Anomaly detection in Context-aware Feature Models

Jacopo Mauro
mauro@imada.sdu.dk
University of Southern Denmark

ABSTRACT

Feature Models are a mechanism to organize the configuration space and facilitate the construction of software variants by describing configuration options using features, i.e., a name representing a functionality. The development of Feature Models is an error prone activity and detecting their anomalies is a challenging and important task needed to promote their usage.

Feature Models have been extended with context to capture the correlation of configuration options with contextual influences and user customizations. Unfortunately, this extension makes the task of detecting anomalies harder. In this paper, we formalize the anomaly analysis in Context-aware Feature Models and we show how Quantified Boolean Formula (QBF) solvers can be used to detect anomalies without relying on iterative calls to a SAT solver. By extending the reconfigurator engine HyVarRec, we present findings evidencing that QBF solvers can outperform the common techniques for anomaly analysis on some instances.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Software evolution**.

KEYWORDS

Feature Model Anomalies, SMT solver

ACM Reference Format:
Jacopo Mauro. 2021. Anomaly detection in Context-aware Feature Models. In *15th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS'21), February 9–11, 2021, Krems, Austria.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3442391.3442405

1 INTRODUCTION

Software Product Lines (SPLs) are a technology for large-scale reuse for a set of closely related software systems [35], which allows companies to customize their software systems through configuration. At the core of SPL engineering is the modeling of common and variable parts of software systems. On the conceptual side, common and variable parts are described in terms of *features*, which represent the configurable functionality of a system [20]. Features are often not independent of each other and to represent the relation between features a *variability model* can be employed. Among the most popular variability models are Feature Models (FMs) [20] which often are represented visually in feature diagrams, a tree-like notation that structures features hierarchically.

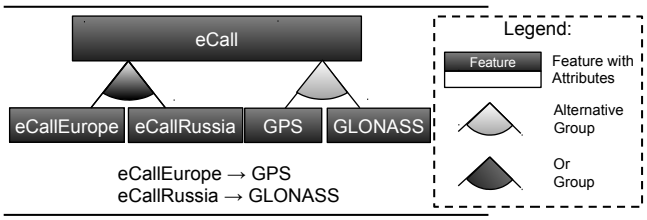

Figure 1: Example of Feature Diagram.

As an example, Figure 1 shows the feature diagram that represents the emergency call feature `eCall` for a car [27]. The `eCall` service is provided by the `eCallEurope` or the `eCallRussia` feature. The `eCallEurope` relies on the positioning data of a `GPS` system while the `eCallRussia` service instead relies on positioning data provided by the `GLONASS` satellite system. The FM in Figure 1 graphically represents the dependencies between the features. In particular, to select the feature `eCall` the feature `eCallEurope` or the feature `eCallRussia` must be selected (or-group). Instead, it is required to select one and only one among the features `GPS` and `GLONASS` (alternative-group). Additional constraints, dubbed Cross-Tree Constraints (CTC), are added. In Figure 1, CTC are implications and require that when the feature `eCallEurope` (resp. `eCallRussia`) is selected, `GPS` (resp. `GLONASS`) is also selected.

A product (or configuration) of the FM is valid if it selects the features without contradicting any of the constraints imposed by the FM (both the structural tree constraints and the Cross-Tree Constraints). Hence, a valid product describes one member of the SPL on a conceptual level without regard to its implementation. On the implementation side, features are realized using *realization artifacts*, such as code or documentation artifacts.

Recently, in [29] the notion of FM has been extended to encompass the possibility to link the validity of one configuration to external factors. The new Context-aware Feature Models (CaFMs) allow expressing SPL that are adaptable to the environment where they are deployed and take user preferences into account. The idea behind CaFM is the possibility to use context variables to represent the external factors and impose constraints between the value of these variables and the features. For instance, for the FM in Figure 1, let us assume that the car manufacturer by law has to provide in the cars sold in Russia and only in those the `eCallRussia` feature. To do so, the FM can be enriched by a new context variable `Location` that can be externally set to `True` if and only if the car is sold in Russia. To bound the context variable to the features, it is possible to add the constraint that imposes the selection of `eCallRussia` if and only if the `Location` is set to `True`.

VaMoS'21, February 9–11, 2021, Krems, Austria

ACM ISBN 978-1-4503-8824-5/21/02…$15.00
https://doi.org/10.1145/3442391.3442405

In common SPL engineering, errors in the creation of (Ca)FMs may happen and for this reason, procedures to determine and explain them are essential. For example, it is of paramount importance to understand if a change of a (Ca)FM makes it void, i.e., does not allow the possibility to have a valid configuration. Many anomaly analyses and tools have been proposed for FM [4, 5, 22, 39] but, as initially investigated in [28], the introduction of context brings some changes. In particular, the voidness analysis that checks if a valid configuration is always possible becomes more complex.

In this paper we first formally define the anomaly analysis for CaFM and their complexity. We then show how Quantified Boolean Formula (QBF) solvers can be used to detect anomalies without relying on (possibly exponential) invocations of SAT solvers. Preliminary results obtained by extending the HyVarRec reconfigurator engine [29] show that the usage of a QBF solvers can improve over the naive strategy of using a SAT-based solver on some of the randomly generated CaFMs.

Structure of the paper In Section 2 we introduce FM and their anomaly analysis. In Section 3 we formalize the notion of CaFM while in Section 4 we formalize and describe how to perform their anomaly analysis. In Section 5 we introduce the HyVarRec reconfiguration engine and we present some initial findings proving that the usage of QBF can be useful for performing the anomaly analysis. We draw some concluding remarks in Section 6.

2 BACKGROUND

In this section, we recap the notion of Feature Model (FM). We then list the most common FM analyses and mention the technologies used to solve them.

Among the different representations of feature models, in the following we adopt the *propositional formula* representation [4, 7, 25, 38] that is not as visual as the feature diagram representation depicted in Figure 1, but allows a more concise formal treatment and is expressive enough to capture complex constraints [21].

Definition 2.1. A Feature Model is a pair $\mathcal{M} = (\mathcal{F}, \phi)$ where:

- $\mathcal{F}$ is a set of features, and
- ϕ is a propositional formula where the variables x are feature names.

$$\phi \ ::= \ x \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \neg\phi.$$

The propositional formula ϕ over a set of features $\mathcal{F}$, represents the feature models whose products are sets $\{f_1, ..., f_n\} \subseteq \mathcal{F}$ $(n \geq 1)$ such that ϕ is satisfied by assigning value true to the variables f_i $(1 \leq i \leq n)$ and false to all other variables.

As an example, the FM introduced in Figure 1 can be represented by the pair $(\mathcal{F}, \phi)$ as defined in Example 2.2.

The formula ϕ in the first line makes sure that the feature `eCall` is selected since it is the root of the FM. The second and third lines capture the constraints imposed by the or/alternative-groups. The fourth and fifth lines state that `eCall` is the parent feature of `eCallEurope`, `eCallRussia`, `GPS`, `GLONASS` and, therefore, it must be selected if its children are selected. The last line reports the Cross-Tree Constraints.

Example 2.2.

$$\begin{aligned}\mathcal{F} =& \{\texttt{eCall}, \texttt{eCallEurope}, \texttt{GPS}, \texttt{eCallRussia}, \texttt{GLONASS}\}\\ \phi =& \texttt{eCall} \wedge\\ & \texttt{eCall} \rightarrow (\texttt{eCallEurope} \vee \texttt{eCallRussia}) \wedge\\ & \texttt{eCall} \rightarrow (\texttt{GPS} \vee \texttt{GLONASS}) \wedge \neg(\texttt{GPS} \wedge \texttt{GLONASS}) \wedge\\ & \texttt{eCallEurope} \rightarrow \texttt{eCall} \wedge \texttt{eCallRussia} \rightarrow \texttt{eCall} \wedge\\ & \texttt{GPS} \rightarrow \texttt{eCall} \wedge \texttt{GLONASS} \rightarrow \texttt{eCall} \wedge\\ & \texttt{eCallEurope} \rightarrow \texttt{GPS} \wedge \texttt{eCallRussia} \rightarrow \texttt{GLONASS}\end{aligned}$$

Given a formal definition of FMs, we can introduce their anomaly analysis. In particular, for conciseness, we report only the major ones referring to the interested reader the survey [5] for more details.

- *Valid Product.* Given a FM $(\mathcal{F}, \phi)$ and a product $\mathcal{P} \subseteq \mathcal{F}$, it is checked if the product is valid, i.e., if the literals in $\mathcal{P} \cup \{\neg f . f \in \mathcal{F} \setminus \mathcal{P}\}$ satisfy ϕ.
- *Voidness.* Given a FM $(\mathcal{F}, \phi)$, it is checked if it does not allow products, i.e., if ϕ is not satisfiable. This is probably the most important analysis because having a void FM means that no possible implementation can be obtained for the SPL.
- *Dead Features.* Given a FM $(\mathcal{F}, \phi)$ and a feature $f \in \mathcal{F}$, the feature f is dead if it can never be selected, i.e., if there is no valid product that contains it or, equivalently, if $f \wedge \phi$ is not satisfiable. The dead feature analysis retrieves all the dead features of a FM. Note that, in general, dead features should be avoided for maintainability purposes since they can never be used in a product.
- *False Optional Features.* In FM features can be marked as mandatory or optional.[1] Given a FM $(\mathcal{F}, \phi)$ and a feature $f \in \mathcal{F}$ marked as optional, the feature f is false optional when it is available in every possible product, i.e., when $\neg f \wedge \phi$ is not satisfiable. Given a subset of features marked as optional, the false optional feature analysis retrieves all the false optional ones. In general, false optional should be marked as mandatory for maintainability purposes.
- *Redundancies.* Redundancies are constraints that do not add information over existing ones. Redundancies can decrease maintainability but can also improve the readability and comprehensibility of the model. Using the propositional formula representation, a FM has redundancies if the formula ϕ contains redundant clauses (i.e., clauses that can be removed without altering the set of all the products).

From the computational complexity point of view, the analysis of checking the validity of a product is polynomial in the size of the formula ϕ, while the other analysis are NP-hard or coNP-hard [34].

Different tools are used to perform the analysis and among the complete ones (i.e., tools that can prove the existence or the non

[1] We would like to remark that the definition of optional feature varies in the literature. For instance in [4] a feature is optional if it can be selected and deselected, while in FeatureIDE [30] a feature is optional if it can be deselected when its parent in the feature diagram representation is selected. In this paper, we require the users to decide which features are defined as optional or not and follow the semantics of Batory [4].

Listing 1: Dead Features Analysis using a SAT solver.

```
1  def dead_feature(F,φ)
2    push(φ)
3    fs = F
4    dead_features = ∅
5    while fs ≠ ∅:
6      f = fs.pop
7      push(f)
8      if not checkSat():
9        dead_features.add(f)
10     else:
11       fs = fs − getModel()
12     pop()
13   return dead_features
```

existence of an anomaly) often the most used ones involve propositional logic-based tools such as SAT solvers, binary decision diagram, Constraint Programming solvers, SMT solvers, or description logic reasoners [5].

The most used tools are based on SAT solvers [26], i.e., tools that check whether a Boolean propositional formula is satisfiable returning an assignment that makes the formula true, if any. For FM analysis such as the voidness one, the SAT solver backend is called only once, while other analysis require more than one invocation. For example, to determine all the dead features, it is possible to iteratively call a backend solver for every feature of the FM. This task is facilitated in modern incremental SAT solvers that support the possibility to perform push and pop operations to dynamically stack and retract formulas, thus avoiding repeating already performed computations. To exemplify the procedure, consider the pseudocode of Listing 1 that implements the analysis of dead features. We assume that the SAT solver can offer the following primitives:

- pop and push to remove and add a formula on the stack;
- checkSat that checks if the conjunction of the formulas on the stack is satisfiable;
- getModel that retrieves the set of the positive literals of the last satisfiable solution computed.

Given a FM $(\mathcal{F}, \phi)$, the idea of this algorithm is to start with ϕ (Line 2) and then in a loop check if a given feature is dead. At every interaction of the loop, a feature is selected (Line 6), pushed to the stack (Line 7), and the SAT solver invoked to check if $\phi \wedge f$ is satisfiable (Line 7). If the formula is not satisfiable the feature added is dead and can be added to the set of dead features (Line 9). If not, at Line 11 the features that have been selected are removed from the set of the features to check (if a feature is selected is not dead). Finally, in Line 12, the formula added in Line 7 is removed from the stack so the procedure can continue checking the next feature, if any.

In the remaining part of the paper, we will use not only SAT solvers but also Quantified Boolean Formula (QBF) and Satisfiability Modulo Theory (SMT) solvers. QBS solvers [14] or QSAT solvers, generalize a SAT solver enabling to check the satisfiability of quantified Boolean formulas, thus being able to process formulas with universal quantifiers. SMT solvers [8] also extend SAT solvers by generalizing variables using predicates from a variety of underlying theories, thus allowing for instance to support integer variables and arithmetic constraints. In this paper, in particular, the experiments rely on the state-of-the-art SMT solver Z3 [9]. This solver supports a huge variety of theories and can handle also formulas involving universal quantifiers, thus making it also a QBF solver. In particular, Z3 uses several approaches to handle quantifiers like pattern/model-based quantifier instantiation or quantifier elimination [42].

3 CONTEXT-AWARE FEATURE MODELS

In this section, we formalize the notion of Context-aware Feature Model (CaFM). Following [29], a context can be considered as a variable that someone externally (e.g., the user of the software or the environment) can set. These new variables impose constraints over features and therefore on the products that can be obtained.

Without loss of generality,[2] for presentation' sake, we can restrict ourselves to consider context variables that can take only two values: `true` or `false`. In this way, the notion of context and feature almost coincide, with the only difference that the value of features can be controlled by the developer of the CaFM, while the value of contexts is decided externally.

Definition 3.1. A Context-aware Feature Model (CaFM) is a tuple $(C, \mathcal{F}, \phi)$ where:

- C is a set of context,
- $\mathcal{F}$ is a set of features, and
- ϕ is a propositional formula where the variables x are feature or context names.

$$\phi \ ::= \ x \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \neg\phi.$$

Fixed the value of the context variables in C, a propositional formula ϕ over a set of features $\mathcal{F}$, represents the feature models whose products are sets $\{f_1, ..., f_n\} \subseteq \mathcal{F}$ $(n \geq 1)$ such that ϕ is satisfied by assigning value true to the variables f_i $(1 \leq i \leq n)$ and false to all other variables.

Example 3.2. Consider the FM introduced in Figure 1 and its propositional representation shown in Example 2.2. As discussed in Section 1, imagine that the car manufacturer by law has to provide in the cars sold in Russia the `eCallRussia` feature, otherwise the `eCallEurope` feature. To capture this situation, the FM can be enriched by a new context variable `Location` that externally can be set to `true` if the car is sold in Russia. Then the CaFM modeling this situation is the tuple $(C, \mathcal{F}, \phi')$ where

$$\begin{aligned} C &= \{\texttt{Location}\} \\ \mathcal{F} &= \{\texttt{eCall}, \texttt{eCallEurope}, \texttt{GPS}, \texttt{eCallRussia}, \texttt{GLONASS}\} \\ \phi' &= \phi \wedge \texttt{Location} \rightarrow \texttt{eCallRussia} \wedge \\ &\quad \neg\texttt{Location} \rightarrow \neg\texttt{eCallRussia} \end{aligned}$$

where ϕ is the formula introduced in Example 2.2.

The introduction of the context forbids the selection of `eCallRussia` when the context variable is set to `false`, while it forces its selection if `Location` is set to `true`. Therefore, the set of valid products available depends on the value of the context `Location`.

[2] In [29] contexts are variables that take values on finite domain. This can easily be modeled with a set of variables having a Boolean domain.

4 ANALYSIS OF CONTEXT-AWARE FEATURE MODELS

In this section, we describe how the common analyses for FMs are lifted to the CaFM, discussing also how QBF solvers can be used to solve them when needed.

4.1 Valid Product

With the introduction of contexts, the natural extension of the product validity check is to verify if the product is valid in a given context.

Definition 4.1 (Valid Product). Given a CaFM $(\mathcal{C}, \mathcal{F}, \phi)$, a context assignment $\mathcal{D} \subseteq \mathcal{C}$ and a product $\mathcal{P} \subseteq \mathcal{F}$, $\mathcal{P}$ is valid in $\mathcal{D}$ when the literals in $\mathcal{D} \cup \{\neg c . c \in \mathcal{C} \setminus \mathcal{D}\} \cup \mathcal{P} \cup \{\neg f . f \in \mathcal{F} \setminus \mathcal{P}\}$ satisfy ϕ.

It is easy to see that the introduction of context does not have a huge impact on the techniques used to decide whether a product is valid or not. Indeed, similarly to what happens for normal FMs, to validate a product of a CaFM the context must be fixed and, thus, the validation of the product requires checking if the ground formula of the CaFM is true. The complexity of this operation is therefore polynomial w.r.t. the size of the formula representing the CaFM.

4.2 Voidness

The straightforward way to extend the notion of voidness to the context-aware case is to say that a CaFM is void if there exists a context that does not admit a valid product.

Definition 4.2 (Voidness). A CaFM $(\mathcal{C}, \mathcal{F}, \phi)$ is void if $\exists \mathcal{C} . \forall \mathcal{F} . \neg \phi$.

Notation: given a set of variables $\mathcal{X} = \{x_i, \ldots, x_n\}$ and a formula ϕ, we write with $\exists \mathcal{X} . \phi$ the existential closure of ϕ, i.e., $\exists x_1 \ldots \exists x_n . \phi$. Similarly, we shorten the formula $\forall x_1 \ldots \forall x_n . \phi$ with $\forall \mathcal{X} . \phi$.

From the definition of voidness, we can derive that checking it is a problem that belongs to the complexity class $\Sigma_2^P = NP^{NP}$ [34], i.e., the class of problems that can be solved by calling a non-deterministic polynomial time algorithm able to use a non-deterministic polynomial time oracle. Therefore, since checking the voidness of a FM is an NP-complete problem, unless the polynomial hierarchy collapses, checking the voidness of a CaFM is more difficult than checking it for FM. As a consequence, a SAT solver is not enough to perform the voidness check in only one call, leaving the possibility of using a QBF solver instead. The implementation of the voidness check with a QBF solver is indeed straightforward: QBF solvers support universally quantified formula and therefore it is possible to check if $\exists \mathcal{C} . \forall \mathcal{F} . \neg \phi$ is satisfiable directly.

The voidness of a CaFM can instead be checked by using iteratively a SAT solver as shown in Listing 2. The idea behind this procedure is to loop over any possible context combinations and check if there exists one in which the resulting FM is void. In Listing 2 this is started by pushing on the stack the original formula ϕ (Line 13) and call the check procedure (Line 14). The check recursive procedure takes each context variable c at a time and tries to set the context first to true (Line 7) and later to false (Line 10) to allow the exploration of any possible combinations of contexts. The push and pop operations of the SAT solver are used to be able to reuse as much as possible the work already performed by the SAT solver. When all the context variables are grounded, the check procedure checks if the conjunction of the formulas on the stack is satisfiable (Line 3). If not, a void FM has been found and the analysis can be stopped. If no void FM is found, the CaFM is not void and this can be returned (Line 15).

Listing 2: FM Void Analysis using a SAT solver

```
1  def check(cs):
2    if cs = []:
3      if not checkSat():
4        print("CaFM is void")
5        exit(1)
6    else:
7      push(cs[0]) # context variable set to true
8      check(cs[1:])
9      pop()
10     push(¬cs[0]) # context variable set to false
11     check(cs[1:])
12     pop()

13 push(φ)
14 check(C)
15 print("CaFM not void")
```

Note that the code in Listing 2 performs up to $2^{|C|}$ invocations to the checkSat procedure, which solves an NP-complete problem.

4.3 Dead Features

In CaFM, a straightforward extension of the dead feature concept is to consider a feature dead if, for each context, it cannot be selected. This leads to the following definition of dead feature.

Definition 4.3 (Dead feature). Given a CaFM $(\mathcal{C}, \mathcal{F}, \phi)$, a future $f \in \mathcal{F}$ is dead if $\neg \exists \mathcal{C} . \exists \mathcal{F} . \phi \wedge f$.

Similarly to what happens for the normal FM, this formula can be easily checked with a SAT solver. To check all the features, the push and pop of the feature variables can be used to iteratively call a SAT solver as done in Listing 1. Hence, the problem of finding if a feature is dead is still a coNP-complete problem and checking if there is at least a dead feature is a problem that belongs to the class $\Delta_2^P = P^{NP}$, i.e., the class of problems that can be solved by calling a polynomial amount of times a non-deterministic polynomial time oracle like a SAT solver.

We would like to note that in case a QBF solver is available, it is possible to exploit it to find out if there is a dead feature in a (Ca)FM in just one call.[3] This requires the addition of an auxiliary variable for every feature. Let us suppose that given a feature f its fresh auxiliary variable is $\mathsf{aux}(f)$ and $\mathcal{F}_{aux} = \{\mathsf{aux}(f) . f \in \mathcal{F}\}$. A feature for a CaFM is dead if the following formula is satisfiable.

$$\exists \mathcal{F}_{aux} . \mathtt{OnlyOne}(\mathcal{F}_{aux}) \wedge \forall \mathcal{C} . \forall \mathcal{F} . (\bigwedge_{f \in \mathcal{F}} \mathsf{aux}(f) \rightarrow f) \rightarrow \neg \phi$$

where

$$\mathtt{OnlyOne}(x_1, \ldots, x_n) = \bigvee_{1 \leq i \leq n} (x_i \wedge (\bigwedge_{1 \leq j \leq n, i \neq j} \neg x_j)$$

[3] Note that checking the satisfiability of a QBF is a PSPACE-complete problem and there is no guarantee that a call to a QBF solver performs better than an exponential number of calls to a SAT solver.

Listing 3: Dead Features Analysis guided by a SAT solver.

```
1 push(φ)
2 fs = F
3 while fs ≠ ∅:
4   push(⋁ fs)
5   if not checkSat():
6     return fs # fs are all dead features
7   fs = fs \ getModel()
8 return ∅
```

In this formula, intuitively, the auxiliary variable $\mathsf{aux}(f)$ is used to force one feature f to be always selected in the universally quantified formula. The `OnlyOne` predicate enforces one and only one auxiliary variable to be true. The universally quantified formula instead checks that for any possible contexts and any possible selection of features, the selection of the feature f for which $\mathsf{aux}(f)$ is true leads always to a void product. If this formula is satisfiable, then at least one $\mathsf{aux}(f)$ is true and the corresponding feature f is a dead feature.

Differently from the standard analysis exemplified in Listing 1, another possible strategy to locate dead features could be to let the SAT solver guide the search and prune the features that are not dead as shown in Listing 3. The idea behind this algorithm is to start with the original formula ϕ and add the disjunction of features that have not been proven yet to belong to a product (Line 4). When the call to the SAT solver (Line 3) proves that the formula is unsatisfiable, none of the features added in the last iteration of the while loop can be selected and therefore all of them are returned as dead feature (Line 6). Otherwise, the selected feature can be excluded (Line 7) and the process can continue until either we reach an unsatisfiable formula or we prune all the features to check.

4.4 False Optional Features

For CaFM, the notion of false optional is naturally extended as a feature that i) is marked as optional and ii) for all the contexts and all the products it cannot be deselected. Formally:

Definition 4.4 (False optional). Given a CaFM $(C, \mathcal{F}, \phi)$, a future $f \in \mathcal{F}$ marked as optional is false optional if $\neg\exists C \,.\, \exists \mathcal{F} \,.\, \phi \wedge \neg f$.

Similarly to what is done for detecting dead features, checking all the false optional features can be done by calling iteratively a SAT solver, either by removing one feature at a time or by using the pruning technique presented in Listing 3. The complexity of finding false features in CaFM is the same as the one for FM.

Even in this case, it is also possible to use a quantified solver to detect if there are any false optional features in just one call. Let us assume that that given a feature f marked as optional $\mathsf{aux}(f)$ is an auxiliary variable. Let us consider $\mathcal{F}_{aux}$ the set containing all the auxiliary variables corresponding to features marked as optional. The existence of a false optional feature can be proven by checking the satisfiability of the following formula.

$$\exists \mathcal{F}_{aux} \,.\, \mathsf{OnlyOne}(\mathcal{F}_{aux}) \wedge \forall C \,.\, \forall F \,.\, (\bigwedge_{f \in \mathcal{F}} \mathsf{aux}(f) \rightarrow \neg f) \rightarrow \neg\phi$$

4.5 Redundancies

A constraint is redundant if it can be removed without altering the set of valid products for any possible context combinations in a CaFM. Formally:

Definition 4.5 (Redundancy). Given a CaFM $(C, \mathcal{F}, \phi \wedge \phi')$, ϕ is redundant if $\exists C . \exists \mathcal{F} . \neg(\phi' \rightarrow \phi)$ is unsatisfiable.

As a consequence, testing if an instance does not contain any redundant clause is an NP-complete problem [23][4] and the techniques used to check redundancies for FMs can be reused for CaFMs.

5 EVALUATION

In this section, we briefly introduce the CaFM analyzer HyVarRec and present the findings on the usage of a QBF solver for the analyses of randomly generated CaFM.

5.1 HyVarRec

HyVarRec [29] is a reconfiguration engine for CaFM that relies on the SMT solver Z3 [9] to find a valid product of a CaFM that minimizes user-defined metrics. Originally intended for the reconfiguration of CaFMs in presence of context changes, HyVarRec has been extended to support the voidness check [28] and later used by DarwinSPL[5] to provide explanations for anomalies [33].

We have extended HyVarRec to support the checking of dead and false optional features by using Z3's QBF solver and the SAT guided pruning technique introduced in Listing 3. In the following we use the term *Iterative* to refer to approaches that call iteratively a SAT solver (e.g., Listing 1 and 2), *Forall* for the approach that uses the QBF solver, and *Pruning* for the approach that uses a SAT solver to guide the pruning of the features for the feature analysis check (e.g., Listing 3). For the *Forall* approach, HyVarRec uses the default tactics implemented by Z3 to solve quantified formulas.[6] The feature analysis is performed by first trying to find the dead features and later the false optional ones. As an optimization, the features that during the detection of dead features were found to be deselected for a valid product were not considered for the false optional feature detection. The dead and false positive feature analysis can be stopped by HyVarRec as soon as one anomaly is discovered or when all anomalies are detected.

HyVarRec is publicly available at https://github.com/HyVar/hyvar-rec and supports the possibility to define features encoded either by integer values or directly as Boolean values. HyVarRec also supports the possibility to use attributes and context variables that can take values in finite domain integer sets.

5.2 Methodology

To the best of our knowledge, there is no established benchmark for CaFMs. To be able to compare the different approaches of anomaly explanation we have created a CaFM random instance generator[7]

[4] Note that modern SAT solvers often remove automatically some form of redundancy as a pre-processing step [12]. They can therefore detect automatically some redundancies without degradation of performances.
[5] https://gitlab.com/DarwinSPL/DarwinSPL
[6] https://stackoverflow.com/questions/20682763/z3-does-qe-tactic-preserve-equivalence-or-only-equisatisfiability/20719090
[7] https://github.com/HyVar/hyvar-rec/tree/master/test/cafm_generator

Table 1: Number of times a voidness check approach is the best by varying the number of context variables (features number = 250).

Contexts	Result	Forall	Iterative	Total
6	Void	11	34	45
	Not Void	12	43	55
8	Void	25	31	56
	Not Void	1	43	44
10	Void	26	20	46
	Not Void	2	52	54
12	Void	31	31	62
	Not Void	1	37	38
14	Void	33	25	58
	Not Void		42	42
Total		142	358	500

by relying on a SAT formula generator [13][8]. Given a target number of context variables and features, the SAT formula generator was used to generate propositional formulas with clauses having 3 literals taking the variables according to the uniform distribution. According to the desired number of context, the first literals were considered context variables while the remaining literals were considered features. The number of clauses of the propositional formula (and therefore its hardness) was controlled by a parameter expressing the ratio between the number of clauses and the number of features. The clauses having only context variable literals, if any, were removed to avoid restricting possible context combinations.

We ran a first set of experiments considering CaFM having 250 features and varying the number of context variables in the set $\{6, 8, 10, 12, 14\}$. For every number in this set, we generated 100 instances, varying the ratio of clauses to number of features between 5 and 6. For the feature anomaly analysis, we stopped as soon as an anomaly was detected and we marked as optional all the features of the CaFM. We repeated every experiment 10 times. All the experiments were run in a Docker container[9] on a virtual machine having 2 vCPU and 8 GB of RAM. The Docker process was terminated if the entire execution was taking more than 6 GB of RAM or running for more than 5 minutes.

A second set of experiments were run by fixing the context number to 10 and varying the number of features in the set $\{200, 300, 350\}$ following the same execution modalities and limitation as the first set of experiments.

5.3 Results

The most interesting finding of our experiments is that no single approach dominates the other. Table 1 reports for instance the number of times an approach is the best (i.e., lower average running time) for the voidness analyses in the first set of experiments. It is possible to see that the Iterative approach is often the best, being the fastest in 358 cases. However, in 142 instances the Forall approach performs better (especially if the instance is void). This can be partially explained by a conjecture of the developers of Z3 that stated that "using quantifiers is a good option only if a very small percentage of the instances are needed to show that a problem is unsatisfiable".[10] It may be the case that the heuristics of the QBF solver led to the finding of the void context early on, compared to the iterative search approach in which the combinations of context are tried in a specific order.

Table 2: Number of times a feature anomaly approach is the best by varying the number of context variables (features number = 250).

Contexts	Result	Forall	Iterative	Pruning	Total
6	Dead	5	29		34
	False	7	21	6	34
	No Anomaly	9	17	6	32
8	Dead	4	26		30
	False	10	15	4	29
	No Anomaly	13	22	6	41
10	Dead	9	19		28
	False	11	10	2	23
	No Anomaly	20	23	6	49
12	Dead	4	22		26
	False	11	8	2	21
	No Anomaly	17	28	8	53
14	Dead	6	18		24
	False	6	4	5	15
	No Anomaly	16	33	12	61
Total		148	295	57	500

As far as the feature analysis is concerned, Table 2 shows the number of times every approach was the best. Also in this case, the Iterative approach is usually better being the fastest in 295 instances, but there is a non-negligible number of cases in which the Forall and Pruning approach are the best. The Forall approach seems to be competitive with the Iterative one for instances that have no dead features but false optional ones. Pruning is usually slower than the other two approaches and is not competitive in case there are dead features, probably because the Pruning approach requires to identify all the dead features to return one.

In the second set of experiments, as expected, we noticed that the more features the higher are the average running times. The approaches start to timeout for some instances already when 300 features are considered. In particular, for the feature check the Iterative approach timeouts for 12 instances (i.e., at least one in 10 repetitions took more than 300 seconds), the Forall timeouts for 1 instance, and the Pruning for 2. For the voidness check, the Forall timeouts in 5 instances with 300 features while the Iterative timeouts for the first time only for instances with 350 features.

To better visualize the improvement in solving times when multiple approaches can be used in parallel, we have plotted in Figure 2 the average times taken by the Iterative approach (black dotted line) and the times taken by the fastest among the Iterative and the Forall approaches (continuous red line) for the voidness analysis.

[8]The SAT generator is available online at https://github.com/RalfRothenberger/Power-Law-Random-SAT-Generator.

[9]The Docker image used for the tests is available at https://hub.docker.com/repository/docker/jacopomauro/hyvar-rec. The script used to perform the experiments is available at https://github.com/HyVar/hyvar-rec/blob/master/test/test_random.py.

[10] https://stackoverflow.com/questions/13268394/avoiding-quantifiers-in-z3

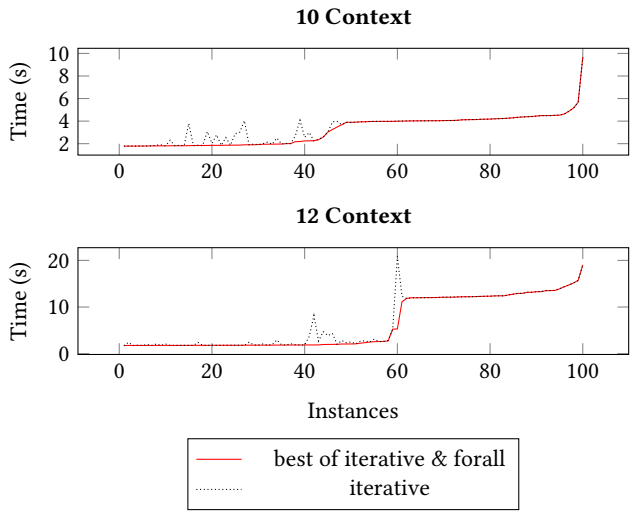

Figure 2: Solving times for voidness analysis with 250 features and varying number of context variables.

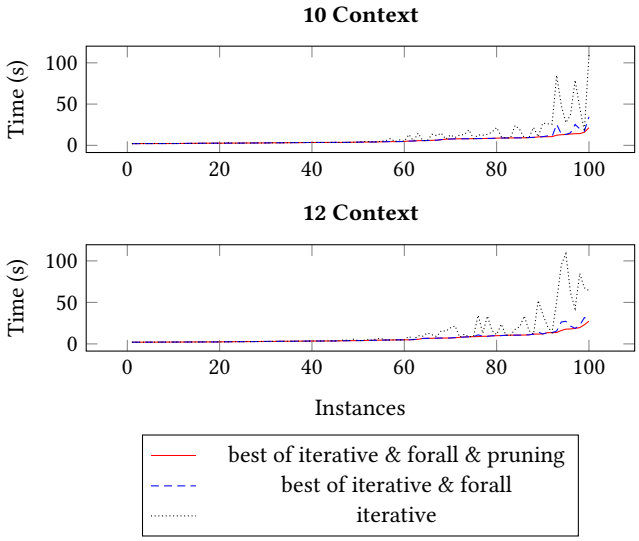

Figure 3: Solving times for feature anomaly analysis with 250 features and varying number of context variables.

For space reasons, the plots are related to the first set of experiments using instances having 250 features with 10 and 12 context variables. For every context, the instances have been sorted by the average solving time. Figure 3 instead presents the plots depicting the average solving time for the feature anomaly analysis, considering the Iterative approach (black dotted line), the best among the Iterative approach and the Forall approaches (blue dashed line), and the best among all the three approaches used (continuous red line).

As far as the voidness analysis is concerned, when more than 10 contexts are used it is possible to see that there is a jump in the average solving times. This is mainly due to the fact that detecting

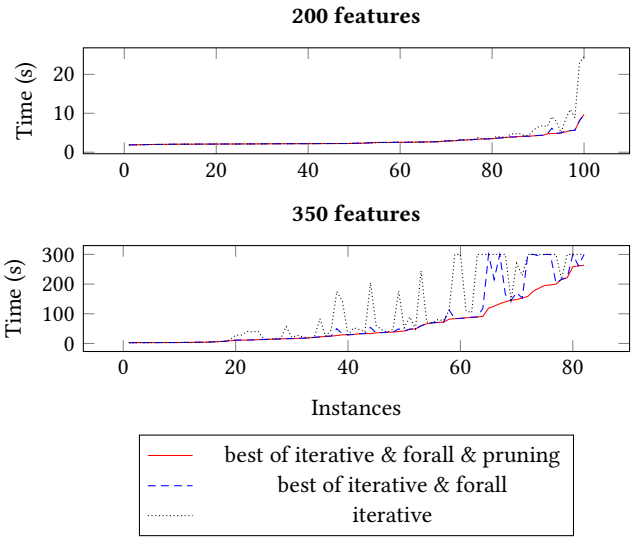

Figure 4: Solving times for feature anomaly analysis with 10 contexts variables and varying number of features.

an anomaly for void instances is often faster than proving that there are no anomalies. The majority of the instances before the jump in solving times are therefore void instances. Note, however, that there are exceptions. For example, considering the instances having 12 contexts, the most difficult 2 instances for the Iterative approach are void (and luckily in one of these two cases the Forall approach is better, reducing the solving time from 20 to 5 seconds as shown by the spike in the 12 context plot of Figure 2). Overall, it is possible to see that there are some instances in which the Forall approach improves the solving times.[11]

The performance difference in some instances is even more visible for the feature anomaly analysis. As shown in Figure 3, there are occurrences where the Iterative approach takes more than 100 seconds while the Forall or Pruning take less than half of the time.

For the second set of experiments, for space reasons, we present in Figure 4 the average solving times for feature anomaly analysis for instances having 10 contexts and 200 and 350 features. Since when using 350 features some instances timeout, for these plots we take into account only the instances that were solved by at least an approach within the timeout. In case of an approach times out for an instance, we consider its average time equal to 300 seconds.

For the feature anomaly analysis, it is possible to see that the Forall and Pruning approaches solve instances that would not have been solved otherwise within the timeout. The results on voidness analysis (here not visualized for space reasons) show instead that, similarly to what happened in the first set of experiments, for small instances the Forall approach can bring benefits for the detection of void instances (instance however also solved fast by the Iterative approach). On the contrary, for bigger instances, the Forall approach can bring benefits also for more difficult non-void instances.

[11] Please note that in average the Iterative approach is faster than the Forall approach. In case only one approach can be used, it is better to use the Iterative one.

We would like to conclude this section by also addressing the variability of the various approaches. We noticed that the Forall approach in general and the Iterative approach for the feature analyses have a big variability (i.e., standard deviation sometimes superior to 50%). For the feature anomaly analysis, the Forall approach also shows a significant performance variability. Surprisingly, for feature anomaly analysis, also the Iterative approach manifests a high variability. We believe that this is due to the fact that the Iterative approach prunes features depending on the solutions found by the SMT solver. The solutions found vary based on some random internal choices, thus probably causing the variability of the times of the Iterative approach. The Pruning approach instead has a lower variability. This is probably due to the fact that to find a single anomaly the Pruning approach has to account for either all the dead features or all the false optional features.

6 RELATED WORK AND CONCLUSIONS

The concept of enriching FMs with contextual information, originally proposed in [16], lead to the definition of Context-aware Feature Models [29] that combine contexts and features into a unique model. The idea that QBF solvers can be used to check the voidness was first presented in [28] without providing, however, any comparison w.r.t. the Iterative approach. In [33] HyVarRec has been extended to provide the explanation of voidness and perform the feature analysis, but only the Iterative approach was used. We are not aware of other tools that perform analysis of Context-aware FM that do not iterate over all the contexts.

One of the early approaches closest to ours is the UbiFex notation to model context-aware SPLs [11] that thanks to a simulator is used to check if the FM is void given a certain context. In particular, in that work they emphasize that it is hard to reason for each possible context regarding the FM being void.

Works on Dynamic SPL [10, 17, 31, 36, 41] adopt variants of CaFM where contexts are modeled with new feature models or constraints. These works, however, do not address the problem of anomaly detection but focus instead on the reconfiguration problem when a new context is given. They either iterate over any possible contexts or consider the reconfiguration over one specific context.

Other works which focus on modeling context-aware SPLs exist. For instance, as previously mentioned, in [16] context-awareness is captured by providing a second FM while in [32] ontologies are used instead to model the context. Unfortunately, these works just present these models without discussing their analysis.

In this paper, we describe how anomaly analysis can be performed for CaFMs before their deployment. In general, the analysis techniques relying on iterative calls to a SAT solver can still be used, but initial findings show that the usage of a QBF solver in parallel to the standard techniques can improve the overall performance. We also formalize a strategy for performing future analysis by letting the SAT solver guide the pruning of the features. Despite the fact that this technique requires finding first all the feature anomalies before providing one, sometimes it performs even better than other approaches that stop as soon as the first anomaly has been detected.

We have implemented the new analysis approaches in HyVarRec. We have evaluated them on randomly generated instances. Our goal was to try to compare these strategies on a uniform framework such as the one provided by the SMT solver Z3. Comparing the performance by using different SAT/QBF solvers along the lines of [37] is left as future work. We expect that the adoption of SAT solvers can improve the performance on certain instances, especially considering that often they are more effective in getting the model when a solution for a formula has been found. For example, when Z3 is used for large FMs in FeatureIDE [40] or HyVarRec, the time it takes to retrieve the model of a satisfiable formula can outweigh the cost of pushing and popping every single feature and check only the satisfiability. Clearly, using SAT solvers directly can therefore potentially speed up the solving time.

The results presented in this paper strongly depend on the random instances considered in the experiments. It may be that the structure and constraints of the real world instances are different from the random instances we generate. For this reason we consider these results only preliminary and we hope that new industrial benchmarks for CaFMs will be created to validate the performance of the current strategies on real instances. Moreover, compared to advanced tools designed for having in input FM diagrams such as [40], HyVarRec cannot perform advanced optimizations to prune features based on the feature diagram notation. It is left as future work to see whether advance pruning techniques based on the FM structure can be adopted to speed up the search when the CaFM feature diagram is available.

Due to the fact that all the problems considered (except product validity) are NP-hard [6] and that there is no approach (e.g., SAT, SMT) that dominates the others for solving such problems, we believe that the results obtained are not surprising. Often the performance of one solver may vary also by orders of magnitude depending on the instance to be solved or the random seed used. This can be exploited by *Algorithm Portfolios* [3, 15] that, based on the instances to solve, run different approaches to obtain an overall better solver. As done for the SAT and Constraint Programming fields [1, 2, 18, 19, 24, 43], further studies are needed to be able to devise strategies to select promising approaches based on the instances to solve.

Another recent direction to consider to possibly speed up the anomaly analysis of CaFM is the usage of Variational Satisfiability Solving [44] that automates the usage of incremental SAT solvers by accepting all related problems as a single variational input. Moreover, in case of critical systems requiring faster responses or Dynamic SPLs, it would be also interesting to study faster analysis techniques by using incomplete methods based on heuristics.

ACKNOWLEDGMENTS

The author would like to thank Michael Nieke for providing precious feedback on a preliminary version of this document.

REFERENCES

[1] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. 2015. A Multicore Tool for Constraint Solving. In *IJCAI*. AAAI Press, 232–238.
[2] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. 2015. SUNNY-CP: a sequential CP portfolio solver. In *SAC*. ACM, 1861–1867.
[3] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. 2015. Why CP Portfolio Solvers Are (under)Utilized? Issues and Challenges. In *LOPSTR (LNCS)*, Vol. 9527. Springer, 349–364.
[4] Don S. Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *SPLC (LNCS)*, Vol. 3714. Springer, 7–20.

[5] David Benavides, Sergio Segura, and Antonio Ruiz Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.* 35, 6 (2010), 615–636.

[6] Stephen A. Cook. 1971. The Complexity of Theorem-Proving Procedures. In *ACM Symposium on Theory of Computing*. ACM, 151–158.

[7] Krzysztof Czarnecki and Andrzej Wasowski. 2007. Feature Diagrams and Logics: There and Back Again. In *SPLC*. IEEE Computer Society, 23–34.

[8] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM* 54, 9 (2011), 69–77.

[9] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS)*, Vol. 4963. Springer, 337–340.

[10] Ismayle de Sousa Santos, Rossana Maria de Castro Andrade, Lincoln S. Rocha, Santiago Matalonga, Káthia Marçal de Oliveira, and Guilherme Horta Travassos. 2017. Test case design for context-aware applications: Are we there yet? *Inf. Softw. Technol.* 88 (2017), 1–16.

[11] Paula Fernandes and Cláudia Maria Lima Werner. 2008. UbiFEX: Modeling Context-Aware Software Product Lines. In *SPLC (Workshops)*. Lero Int. Science Centre, University of Limerick, Ireland, 3–8.

[12] Olivier Fourdrinoy, Éric Grégoire, Bertrand Mazure, and Lakhdar Sais. 2007. Eliminating Redundant Clauses in SAT Instances. In *CPAIOR (LNCS)*, Vol. 4510. Springer, 71–83.

[13] Tobias Friedrich, Anton Krohmer, Ralf Rothenberger, and Andrew M. Sutton. 2017. Phase Transitions for Scale-Free SAT Formulas. In *AAAI*. AAAI Press, 3893–3899.

[14] Enrico Giunchiglia, Massimo Narizzano, Luca Pulina, and Armando Tacchella. 2005. Quantified Boolean Formulas satisfiability library (QBFLIB). www.qbflib.org.

[15] Carla P. Gomes and Bart Selman. 2001. Algorithm portfolios. *Artif. Intell.* 126, 1-2 (2001), 43–62.

[16] Herman Hartmann and Tim Trew. 2008. Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains. In *SPLC*. IEEE Computer Society, 12–21.

[17] Jose Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2019. Context-aware energy-efficient applications for cyber-physical systems. *Ad Hoc Networks* 82 (2019), 15–30.

[18] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *LION (LNCS)*, Vol. 6683. Springer, 507–523.

[19] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. 2010. ISAC - Instance-Specific Algorithm Configuration. In *ECAI (Frontiers in Artificial Intelligence and Applications)*, Vol. 215. IOS Press, 751–756.

[20] 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-021. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231

[21] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is there a mismatch between real-world feature models and product-line research?. In *ESEC/FSE*. ACM, 291–302.

[22] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. 2016. Explaining anomalies in feature models. In *GPCE*. ACM, 132–143.

[23] Paolo Liberatore. 2002. The Complexity of Checking Redundancy of CNF Propositional Formulae. In *Proceedings of the 15th Eureopean Conference on Artificial Intelligence, ECAI'2002, Lyon, France, July 2002*. IOS Press, 262–266.

[24] Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. 2013. Algorithm Portfolios Based on Cost-Sensitive Hierarchical Clustering. In *IJCAI*. IJCAI/AAAI, 608–614.

[25] Mike Mannion. 2002. Using First-Order Logic for Product Line Model Validation. In *SPLC (LNCS)*, Vol. 2379. Springer, 176–187.

[26] Filip Maric. 2009. Formalization and Implementation of Modern SAT Solvers. *J. Autom. Reasoning* 43, 1 (2009), 81–119.

[27] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. 2016. Context Aware Reconfiguration in Software Product Lines. In *VaMoS*. ACM, 41–48.

[28] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. 2017. Anomaly Detection and Explanation in Context-Aware Software Product Lines. In *SPLC*. ACM, 18–21.

[29] Jacopo Mauro, Michael Nieke, Christoph Seidl, and Ingrid Chieh Yu. 2018. Context-aware reconfiguration in evolving software product lines. *Sci. Comput. Program.* 163 (2018), 139–159.

[30] Jens Meinicke, Thomas ThŘm, Reimar Schr, Fabian Benduhn, Thomas Leich, Gunter Saake, et al. 2017. Quality Assurance for Feature Models and Configurations. In *Mastering Software Variability with FeatureIDE*. Springer, 81–94.

[31] Kim Mens, Rafael Capilla, Herman Hartmann, and Thomas Kropf. 2017. Modeling and Managing Context-Aware Systems' Variability. *IEEE Softw.* 34, 6 (2017), 58–63.

[32] Sinisa Neskovic and Rade Matic. 2015. Context modeling based on feature models expressed as views on ontologies via mappings. *Comput. Sci. Inf. Syst.* 3 (2015), 961–977.

[33] Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzke. 2018. Anomaly analyses for feature-model evolution. In *GPCE*. ACM, 188–201.

[34] Christos H. Papadimitriou. 1994. *Computational complexity*. Addison-Wesley.

[35] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc.

[36] Karsten Saller, Malte Lochau, and Ingo Reimund. 2013. Context-aware DSPLs: model-based runtime adaptation for resource-constrained systems. In *SPLC*. ACM, 106–113.

[37] Joshua Sprey, Chico Sundermann, Sebastian Krieter, Michael Nieke, Jacopo Mauro, Thomas Thüm, and Ina Schaefer. 2020. SMT-based variability analyses in FeatureIDE. In *VaMoS*. ACM, 6:1–6:9.

[38] Jing Sun, Hongyu Zhang, Yuan-Fang Li, and Hai H. Wang. 2005. Formal Semantics and Verification for Feature Modeling. In *ICECCS*. IEEE Computer Society, 303–312.

[39] Maurice H. ter Beek, Ferruccio Damiani, Michael Lienhardt, Franco Mazzanti, and Luca Paolini. 2019. Static analysis of featured transition systems. In *SPLC*. ACM, 9:1–9:13.

[40] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Sci. Comput. Program.* 79 (2014), 70–85.

[41] Markus Weckesser, Roland Kluge, Martin Pfannemüller, Michael Matthé, Andy Schürr, and Christian Becker. 2018. Optimal reconfiguration of dynamic software product lines based on performance-influence models. In *SPLC*. ACM, 98–109.

[42] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. 2010. Efficiently solving quantified bit-vector formulas. In *FMCAD*. IEEE Computer Society, 239–246.

[43] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2008. SATzilla: portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.* 32 (2008), 565–606.

[44] Jeffrey M. Young, Eric Walkingshaw, and Thomas Thüm. 2020. Variational satisfiability solving. In *SPLC*. ACM, 18:1–18:12.