preorder* (used by FeatureIDE [12, 18]), *reverse level order* (used by FAMILIAR [3]), and *postorder*.

In detail, we measure and compare the time that each algorithm variant takes to remove a certain amount of features from a feature model. In order to do a fair comparison, we do not use any original tools, but use the same base implementation of our resolution algorithm. This ensures that there is no bias introduced by different data structures.

4.1 Evaluation Setup

In the following, we present our *feature models* and the *set of features* we are removing from the models. Our evaluation system runs on Linux and uses an Intel i5 processor with 4 cores and 8 GB main memory.

Used Feature Models. In total, we use 32 different feature models for our evaluation. Two of them are real-world feature models from our industrial partners from the automotive domain with 2,513 and 17,365 features. The other 30 feature models were generated by S.P.L.O.T. and were already used in other evaluations [13]. We use feature models that consist of 1,000, 2,000, and 5,000 features respectively.

Feature Removal. Since we do not know a user's intention to remove features, we cannot determine which features should be removed. Thus, we use a random set of features to remove from each model. In detail, we create a shuffled list of all features from a feature model. Then, we create 19 sublists that contain the first 5%, 10%, ..., 95% features of the shuffled list. This results in 19 measurements for each algorithm variant and feature model. For reliable results, we perform multiple time measurements with, in total, 25 different random seeds.

4.2 Evaluation Results

We depict a comparison for all evaluated feature models in Figure 2. In the diagram, we show the results of all measurements for the three tree-traversal heuristics relative to the corresponding value measured for our min-clause heuristic. Each box plot consists of 800 measured values as it contains 25 values for every one of the 32 feature models. The diagram uses a logarithmic scale. Overall, the results show that the tree-traversal heuristics tend to be faster than the min-clause heuristic when removing 20% or less features from a feature model. However, the absolute measured values of all heuristics are still comparably low. The median of all measured values for removing 10% of the features with the min-clause heuristic is 50 ms. Furthermore, when removing 30% and more features, the tree-traversal heuristics tend to be slower. For a removal of 80% with the min-clause heuristic, the median of the absolute values reaches its maximum with 438 ms.

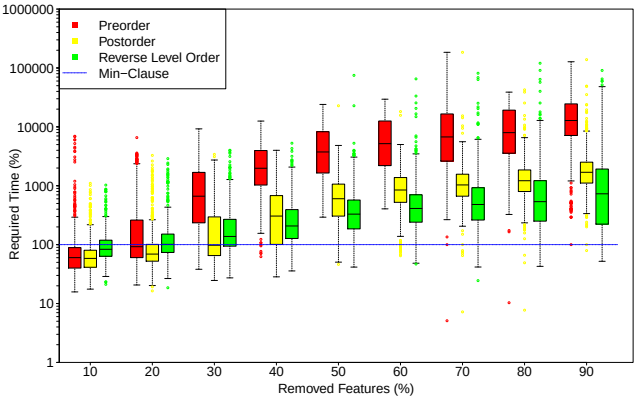

Figure 2: Relative comparison of different feature-order heuristics with all evaluated feature models (logarithmic scale).

We can see from our results that when removing 30% or more features the min-clause heuristic outperforms all other heuristics. Considering the relatively low absolute numbers for the removal of 20% or less features, one explanation for the evaluation result is the overhead introduced by the min-clause heuristic, since it has to estimate the number of newly generated clauses for each removed variable. In addition, preorder has the worst performance of all compared tree-traversal heuristics. This is most likely due to the parent-child dependencies in the feature-tree structure, as features in higher levels of a feature tree tend to have more relations than features in lower levels.

4.3 Threats to Validity

In Section 4.1 we specified that we used 25 random seeds to determine the feature sets removed from each feature model. This sampling size is insignificant compared to the number of possible feature sets (i.e., 2^n). Thus, it might be possible that our results are biased due to bad sampling. We tried to avoid this situation by choosing random feature sets of different sizes.

Since we performed the evaluation ourselves, we cannot exclude bugs in our evaluation tool, which might affect the final results. However, we extensively tested our tool with smaller feature models to avoid such kinds of errors.

Unfortunately, we were not able to get more large-scale, real-world feature models that are organized in a feature tree. Due to this lack of real-world feature models, our results may not be generalizable. Therefore, we used a high amount of artificial feature models that were used in other evaluations before.

5. 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 use of feature-model views [9, 15]. 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 (e.g., 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 algorithm.

6. CONCLUSION AND FUTURE WORK

In our work, we proposed a new algorithm for feature-model slicing, which aims to run more efficiently than existing algorithms. By removing different amounts of features from several feature models, we compared four variants of our algorithm, which differ in the feature-order heuristic they employ.

As our evaluation indicates, a traversal-order heuristic like postorder or reverse level order can be helpful when removing a low number of features from a feature model (e.g., less than 30%). Thus, these heuristics are most suited for applications such as removing single features or small subtrees. In contrast, our proposed min-clause heuristic performs best when removing a high amount of features. All applications that require the removal of more than 30% of features can benefit from our approach. One possible application is the removal of abstract features. However, the applications that benefit most are the generation of feature-model interfaces and the decomposition of a large feature models into smaller sub models, as they often require to remove a large number of features.

Concerning future work, we propose multiple possibilities for further optimizations. First, the usage of other CNF simplification methods could speed up the algorithm. If we are able to define an efficient heuristic for removing enough redundancy in a CNF to avoid an exponential increase of the number of clauses, the algorithm might scale better for larger feature models. Second, the current implementation relies solely on a SAT-solver implementation. Other methods can be used to avoid using the SAT-solver, which currently is the most expensive task in the implementation.

Acknowledgments. This work was partially supported by BMBF grant (01IS14017B). A preliminary version of this paper is also available as technical report [11].

7. REFERENCES

[1] M. Acher, P. Collet, P. Lahire, and R. B. France. Decomposing Feature Models: Language, Environment, and Applications. In *ASE*, pages 600–603. IEEE, 2011.

[2] M. Acher, P. Collet, P. Lahire, and R. B. France. Slicing Feature Models. In *ASE*, pages 424–427. IEEE, 2011.

[3] M. Acher, P. Collet, P. Lahire, and R. B. France. FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models. *SCP*, 78(6):657–681, 2013.

[4] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC*, pages 7–20. Springer, 2005.

[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 *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 *SAT*, pages 61–75. Springer, 2005.

[8] M. Heule, M. Järvisalo, and A. Biere. Clause Elimination Procedures for CNF Formulas. In *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. *SoSyM*, 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] S. Krieter, R. Schröter, , T. Thüm, and G. Saake. An Efficient Algorithm for Feature-Model Slicing. Technical report, University of Magdeburg, 2015.

[12] J. Meinicke, T. Thüm, R. Schröter, S. Krieter, F. Benduhn, G. Saake, and T. Leich. FeatureIDE: Taming the Preprocessor Wilderness. In *ICSE*, pages 629–632. ACM, 2016.

[13] M. Mendonça, M. Branco, and D. Cowan. S.P.L.O.T.: Software Product Lines Online Tools. In *OOPSLA*, pages 761–762. ACM, 2009.

[14] M. Mendonça, A. Wąsowski, and K. Czarnecki. SAT-Based Analysis of Feature Models is Easy. In *SPLC*, pages 231–240. Software Engineering Institute, 2009.

[15] J. Schroeter, M. Lochau, and T. Winkelmann. Multi-Perspectives on Feature Models. In *MODELS*, pages 252–268. Springer, 2012.

[16] R. Schröter, S. Krieter, T. Tüm, F. Benduhn, and G. Saake. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *ICSE*, pages 667–678. ACM, 2016.

[17] 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.

[18] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *SCP*, 79(0):70–85, 2014.

[19] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund. Abstract Features in Feature Modeling. In *SPLC*, pages 191–200. IEEE, 2011.