SMT-Based Variability Analyses in FeatureIDE

Joshua Sprey, Chico Sundermann
TU Braunschweig
Brunswick, Germany

Sebastian Krieter
Otto-von-Guericke-University
Magdeburg, Germany
Harz University of Applied Sciences
Wernigerode, Germany

Michael Nieke
TU Braunschweig
Brunswick, Germany

Jacopo Mauro
University of Southern Denmark
Odense, Denmark

Thomas Thüm
University of Ulm
Germany

Ina Schaefer
TU Braunschweig
Brunswick, Germany

ABSTRACT

Handling configurable systems with thousands of configuration options is a challenging problem in research and industry. One of the most common approaches to manage the configuration options of large systems is variability modelling. The verification and configuration process of large variability models is manually infeasible. Hence, they are usually assisted by automated analyses based on solving satisfiability problems (SAT). Recent advances in satisfiability modulo theories (SMT) could prove SMT solvers as a viable alternative to SAT solvers. However, SMT solvers are typically not utilized for variability analyses. A comparison for SAT and SMT could help to estimate SMT solvers potential for the automated analysis. We integrated two SMT solvers into FeatureIDE and compared them against a SAT solver on analyses for feature models, configurations, and realization artifacts. We give an overview of all variability analyses in FeatureIDE and present the results of our empirical evaluation for over 122 systems. We observed that SMT solvers are generally faster in generating explanations of unsatisfiable requests. However, the evaluated SAT solver outperformed SMT solvers for other analyses.

KEYWORDS

variability analysis, feature models, smt, smt analysis, sat, sat analysis, sat vs smt, feature model analysis, configuration analysis, preprocessor analysis, attribute optimization, feature attributes

1 INTRODUCTION

Variability management is an approach to handle the complexity of configurable systems without having a large negative impact on economical factors for a company [1, 36]. Besides the ability to manage the variability of products, industry values benefits such as product configuration, requirement specification and product derivation [7]. The configuration of product lines (i.e., a family of products that shares a base of reusable artifacts, also called features [24]) is complex and still an active challenge in research and industry [10, 38]. Users can define a set of selected and unselected features, also called configuration, that can be used to compose a product from the product line using the base of reusable artifacts.

Feature modelling is one of the most common notations for the variability management of product lines [7, 12, 22]. The feature model describes all features of a product line for a given domain and their relationships in a tree-like structure [1].

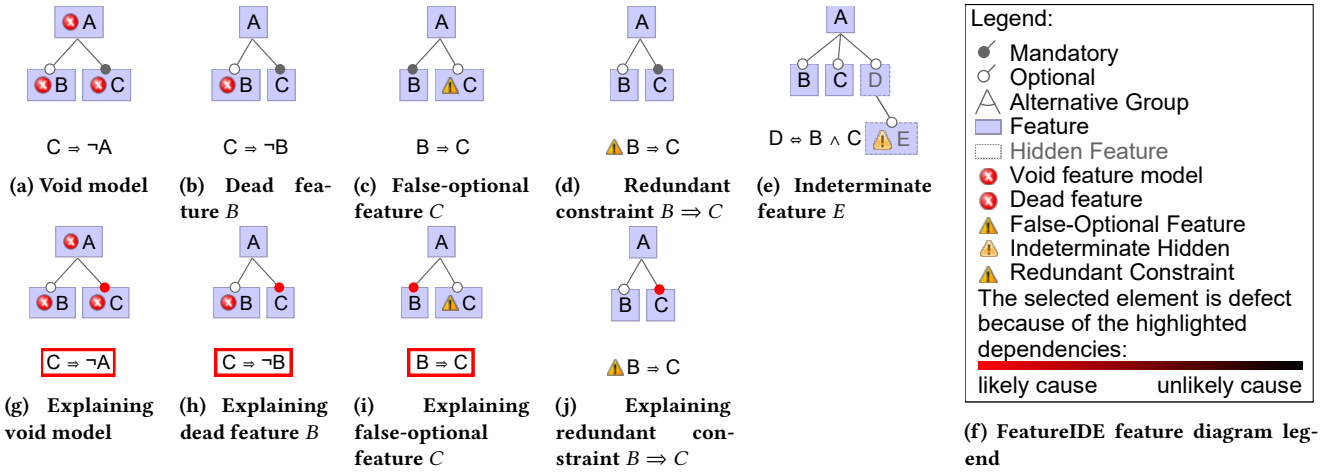
However, feature models often contain defects that negatively impact the benefits of feature models. The manual detection of defects is hard, time-consuming, and for large feature models infeasible, so automated analyses are needed [1, 6]. Furthermore, proposed variability analyses in the literature cover more than just detecting feature model defects, such as supporting the configuration process [6, 19] and the verification of realization artifacts [44, 45]. One of the common approaches for feature models is to translate the feature model into a propositional formula which is then analysed by off-the-shelf SAT solvers [3, 6, 30].

SAT solvers are traditionally used for variability analyses of feature models. However, SMT solvers received increasing attention and tremendous advances in recent decades, motivating their use for variability analyses [14]. In addition, problems that are hard to encode into SAT, such as the encoding of at-most-k constraints [9], might be easier to express with SMT. Furthermore, SMT is more expressive and can be used to solve more complex problems for feature models such as the computation of attribute ranges, and the anomaly analysis for evolving [34] or context-aware [31] feature models. As SMT is a generalization of SAT, we are interested whether SMT solvers can perform SAT-based analyses with the same efficiency as traditional SAT solvers. We want to to empirically assess the potential of SMT solvers for variability analyses.

We integrated JavaSMT [25], a unified interface for SMT solvers, into the feature modelling tool FeatureIDE [47] because JavaSMT provides easy access to multiple SMT solvers and is also written in Java, just like FeatureIDE [47]. Besides, FeatureIDE already provides many analyses for feature models, configurations, and realization artifacts. We compared the runtime of FeatureIDE's SAT solver Sat4J [8] against the two SMT solvers SMTInterpol [11] and Z3 [13] provided by JavaSMT for a subset of all analyses. We used 117 real-world feature models for the empirical evaluation of feature model and configuration analyses. The empirical evaluation for the realization artifacts is based on two Antenna [20]

VaMoS '20, February 5–7, 2020, Magdeburg, Germany

ACM ISBN 978-1-4503-7501-6/20/02…$15.00
https://doi.org/10.1145/3377024.3377036



(a) Void model **(b) Dead feature B** **(c) False-optional feature C** **(d) Redundant constraint $B \Rightarrow C$** **(e) Indeterminate feature E**

(g) Explaining void model **(h) Explaining dead feature B** **(i) Explaining false-optional feature C** **(j) Explaining redundant constraint $B \Rightarrow C$**

(f) FeatureIDE feature diagram legend

Figure 1: Examples for all kinds of feature model defects and their explanations.

preprocessor examples available in FeatureIDE. Furthermore, we implemented an SMT-based analysis to compute attribute ranges for partial configurations with the aim to assist users in interactive configuration processes. We evaluated the runtime of the analysis for three real-world models.

We summarize our contribution as follows:

- We give an overview of the different variability analyses which are currently supported by FeatureIDE.
- We provide tool support for the comparison of SAT and SMT solvers in FeatureIDE.
- We performed an empirical evaluation to estimate the usefulness of SMT solvers for variability analyses.

We categorize the variability analyses supported by FeatureIDE into analyses that depend on a feature model (cf. Section 2), on a feature model and a configuration (cf. Section 3), on a feature model and realization artifacts (cf. Section 4), and on an extended feature model and a partial configuration (cf. Section 5).

2 ANALYSES OF FEATURE MODELS

The analyses described in this section rely only on the feature model as input. They cover detecting and explaining defects, identifying feature model edits, computing information on products, and identifying atomic sets.

Defects can occur when developing feature models with cross-tree constraints. Von der Maßen et al. classified them into redundancies, anomalies, and inconsistencies [49]. Most serious are inconsistencies which occur when conflicting information is modelled, resulting in no products for the product line. Moderate are anomalies that indicate an inadvertent loss of product space. The least severe are redundancies, which are information that has already been modelled differently. They can be removed to simplify the maintenance of the feature model [49].

The detection of defects can help users identifying a problem in their feature model. However, understanding the reasons for defects and fixing them is hard. Assisting the user in finding the actuator for an actual defect requires the generation of explanations [4, 6, 27]. An explanation contains information about the type of defect they explain and a minimal set of features and constraints that causes the defect. Both the detection and explanation of defects are important operations for the feature model error analysis [6].

Void Analysis detects whether a feature model is *void*, meaning it represent no products [6]. Void feature models are the results of inconsistencies in the model and their detection is regarded as critical in feature modelling tools [6, 33]. A non-void feature model is a prerequisite to do other analyses, so it is always performed first. An example for a void feature model is shown in Figure 1a. It can be explained textually: *The feature model is void because C is a mandatory child of A (i.e., $A \Rightarrow C$) and $C \Rightarrow \neg A$ is a constraint* or visually by highlighting all relationships that cause the void feature model. For instance, Figure 1g shows the visual explanation of our void feature model.

Core & Dead Feature Analysis detects all core and dead features for a given feature model. A *core feature* is a feature that is part of every product of the feature model. In contrast, a *dead feature* is a feature that is not part of any product [1, 6, 27]. Dead features are severe anomalies as they can be never selected, and, hence, the effort to realize dead features is useless. Core features on the other hand are no defects and have a rather positive effect on feature modelling. They indicate which features should be focused when starting to implement the product line [6]. The analyses for core and dead features are similar which enables us to compute both sets of features in one analysis [27]. An example for a dead feature is shown in Figure 1b. The concrete feature B is dead because C is a mandatory child of the root and excludes feature B. The according visual explanation is shown in Figure 1h.

False-Optional Feature Analysis detects all false-optional features for a given feature model. A *false-optional feature* is modelled as optional but has a mandatory relationship to its parent due to cross-tree constraints [34]. This anomaly type is not severe but prevents the selection of feature combinations which are modelled. Figure 1c shows an example for a false-optional feature. The feature C is false-optional because it is included by B which is a mandatory child of the root. Figure 1i shows the according visual explanation.

Indeterminate Feature Analysis detects all indeterminate features for a given feature model. Sometimes it is possible to hide configuration options in the configuration process [3, 16, 53]. We adapted this concept to feature models to hide features in the configuration process. We name such features *hidden features*. They are useful in particular for features which are relevant for the product-line implementation (e.g., derivative modules [29]) but not for the customers configuration process. As a hidden feature is not manually configurable, its selection must be determined by the structure and constraints of visible features in the feature model. However, if a hidden feature's selection is *not* determined, we call it *indeterminate*. We regard indeterminate features as weak anomalies as they indicate wrong usage of hidden features or constraints. Figure 1e shows an example for an indeterminate feature. The example contains two hidden features D, E. The selection of D is determined by the constraint $D \Leftrightarrow B \wedge C$. However, E is optional and not contained in any constraint, and, thus, is indeterminate.

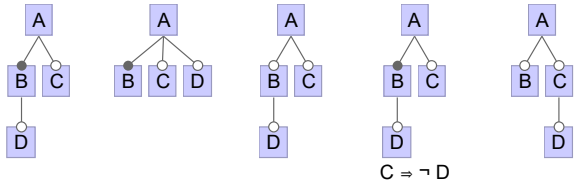
Redundant Constraint Analysis detects all constraints whose semantic information is already modelled by the feature model structure or other constraints. Such constraints are called redundant constraints [6, 49]. Redundant constraints have no influence on the available products of the product line, so they are not severe. They can be introduced for better readability and understanding, but at the cost of higher maintainability for feature models. Figure 1d shows an example for a redundant constraint. The constraint $B \Rightarrow C$ is redundant because C is a mandatory child of the root feature A. Figure 1j shows the according visual explanation.

Tautological Constraint Analysis detects all redundant constraints that are always satisfied. Such redundant constraints are called *tautologies* [27]. An example for a tautology is $B \vee \neg B$. Tautological constraints are superfluous and should always be removed.

Feature Model Edits describe the relationship between a feature model before and after an evolution. Thüm et al. [46] classified feature model edits into *refactoring*, *generalization*, *specialization*, and *arbitrary edit*. We consider the feature model in Figure 2a as input model for the following examples. A *refactoring* does not have an impact of the set of possible products. Figure 2b shows a refactoring of the input model. A *generalization* introduces new products without removing any existing ones. As an example, Figure 2c shows a generalization of the input model. A *specialization* removes existing products without introducing new ones. For instance, Figure 2d shows a specialization of the input model. An *arbitrary edit* removes existing products and introduces new ones. Arbitrary edits hinder the understandability of feature model evolution and should be replaced with step-by-step generalization and specialization [46]. Figure 2e shows an arbitrary edit of the input model.

Number of Configurations computes the number of valid configurations represented by the feature model. This number can be used to measure the complexity and variability of a product line [6].

Number of Product Variants computes the number of product variants which are represented by the feature model. Product variants are different from valid configuration as they ignore features that have no implementation [48]. The computation for both the number of valid configurations and program variants is hard for regular SAT solver as every assignment needs to be considered.



(a) Original **(b) Refactoring** **(c) Generalization** **(d) Specialization** **(e) Arbitrary Edit**

Figure 2: Examples for all kinds of feature model edits, see Figure 1 for a legend.

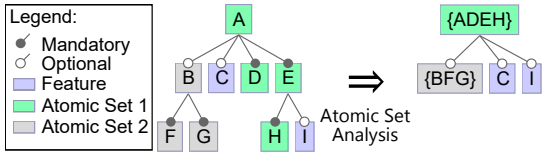


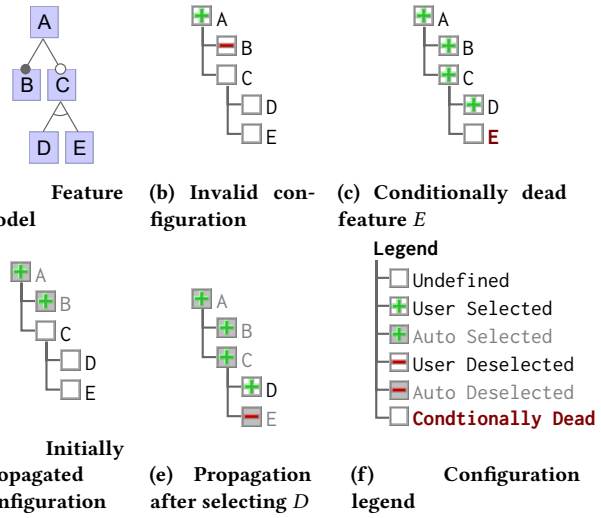
Figure 3: Example for a feature model before (left) and after (right) the atomic set reduction.

Atomic Set Analysis computes all subsets of features whose features are all included or all excluded in every product of a product line. These subsets are called *atomic sets* [15, 39]. Atomic sets can be used to reduce the size of a product line prior to its analysis by replacing each feature of an atomic set by one feature that represents the entire atomic set [40, 52]. Segura et al. [40] showed that atomic sets can improve the efficiency of the automated analysis of feature models. However, they only compute atomic sets that contain mandatory features and their parents. In contrast, computing all atomic sets for a feature model is more expensive and reduces the advantages of atomic sets for the automated analysis. For instance, Figure 3 shows the states of a feature model before and after the atomic set analysis.

FeatureIDE provides an editor that interactively detects void models [27], core and dead features [27], false-optional features [27, 34], redundant constraints [27, 49], and tautologies [27]. Additionally, textual and visual explanations can be generated on-the-fly [17, 27]. Furthermore, FeatureIDE supports the hidden features and their indeterminate analysis that we introduced above. FeatureIDE also automatically identifies feature model edits [46]. In addition, statistical information about a feature model can be computed on-demand such as all valid configurations [48], all program variants [48], and atomic sets [15, 39]. We refer readers interested in the different analyses to the works mentioned above.

3 ANALYSES OF CONFIGURATIONS

The configuration process is a difficult challenge as each choice can affect the remaining configuration options available to users. Especially for larger models, it is infeasible to manually consider all constraints and structural information of the feature model when selecting or deselecting a feature. Therefore, automated analyses are required to support the interactive configuration process. Analyses



(a) Feature model **(b) Invalid configuration** **(c) Conditionally dead feature *E***

(d) Initially propagated configuration **(e) Propagation after selecting *D*** **(f) Configuration legend**

Figure 4: Examples for configuration analyses. All examples are use the feature model shown in (a).

described in this section rely on the feature model and a configuration as input. They cover the validation of configurations, automatic error resolution, decision propagation, and their explanations.

Validity Analysis computes whether a given configuration is *valid*, meaning it can be used to compose a product of the product line [6]. Figure 4b shows an invalid partial configuration because feature *B* is manually deselected but has a mandatory relationship to the root feature *A*. It is the most important analysis for users as an invalid configuration indicates a dead end in the configuration process.

Conditionally Core & Dead Feature Analysis computes all core and dead features while considering all selections for a given configuration. A feature that is core/dead under certain conditions (e.g., for a given partial configuration) is called *conditionally core/dead* [6]. Figure 4c shows an example for such a conditionally dead feature. The feature *E* is conditionally dead under the condition that feature *D* is selected because of the alternative relationship between them.

Automatic Decision Propagation Analysis assists users after selecting or deselecting a feature by automatically determining the selection of dependent features. Figure 4d shows the automatic propagation after a new configuration is opened for the first time. The features *A* and *B* are automatically selected as they appear in every product. Furthermore, Figure 4e shows the same configuration after feature *D* is selected. The feature *C* must be selected as well because of the parent-children relationship with feature *D*. In addition, feature *E* must be automatically deselected as it is an alternative sibling of feature *D*. However, it is infeasible to manually keep track of all feature dependencies. In order to assist users after selecting a feature, it is necessary to compute all features that cannot be selected after a change (i.e., are conditionally dead) and features that must be selected after a change (i.e., are conditionally core). To do so, we assume that all selected/deselected features from our partial configuration are `true`/`false`. Performing a conditionally core and dead feature analysis results in all dead features which never appear in any product that can be configured with the user's current partial configuration. Therefore, we can propagate all these features as automatically deselected. In contrast, the conditional core features can be propagated as automatically selected [28]. While automatically selected features prevent the introduction of invalid configurations, they also restrict the freedom of users. Guenther [17] introduced the SAT-based generation of explanations for automatically selected features. These explanations are required to assist users in understanding the reasons for the automatic selection of a feature.

Error Resolution Analysis assists users by resolving conflicts for invalid configurations. Changing a feature model can lead to conflicts in existing configurations. Thus, it is necessary to review those configurations and correct them if necessary. This process can be very tedious especially if many configuration options are available which users need to review. To assist users it is possible to generate an explanation for a given invalid configuration. The resulting explanation contains all contradicting selections that make the configuration invalid. These can be automatically reset to revert an invalid configuration back to a valid one [28].

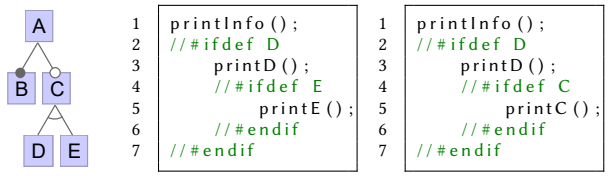
FeatureIDE provides a configurator that interactively validates configurations [6] and performs automatic decision propagation. Furthermore, users are prevented from changing the selection state of an automatically propagated feature to prevent the introduction of an invalid configuration. However, FeatureIDE generates explanations for automatically propagated features on-the-fly to assist users in understanding their automated selection [17]. In addition, FeatureIDE provides automatic error resolution that we introduced above on-demand when the configurations of users are invalid.

4 ANALYSES OF PREPROCESSOR DIRECTIVES

With the help of realizations artifacts and a valid configuration of a product line, it is possible to automatically compose a program variant. Preprocessor-based product lines combine the artifacts for all features into a single code base. Annotations are used to control the variability of the implementation. Each annotation has a presence condition that needs to be fulfilled for the annotation's implementation to be included in the product. Figure 5b and Figure 5c show two code examples for the implementation of the feature model shown in Figure 5a. The examples use the syntax of the Java preprocessor *Antenna* [37]. However, the presence conditions in the realization artifacts are not always satisfiable with variability modelled in the feature model. A consistency check can be performed to avoid deviations between artifacts and feature model.

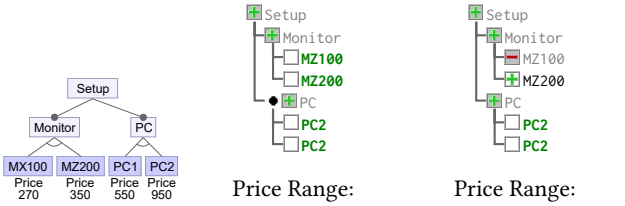
In the following, we describe the SAT-based analyses to detect unsatisfiable and unnecessary presence conditions. Each analysis needs a feature model, a preprocessor annotation, and the annotation's context as input.

Dead Code Block Analysis checks the presence condition of a given annotation for satisfiability in accordance with the variability modelled in the feature model. Each code block whose presence condition is not satisfiable is called *dead* [41]. Hence, the resulting implementation is not part of any product. A dead code block can



(a) Feature model **(b) Dead code block** **(c) Superfluous annotation**

Figure 5: Examples of a dead code block (b) and a superfluous annotation (c) for the feature model shown in (a).



(a) Extended FM **(b) Initial configuration with price range** **(c) Price range changes interactively after $MZ200$ was selected**

Figure 6: Example for an extended feature model and the calculation of attribute ranges during the interactive configuration process.

indicate a wrong presence condition or they can be removed for the sake of long-term maintenance costs and the readability of the source code [44]. For instance, Figure 5b shows an example for a dead code block. The code block at lines 4-6 is dead because it requires the selection of feature E but is nested inside a block from it's alternative sibling feature D. Guenther showed that dead code blocks can be effectively explained which helps the user in resolving them [17].

Superfluous Annotation Analysis checks whether the presence condition of a given annotation is always satisfied in accordance with the variability modelled in the feature model. Such an annotation is called *superfluous annotation* [44]. Their corresponding code blocks are included in each product making their presence condition dispensable. Superfluous annotations can indicate a wrong presence condition or the annotation can be removed for the sake of long-term maintenance costs and the readability of the source code [44]. Figure 5c shows an example for a superfluous annotations at line 4. The annotation is superfluous as the presence condition requires feature C. However, feature C is always selected as the parent block requires feature D which includes C because of their parent-child relationship. Guenther showed the efficient generation of explanations for superfluous annotation [17].

FeatureIDE supports the detection of dead code blocks [41], superfluous annotations [44], and provides the automatic generation of explanations for both of them [17]. We refer readers interested in the realization of different analyses to the works mentioned above.

5 ATTRIBUTE RANGE COMPUTATION FOR PARTIAL CONFIGURATIONS

In the previous sections, we elaborated about SAT-based variability analyses. However, some problems cannot be easily translated to SAT. A more expressive solver can be used to simplify the translation. In this section, we introduce an SMT-based analysis for feature attributes. We begin with a short introduction to extended feature models and attributes before presenting the SMT-based analysis.

Feature attributes [6] are used to extend the variability of the features and capture non-functional properties. For instance, features can be attributed with a price. When creating configurations, the price can be used to decide between alternative features [23]. Each attribute consists of at least a name, a domain, and a value. The value of a feature attribute can be set globally in the feature model or set during the configuration process. Feature models that contain features with attributes are called extended feature models [6]. Figure 6a shows an extended feature model of a simplified product line for a computer setup. In the configuration process, the user can choose between two monitors and two PCs. Each monitor and PC feature is globally attributed with a particular price. Further assistance in the configuration process is possible by performing optimization analyses.

An optimization analysis determines the best solution for a given optimization problem [5, 6]. The analysis receives an extended feature model and an objective function as input. The extended feature model provides attributes for the calculation and is used to verify that only valid configurations are considered when calculating the optimal product. The objective function is used to determine the quality of a solution [6]. However, the most expensive or cheapest product is often automatically determined by native algorithms that recursively select the most expensive/cheapest features. The resulting products are often incorrect because the approach does not consider constraints that heavily influence the selection of features (e.g., selecting the most expensive feature can later prevent the selection of two other most expensive features due to constraints). Hence, we introduce the attribute range analysis to improve the interactive configuration process by computing the actual impact after selecting or deselecting a feature.

Attribute Range Analysis computes the ranges for an attribute of interest. An attribute range is the minimum and maximum bound of the sum for all values of attributes that share the same name [42]. The input is an extended feature model and a numerical attribute of interest. Then, two individual optimization analyses are performed to find the products with the lowest and highest possible sum of all attribute values. The encoding for these problems is not trivial for SAT. Hence, we use SMT to compute the attribute ranges thanks to its support of arithmetic operations and numerical domains.

FeatureIDE releases the attribute range computation for the version 3.7.0. The analysis is utilized for the configuration process to interactively compute attribute ranges. For instance, Figure 6b shows the initial configuration with the computed attribute range for the extended feature model of Figure 6a. Figure 6c shows the price range after the user selects the monitor $MZ200$.

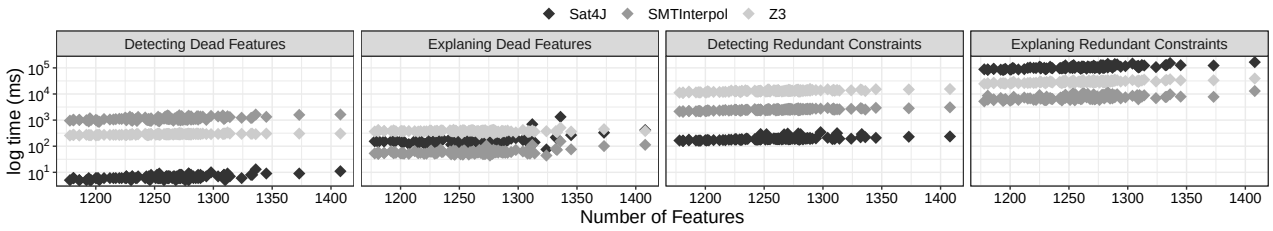


Figure 7: Evaluation results for the first experiment. From left to right: finding dead features, explaining dead features, finding redundant constraints, and explaining redundant constraints.

6 EMPIRICAL EVALUATION

SMT solvers do not only provide the foundation to solve more expressive problems but could prove as a viable replacement for SAT solvers in variability analyses. Thus, we are interested in the following research questions:

- *RQ1*: Are SMT solvers superior to SAT solvers regarding efficiency when performing variability analyses?
- *RQ2*: Is a combination of SAT and SMT solvers more efficient for variability analyses?
- *RQ3*: Is it efficient to calculate attribute ranges for partial configurations with SMT solvers?

We performed an empirical evaluation to answer *RQ1-RQ3* by conducting four experiments with more than 100 real-world systems which we describe in the following.

6.1 Setup

We choose JavaSMT [25] to integrate the SMT solvers SMTInterpol [11] and Z3 [13] into FeatureIDE [47]. JavaSMT allows us to easily add multiple SMT solvers into FeatureIDE without creating an implementation for each of them. We performed the following four experiments for our empirical evaluation:

Experiment 1 focuses on the analysis of feature models. We use 116 real-world models translated from the component definition language (CDL). All models are provided by a benchmark from Knüppel et al. [26] and contain between 1,178 to 1,408 features and 816 to 956 cross-tree constraints. We cover the detection and explanation of void feature models, core/dead features, false-optional features, and redundant constraints. All other analyses for only feature models were described for the completeness of the paper but are excluded from the evaluation as we expect similar results. At first, we measure the runtime for all solvers to detect all defects for a given model. Afterwards, we measure the runtime required to generate explanations for all found defects.

Experiment 2 focuses on the analysis of configurations. We use the large industrial feature model *automotive* provided by the partners of FeatureIDE as input for all analyses. The model contains 18,616 features and 1,369 cross-tree constraints. We cover the validity of configurations, automated decision propagation, and their explanations. We exclude the detection of conditionally core and dead features because they are already included in the automated decision propagation. We also exclude the error resolution analysis because we expect similar results as the generation of explanations. We start with a partial configuration containing one selected feature (i.e., the root feature), and 18,615 undecided features. We proceed by selecting a random undecided feature, validating the resulting configuration, and performing automated decision propagation. We set the timeout for the process to four hours and measure the runtime required for the validation and propagation separately for all solvers. Furthermore, we measure the runtime for generating explanations of automatically selected features.

Experiment 3 focuses on the analysis of preprocessor annotations. We use two small preprocessor example projects from FeatureIDE for our evaluation. We proceed by detecting all preprocessor annotations defects and explaining them directly. The runtime are separately measured for all solvers.

Experiment 4 focuses on calculating price attribute ranges for partial configurations. We use three real-world extended feature models *Sandwich*, *Bike*, and *PC* that we engineered based on existing configurators. The price attributes are distributed based on the values of their existing configurators. We refer the reader to [42] for detailed information on the engineering process. The models contain 19, 55, 377 features and 0, 0, 12 cross-tree constraints, respectively. In the experiment, we calculate the price range of randomly generated partial configurations. The runtime for Z3 is measured for the optimization. We excluded SMTInterpol because the solver does not provide the function to optimize variables.

All experiments are performed on a system with the following specifications OS: Windows 10, CPU: AMD Ryzen 7 1700X 8x3.4GHz, RAM: 16 GB DDR4-RAM, 2400 MHz. We use the following versions of tools: Java 1.8.0_201, Sat4J 2.3.5.v20130525, JavaSMT 2.2.0, SMTInterpol 2.5-66-g453d36e, Z3 4.6.0. Our implementation is based on FeatureIDE 3.6.0.

6.2 Results

We omitted similar results for many experiments. However, a git repository containing all results is publicly available[1].

Figure 7 shows the runtime in logarithmic scale of every solver for all 116 feature models of Experiment 1. We show only the results for detecting and explaining dead features and redundant constraints. As other results are similar, we omitted them for brevity. The results show that Sat4J finds defects multiple times faster than any of the SMT solvers. However, SMTInterpol is always faster when generating explanations, especially for redundant constraints.

Figure 8 shows the results for the automatic decision propagation and their explanations. Again, we omitted similar results for brevity. Each plot shows the runtime in logarithmic scale for all solvers. The results show, that the gap between Sat4J and the SMT

[1] https://github.com/Subaro/SMT-Based-Variability-Analyses-for-FeatureIDE

Figure 8: Results for the second experiment. Decision propagation on the left and their explanations on the right.

Figure 9: Results for the third experiment. Detecting dead code blocks on the left and explaining them on the right.

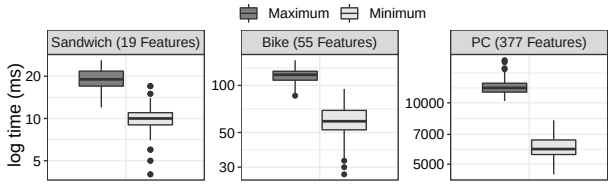


Figure 10: Results for the fourth experiment. Calculating attribute ranges for all three extended feature models.

solvers for solving satisfiability is growing compared to the results of Experiment 1. Surprisingly, the SMT solvers find explanations quite fast having again a slight advantage over Sat4J.

Figure 9 shows the results for our third experiment. The box plots show the runtime in logarithmic scale for all solvers on detecting and explaining dead code blocks. Again, we omitted similar results for brevity. The results for the detection of dead code blocks show an advantage for Sat4J when detecting defects. The results for the generation of explanations show that SMTInterpol is faster and Z3 is slower than Sat4J when generating explanations.

Figure 10 shows the results for our fourth experiment. The box plots show the logarithmic runtime of Z3 on calculating the minimum and maximum price range for each model. The results show that the effort to compute the ranges grows with the numbers of features reaching a runtime of about 12 seconds for the *PC* model. Furthermore, the computation for the minimum is generally twice as fast as the computation for the maximum.

6.3 Discussion

In the following, we discuss the results of our evaluation and use them to answer *RQ1-RQ3*.

RQ1: The results clearly show that Sat4J is superior regarding solving the evaluated satisfiability requests, including the detection of defects, the automated decision propagation, and detecting preprocessor annotations. Sat4J could solve some analyses about 100 times faster than each SMT solver. We can clearly see that SMTInterpol and Z3 are not superior regarding the efficiency for our experiments. However, using JavaSMT has a slight overhead compared to using the solver API directly and for many years now, SAT solvers are common, and, thus, typically used for variability analyses. Optimizations used by Sat4J such as the exploitation of the variable selection strategy or the utilization of filtering techniques can considerably improve the efficiency of an analysis [21]. We adapted the filtering technique for SMTInterpol and Z3. However, exploiting the variable selection strategy was not possible because the access to the native solvers is restricted in JavaSMT. Furthermore, research has to be conducted on whether analyses can be expressed with an SMT-encoding instead of a pure SAT-encoding to potentially improve the efficiency of SMT solvers.

RQ2: Our evaluation showed that the SMT solvers can improve the efficiency of finding explanations, and, thus, a combination of Sat4J and SMTInterpol or Z3 is more efficient than relying on Sat4J only. The only exception was Z3 that sometimes performed worse than Sat4J for generating explanations. We saw that Z3 has performance issues especially for small projects. Still, both SMT solvers had the biggest advantage over Sat4J when explaining redundant constraints finishing the analysis 12 times faster on average. However, we only evaluated the SAT solver Sat4J against two SMT solvers for a subset of all described analyses. Hence, a more detailed comparison between multiple SAT and SMT solvers covering all analyses is needed to fully answer this question.

RQ3: In our previous approach for the attribute range analysis [42], we did not fully utilize SMT theories. As a result, this did not scale for larger models, always exceeding the timeout of 24 hours for the *PC* model. We optimized the approach by using more SMT theories directly provided by the solvers. The results of Experiment 4 shows the large improvement of the new approach as we could compute ranges for the *PC* model within 12 seconds in general. Furthermore, it shows the large efficiency difference when exploiting an SMT-based encoding. This could also motivate the research of an SMT-based encoding for feature models to potentially improve the efficiency of SMT solvers for variability analyses. With our results, we conclude that the computation of attribute ranges is efficient for smaller models to support the interactive configuration process. However, further research is needed to identify the break-even point between the analysis runtime and the scale of feature models. In addition, the different computation times between minimum and maximum as well as the impact of different attribute distributions and varying domains for attributes need further investigation.

7 RELATED WORK

Variability Analyses The variability analyses introduced in the last decades are vast. Benavides et al. [6] published a systematic literature review covering 30 analyses and categorize the proposals for the different analyses based on the notation of feature models. The proposals are grouped by basic, cardinality-based, and extended feature models. Additionally, the analyses are classified into propositional logic, constraint programming, description logic, and others. Furthermore, they give an overview about the tools used for the constraint programming based analyses.

Chico et. al [43] compare various #SAT solvers by computing the number of all valid configurations for industrial systems. Their results show that most systems could be computed within acceptable ranges. However, they faced issues with the two largest systems and indicate that reducing the variability prior to the computation might be a necessary step.

SAT vs SMT on SAT-Based Analyses The only work regarding a direct comparison of SAT and SMT for variability analyses were published by Michel et al. [32]. The work from Michel et al. presents an approach to automate the configuration process for software product lines with the help of an SMT solver. They introduce translation rules to transform a given feature diagram into the input of their SMT solver. In difference to our comparison, they use SMT-based encoding of bit-vectors and arrays theories for their feature diagram transformation. In addition, instead of evaluating multiple SAT and SMT solvers they compared the SMT solver STP and his default backend SAT solver CryptoMiniSat. Michel et al. conclude that the optimizations internally executed by STP results in a twice as fast runtime as CryptoMiniSat.

SMT-Based Variability Analyses One of the most often given reasons for the usage of SMT solvers instead of SAT solvers in variability analyses is their expressiveness. Thus, many new analyses were introduced to tackle more complex problems.

Xiong et al. [51] introduce an alternative approach for conflict resolution, namely *range fix*, to fix constraint violations in software configurations. Their approach considers non-Boolean properties and constraints for configuration options and finding multiple explanations for a given violation. Furthermore, an evaluation was performed with the SMT solver Z3 showing an acceptable runtime for the interactive usage within configuration tools.

Arcaini et al. [2] showed the utilization of SMT solvers for the generation of test suites. The generated suites showed a guaranteed fault detection capability for feature models using distinguishing configurations. Such configurations can be used to differentiate between a valid and a faulty feature model. They exploited the capabilities of the SMT solver Yices and their evaluation showed that their approach has advantages over approaches that use all products or a subset of the products such as pairwise approaches.

Weckesser et al. [50] introduce a formalization of cardinality-based feature models. They use a combination of integer linear programming (ILP) and SMT solvers to automate consistency checking and anomaly detection. Their approach uses an ILP solver to detect unsatisfiable lower/upper bounds and an SMT solver to find unsatisfiable sub-ranges of cardinalities. The evaluation shows an exponential growth for the runtime of the SMT solver.

Nieke et al. [34, 35] propose temporal feature models (TFM) as a notion for evolution of feature models. The history of changes for a TFM helps to easily detect anomalies and to reduce the size of large explanations in particular for large models. Further, the authors introduce a method to reduce the entire TFM history to an SMT-problem and automatically find anomalies and explanations using an SMT solver. The results showed that their approach detects all anomalies, has an acceptable runtime, and significantly reduces the size of explanations.

Mauro et al. [31] introduce automated anomaly detection and the generation of respective explanations for context-aware product lines. Context-aware product lines introduce further configuration options by allowing the definition conditions that influence the selection of features. The authors use SMT solvers for their approach as SAT solvers are not expressive enough. The runtime was acceptable but still leaves room for improvement as no particular optimization has been performed.

Feature Attributes Optimization Benavides et al. [5] were the first to perform automated reasoning on software product lines using constraint programming. They introduce extended feature models and the optimization analysis with the aim to find optimal solutions. The difference between their and our optimization of feature attributes is that their approach is based on constraint programming (CP). Another approach for the optimization of feature attributes was introduced by Heijblom [18]. He translated extended feature models into integer problems and solved them using integer programming (IP) as an optimization technique. Furthermore, an empirical evaluation was performed to compare the runtime of CP vs IP. Heijblom concludes that IP is slightly inferior regarding efficiency but is generally preferred as less information is required.

8 CONCLUSION

We presented an overview about solver-based variability analyses in FeatureIDE. They were categorized into SAT-based feature model analyses, configuration analyses, and preprocessor analyses. Additionally, we proposed the usage of SMT solvers for the calculation of feature attribute ranges of partial configurations to assist the interactive configuration process. Most of the analyses are SAT-based and are traditionally performed by SAT solvers. However, the recent improvements of SMT and their higher expressiveness motivate the replacement of SAT solvers by SMT solvers.

We empirically answered the question of whether SAT solvers are replaceable by SMT solvers based on their efficiency for more than 100 real-world systems for a subset of the presented analyses. Our results showed that the SMT solvers we evaluated are only superior when generating explanations. Thus, we recommend SMT solvers for the generation of explanations while all other SAT-based analyses should be further delegated to SAT solvers.

Our empirical evaluation for the proposed attribute range analysis was performed with the SMT solver Z3 for three real-world feature models. Our results show that the computation of attribute ranges is efficient for small feature models.

Our comparison of SAT and SMT showed that further work is needed to establish SMT solvers as common alternatives in feature modelling tools. First, the impact of an SMT-based encoding for feature models, configurations, and presence conditions needs to be investigated. Second, SAT solvers were the first choice for many years of variability analyses and many optimizations were proposed. Thus, we need to adapt such optimizations for SMT solvers to improve their efficiency. In addition, further research for the computation of attribute ranges is needed. We need to identify the break-even point between the analysis runtime and the scale of feature models. Also, the impact of feature attribute values and their distribution needs to be further investigated. This includes the relevance of unnecessary attribute values as certain attributes only need to be applied to a restricted group of features. The runtime for the computation of minimum and maximum bounds also showed significant differences and need to be further investigated.

REFERENCES

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*.

[2] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. 2015. Generating Tests for Detecting Faults in Feature Models. 1–10. https://doi.org/10.1109/ICST.2015.7102591

[3] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. 7–20.

[4] Don Batory, David Felipe Benavides Cuevas, and Antonio Ruiz Cortés. 2006. Automated analysis of feature models: challenges ahead. *Communications of the ACM-Software product line, 49 (12), 45-47.* (2006).

[5] David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. 2005. Automated Reasoning on Feature Models. 491–503.

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

[7] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. 7:1–7:8. https://doi.org/10.1145/2430502.2430513

[8] Daniel Le Berre. [n. d.]. SAT4J: A Satisfiability Library for Java. Website. Available online at http://www.sat4j.org/; visited on November 9th, 2010.

[9] Paul Maximilian Bittner, Thomas Thüm, and Ina Schaefer. 2019. SAT Encodings of the At-Most-k Constraint – A Case Study on Configuring University Courses. To appear.

[10] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, J Henk Obbink, and Klaus Pohl. 2001. Variability issues in software product lines. In *International Workshop on Software Product-Family Engineering*. Springer, 13–21.

[11] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. 2012. SMTInterpol: An interpolating SMT solver. In *International SPIN Workshop on Model Checking of Software*. Springer, 248–254.

[12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. 173–182. https://doi.org/10.1145/2110147.2110167

[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[14] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54, 9 (2011), 69–77.

[15] Amador Durán, David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2017. FLAME: a formal framework for the automated analysis of software product lines validated by automated specification testing. *Software & Systems Modeling* 16, 4 (2017), 1049–1082.

[16] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2015. Analysing the KConfig Semantics and its Analysis Tools. 45–54. https://doi.org/10.1145/2814204.2814222

[17] Timo Günther. 2017. *Explaining Satisfiability Queries for Software Product Lines*. Ph.D. Dissertation. Master's thesis, Technische Universität Braunschweig.

[18] Richard Heijblom. 2013. Potential of Integer Programming for Optimization Analysis of Extended Feature Models. (2013).

[19] Mikolas Janota. 2008. Do SAT solvers make good configurators?. In *SPLC (2)*. 191–195.

[20] Richard C Johnson and Henry Jasik. 1984. Antenna engineering handbook. *New York, McGraw-Hill Book Company, 1984, 1356 p. No individual items are abstracted in this volume.* (1984).

[21] Andreas Kaiser and Wolfgang Küchlin. 2001. Detecting inadmissible and necessary variables in large propositional formulae. In *University of Siena*. Citeseer.

[22] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Software Engineering Institute.

[23] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Gerard Jounghyun Kim, and Euiseob Shin. 1998. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering* 5, 1 (Jan. 1998), 143–168.

[24] Kyo C Kang, Jaejoon Lee, and Patrick Donohoe. 2002. Feature-oriented product line engineering. *IEEE software* 19, 4 (2002), 58–65.

[25] Egor George Karpenkov, Karlheinz Friedberger, and Dirk Beyer. 2016. JavaSMT: A unified interface for SMT solvers in Java. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 139–148.

[26] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2018. Is There a Mismatch between Real-World Feature Models and Product-Line Research?, Matthias Tichy, Eric Bodden, Marco Kuhrmann, Stefan Wagner, and Jan-Philipp Steghöfer (Eds.). 53–54. https://dl.gi.de/20.500.12116/16312

[27] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. 2016. Explaining Anomalies in Feature Models. 132–143. https://doi.org/10.1145/2993236.2993248

[28] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. 2018. Propagating Configuration Decisions with Modal Implication Graphs. 898–909. https://doi.org/10.1145/3180155.3180159

[29] Jia Liu, Don Batory, and Christian Lengauer. 2006. Feature Oriented Refactoring of Legacy Applications. 112–121. https://doi.org/10.1145/1134285.1134303

[30] Mike Mannion. 2002. Using First-Order Logic for Product Line Model Validation. 176–187.

[31] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. 2017. Anomaly Detection and Explanation in Context-Aware Software Product Lines. 18–21. https://doi.org/10.1145/3109729.3109752

[32] Raphaël Michel, Arnaud Hubaux, Vijay Ganesh, and Patrick Heymans. 2012. An SMT-based approach to automated configuration. In *SMT Workshop 2012 10th International Workshop on Satisfiability Modulo Theories SMT-COMP*. Citeseer, 107.

[33] Mukelabai Mukelabai, Damir Nešić, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. 2018. Tackling combinatorial explosion: a study of industrial needs and practices for analyzing highly configurable systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 155–166.

[34] Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzke. 2018. Anomaly Analyses for Feature-Model Evolution. 188–201. https://doi.org/10.1145/3278122.3278123

[35] Michael Nieke, Christoph Seidl, and Sven Schuster. 2016. Guaranteeing configuration validity in evolving software product lines. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 73–80.

[36] Linda Northrop and P. Clements. 2007. A Framework for Software Product Line Practice, Version 5.0. *SEI* (2007).

[37] Jörg Pleumann, Omry Yadan, and Erik Wetterberg. 2011. Antenna: An Ant-to-End Solution For Wireless Java. Website. Available online at http://antenna.sourceforge.net/; visited on November 22nd, 2011.

[38] Klaus Schmid and Isabel John. 2004. A customizable approach to full lifecycle variability management. *Science of Computer Programming* 53, 3 (2004), 259–284.

[39] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. 2016. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. 667–678. https://doi.org/10.1145/2884781.2884823

[40] Sergio Segura. 2008. Automated Analysis of Feature Models Using Atomic Sets.. In *SPLC (2)*. 201–207.

[41] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. 2010. Efficient Extraction and Analysis of Preprocessor-Based Variability. 33–42. https://doi.org/10.1145/1868294.1868300

[42] Joshua Sprey and Chico Sundermann. 2018. Computing Attribute Ranges for Partial Configurations with JavaSMT. (2018).

[43] Chico Sundermann, Thomas Thüm, and Ina Schaefer. 2020. Evaluating #SAT Solvers on Industrial Feature Models. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*. ACM, 9.

[44] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. 47–60. https://doi.org/10.1145/1966445.1966451

[45] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. 47, 1 (June 2014), 6:1–6:45. https://doi.org/10.1145/2580950

[46] Thomas Thüm, Don Batory, and Christian Kästner. 2009. Reasoning about Edits to Feature Models. 254–264. https://doi.org/10.1109/ICSE.2009.5070526

[47] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. 79, 0 (Jan. 2014), 70–85. https://doi.org/10.1016/j.scico.2012.06.002

[48] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. 2011. Abstract Features in Feature Modeling. 191–200. https://doi.org/10.1109/SPLC.2011.53

[49] Thomas von der Maßen and Horst Lichter. 2004. Deficiencies in feature models. In *workshop on software variability management for product derivation-towards tool support*, Vol. 44. 21.

[50] Markus Weckesser, Malte Lochau, Thomas Schnabel, Björn Richerzhagen, and Andy Schürr. 2016. Mind the gap! Automated anomaly detection for potentially unbounded cardinality-based feature models. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 158–175.

[51] Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. 2012. Generating range fixes for software configuration. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 58–68.

[52] Wei Zhang, Haiyan Zhao, and Hong Mei. 2004. A Propositional Logic-Based Method for Verification of Feature Models. *Formal Methods and Software Engineering* (2004), 115–130.

[53] Roman Zippel. 2017. KConfig Documentation. Website. Available online at http://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt; visited on May 10th, 2017.