Achieving change requirements of feature models by an evolutionary approach

Paolo Arcaini

National Institute of Informatics, Japan

Angelo Gargantini

University of Bergamo, Italy

Marco Radavelli

University of Bergamo, Italy

Abstract

Feature models are a widely used modeling notation for variability and commonality management in software product line (SPL) engineering. In order to keep an SPL and its feature model aligned, feature models must be changed by including/excluding new features and products, either because faults in the model are found or to reflect the normal evolution of the SPL. The modification of the feature model to be made to satisfy these change requirements can be complex and error-prone. In this paper, we present a method that is able to automatically update a feature model in order to satisfy a given update request. The method is based on an evolutionary algorithm that iteratively applies structure-preserving mutations to the original model, until the model is completely updated or some other termination condition occurs. Among all the possible models achieving the update request, the method privileges those structurally simpler. We evaluate the approach on real-world feature models; although it does not guarantee to completely update all the possible feature models, empirical analysis shows that, on average, around 89% of requested changes are applied.

Keywords: Software Product Line, feature model, update request, evolutionary approach, mutation

1. Introduction

Software Product Lines (SPLs) are families of products that share some common characteristics, and differ on some others [1, 2]. Software product line engineering consists in the development and maintenance of SPLs by taking into account their commonalities and differences. The variability of SPLs is usually already described at design time by using *variability models* [3]; one of the main used variability models are *feature models* (FMs). A feature model lists the *features* in an SPL together with their possible *constraints*. In this way, it can represent, in a compact and easily manageable way, millions of variants, each describing a possible product. The availability of a feature model enables several analysis activities on the SPL, like verification of consistency, automatic product configuration, interaction testing among features on the products, and similar actions [4].

Overtime, feature models need to be updated in order to avoid the risk of having a model with wrong features and/or wrong constraints. Two main causes for *change requirements* can be identified: either the model is incorrect (it excludes/includes some products that should be included/excluded), or the SPL has changed. The change requirements can come from different sources: *failing tests* identifying configurations evaluated not correctly, or *business requirements* to add new products, to allow new features, to not support some products any longer, and so on. All these change requirements identify configurations/features to add or remove, but do not identify a way to modify the feature model to achieve them (differently from other approaches [5]). Manually updating a feature model to achieve all the change requirements can be particularly difficult and, in any case, error-prone and time consuming; moreover, also an analytical approach to apply the required changes is difficult to devise.

For these reasons, in this paper we investigate an approach to automatically update a feature model upon change requirements. The user must only specify an *update request*, based on the change requirements coming from testing or from business requirements. The update request is composed of three kinds of *feature-based* change requirements and two kinds of *product-based* ones; the feature-based ones consist in features that must be renamed, and features that must be added to and removed from the products of the original feature model; the product-based ones, instead, consist in configurations that should be no more accepted as products by the final model, and configurations that should instead be accepted as new products (both removed and added configurations can be defined in a symbolic

[1]P. Arcaini is supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST. Funding Reference number: 10.13039/501100009024 ERATO.

way). Starting from the update request, the approach tries to apply the feature-based change requirements directly on the starting model; however, some of these requirements could be not completely fulfilled. Then, by means of an evolutionary algorithm, the approach tries to obtain a feature model that captures all the change requirements: the feature-based ones not fulfilled in the previous phase, and the product-based ones. The process iteratively generates, from the current population of candidate solutions, a new population of feature models by mutation. All the members of a population are evaluated considering primarily the percentage of correctly evaluated configurations, and secondly a measure of the *structural complexity* of the model, defined in terms of number of cross-tree constraints. When a correct model is found or some other termination condition holds, the process terminates returning as final model the one having the highest fitness value.

We originally proposed the approach and did some preliminary experiments in [6]. This paper extends the work in [6] by (a) defining a more expressive way to specify the update request (allowing feature renaming and the specification of sets of products to be added/removed in a symbolic way), (b) introducing a new version of the fitness function that also considers the *structural complexity* of the model, (c) introducing new mutation operators, (d) checking, during the evolutionary process, that a mutant does not contain any anomaly, and (e) performing a more extensive evaluation using real evolutions of feature models.

The paper is structured as follows. Sect. 2 provides some basic definitions on feature modeling. Sect. 3 introduces the definitions of update request and target, and Sect. 4 describes the process we propose to modify the starting feature model in order to achieve the specified update request. Then, Sect. 5 presents the experiments we performed to evaluate the approach, and Sect. 6 discusses possible threats to its validity. Finally, Sect. 7 reviews some related work, and Sect. 8 concludes the paper.

2. Basic definitions

In software product line engineering, feature models are a special type of information model representing all possible products of an SPL in terms of features and relations among them. Specifically, a feature model *fm* is a hierarchically arranged set of features F, where each parent-child relation between them is a constraint of one of the following types[2]:

- *Or*: at least one of the sub-features must be selected if the parent is selected.
- *Alternative* (xor): exactly one of the children must be selected whenever the parent feature is selected.
- *And*: if the relation between a feature and its children is neither an *Or* nor an *Alternative*, it is called *and*. Each child of an *and* must be either:
  - *Mandatory*: the child feature is selected whenever its respective parent feature is selected.
  - *Optional*: the child feature may or may not be selected if its parent feature is selected.

Only one feature in F has no parent and it is the *root* of *fm*.

In addition to the parental relations, it is possible to add *cross-tree constraints*, i.e., relations that cross-cut hierarchy dependencies. *Simple* cross-tree constraints are:

- A *requires* B: the selection of feature A in a product implies the selection of feature B. We also indicate it as $A \rightarrow B$.
- A *excludes* B: A and B cannot be part of the same product. We also indicate it as $A \rightarrow \neg B$.

Some frameworks for feature models also support *complex* cross-tree constraints [9] through *general* propositional formulas. In our approach, we allow feature models to contain complex cross-tree constraints.

Feature models can be visually represented by means of feature diagrams, and their semantics can be expressed by using propositional logic [2, 4]: features are represented by propositional variables, and relations among features by propositional formulae. We identify with $\textsc{bof}(fm)$ the BOolean Formula representing a feature model *fm*.

Definition 1 (**Configuration**). *A* configuration *c of a feature model fm is a subset of the features F of fm (i.e., $c \subseteq F$).*

If *fm* has n features, there are 2^n possible configurations.

Definition 2 (**Validity**). *Given a feature model fm, a configuration c is* valid *if it contains the* root *and respects the constraints of fm. A valid configuration is called* product*.*

For our purposes, we exploit the propositional representation of feature models for giving an alternative definition of configuration as a set of n literals $c = \{l_1, \ldots, l_n\}$ (with $n = |F|$), where a positive literal $l_i = f_i$ means that feature f_i belongs to the configuration, while a negative literal $l_i = \neg f_i$ means that f_i does not belong to the configuration. We will also represent a configuration as a BOolean Formula as follows: $\textsc{bof}(c) = \bigwedge_{i=1}^{n} l_i$.

Furthermore, since in the proposed approach we need to evaluate a feature model over a possibly wider set of features U, we introduce $\textsc{bof}(fm, U) = \textsc{bof}(fm) \wedge \bigwedge_{f \in U \setminus F} \neg f$, where *fm* explicitly refuses all the configurations containing a feature not belonging to its set of features F; such technique has been already employed by different approaches that need to compare feature models defined over different sets of features [10, 11].

Example 1. Let's consider the feature model shown in Fig. 1. It is the third version of the CAR SPL described in [5]. It is composed of eight features F = {`CarBody`, `MultimediaDevices`, `OtherFeatures`, `Radio`, `Navigation`, `MonochromeRadioDisplay`, `MonochromeNavigationDisplay`, `ColorNavigationDisplay`}. `CarBody` is the root feature; its children `MultimediaDevices` and `OtherFeatures` are respectively optional and mandatory. `MultimediaDevices` has two optional children: `Radio` that is the father of the mandatory feature `MonochromeRadioDisplay`, and `Navigation` that is the father of

[2] As done by FeatureIDE [7], we assume that each feature can be the father of only one group (either *Or*, *Alternative*, or *And*). This is not a limitation, as having different groups as children can be obtained by using abstract features [8].

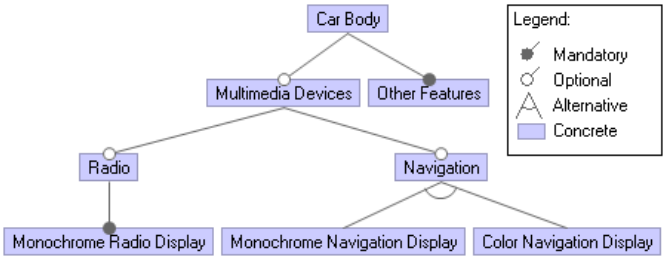

Figure 1: Example of feature model (taken from [5])

the alternative group between `MonochromeNavigationDisplay` and `ColorNavigationDisplay`.

3. Specifying an update request

We suppose that the product line engineer wants to update an existing feature model fm_i (*initial feature model*); although (s)he knows which are the desired updates in terms of products and features to add or remove, (s)he does not know how to write a feature model fm' that satisfies all these updates.

In this section, we describe how the user can specify her/his *change requirements*. By analysing existing evolutions of feature models described in literature [5, 12, 13, 14, 15], we identified the following five types of change requirements:

- a feature must be identified with a new name. Although this change requirement is straightforward to achieve, it is widely used (e.g., for the SmartHome SPL [13] considered in the Ample Project, we have observed 12 feature renames) and it is important to keep track of it, otherwise someone reading the updated feature model may have the impression that one feature has been added and one removed (in particular, if also other changes have been done on the feature model and, therefore, spotting the renamed feature is not easy).
- a feature must be added to the feature model. This change requirement occurs when a new feature must now be supported by the SPL. For example, in the CAR SPL [5], the support for *DVD entertainment* has been introduced in 2012 (documented in the fourth version of the SPL feature model).
- a feature must be removed from the feature model. This change requirement occurs when a feature is no more supported by the SPL.[3] For example, in the CAR SPL [5], feature *Color Radio Display* has been removed in 2011 (third version of the SPL feature model).
- some products must now be accepted by the SPL. This change requirement occurs when some constraints existing on the SPL (due to technical reasons or regulations) do not hold anymore and so some configurations that were forbidden in the past can now be accepted. For example, in the second version of the Pick-&-Place (PPU) Unit SPL [12], the system is able to process either *plastic* or *metal* workpieces; in the third version of the SPL, instead, a technical improvement has allowed to handle plastic and metal workpieces in combination.
- some products cannot be accepted anymore by the SPL. This change requirement happens when some constraints over the SPL emerge (e.g., due to new regulations). For example, in the HelpSystem SPL [14], the second version of the SPL does not allow anymore that the sensor only detects either *pressure* or *not pressure*, i.e., the products with only *pressure* or *not pressure* are not allowed and have been removed.

In the following, we provide the formal definition of update request. We assume that the initial feature model does not contain any dead feature or redundant constraint [4, 16]; in case it has any, we can remove them using standard techniques, as, for example, that provided by FeatureIDE [7].

Definition 3 (Update request). *Given an initial feature model fm_i defined over a set of features F, we call* update request *UR the modifications a user wants to apply to fm_i in order to obtain the desired updated model fm'. An update request is composed of five change requirements, three regarding the features of the feature model, and two the configurations/products. The first* feature-based *change requirement regards the features names:*

- **Rename features***: $F_{TBR} = \{f_1, \ldots, f_m\}$ is a subset of features of F that must be renamed and ren a renaming function; F_R is the set obtained by replacing every feature $f_i \in F_{TBR}$ with $ren(f_i)$, i.e., $F_R = (F \setminus F_{TBR}) \cup \bigcup_{f_i \in F_{TBR}} \{ren(f_i)\}$. We identify with fm_{ren} the feature model obtained by renaming the features according to F_{TBR} and ren.*[4]

The other two feature-based *change requirements over the features of fm_{ren} are:*

- **Add features***: F_{add} is a set of features the user wants to add to fm_{ren}. For each e in F_{add}, the user has to define in $F_R \cup F_{add}$ the parent of e, denoted by $parent(e)$. By adding a feature e with parent p, we assume that the user wants to duplicate all the products that contain p by adding also e.*[5]
- **Remove features***: F_{rem} is a subset of the features of F_R to remove; by removing a feature f, we assume that the user wants to remove f from all the existing products and prohibit its selection in new products.*

We identify with F' the new set of features that must be used in the updated feature model fm', namely $(F_R \cup F_{add}) \setminus F_{rem}$.

Two product-based *change requirements, instead, are related to the products/configurations of the feature model (over the new set of features F'):*

[3] Note that this change requirement could be achieved by removing all the products containing the feature (see last change requirement), but not removing the feature from the feature model. However, in this way, the feature would become *dead* [4], making the feature model less readable.

[4] Note that feature renaming is already supported by feature model editors, such as FeatureIDE. We keep it among our change requirements for completeness with respect to the feature model evolutions observed in literature.

[5] Note that this corresponds to have e as optional feature of p. However, this could not be always achievable, as explained in Sect. 4.2.1 (unless abstract features [8] are used); therefore, we define the change requirement in this more general form, in order to keep the semantics of change requirement and the way to achieve it clearly distinguished.

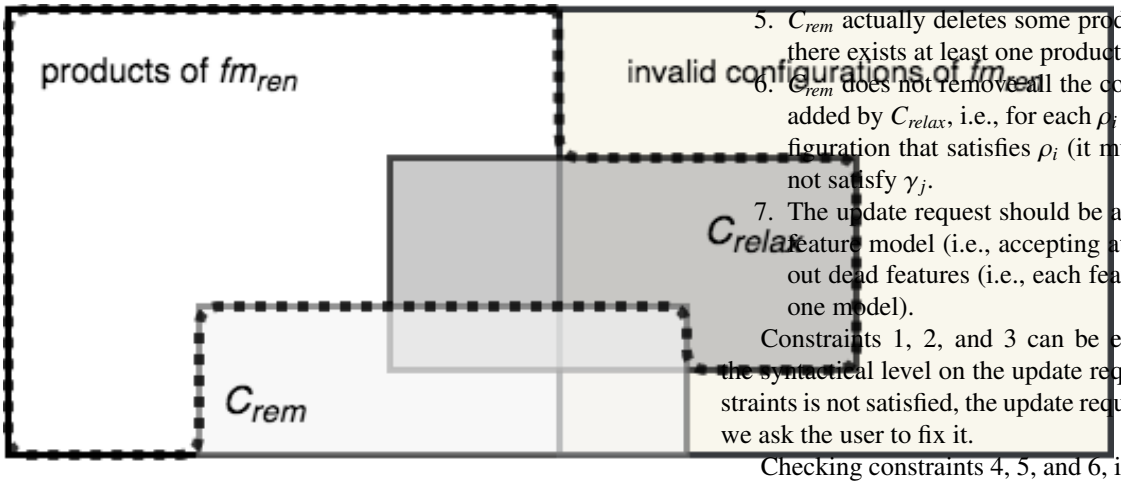

Figure 2: Added and removed products

- **Add products**: *C_{relax} is a set of predicates $\{\rho_1, \ldots, \rho_n\}$ over F' that denote conditions that must be allowed in fm'. The logical models satisfying ρ_i are products that must be added to the feature model, provided that they do not violate feature model constraints defined over features not included in ρ_i.*
- **Remove products**: *C_{rem} is a set of predicates $\{\gamma_1 \ldots, \gamma_m\}$ over F' that denote a set of products to be removed. Each product (i.e., logical model) satisfying a γ_i must be removed from the valid product set.*

The meaning of C_{relax} and C_{rem} is to modify the set of valid products, as depicted in Fig. 2. The new product set (in dotted line) is enriched by configurations identified by C_{relax} but deprived of the products identified by C_{rem}; note that C_{rem} could also remove some products added by C_{relax}.

If the user wants to include/exclude a specific configuration c, then (s)he simply adds $\text{BOF}(c)$ to C_{relax}/C_{rem}.[6]

3.1. Well-formedness of an update request

We impose some constraints on the update request to be sure that the different change requirements are not contradictory and useful (i.e., they actually affect the product set):

1. Each added feature in F_{add} must have an ancestor in $F_R \setminus F_{rem}$. This implies that it is not possible to remove by F_{rem} the parents p of features added in F_{add}, i.e., $F_{rem} \cap (\bigcup_{f \in F_{add}} \{parent(f)\}) = \emptyset$.
2. It is not possible to remove features that have been renamed, i.e., $F_{rem} \cap (\bigcup_{f_i \in F_{TBR}} \{ren(f_i)\}) = \emptyset$.
3. Predicates in C_{relax} cannot predicate over features that have been removed in F_{rem}.
4. C_{relax} actually increases the set of products: for each ρ_i in C_{relax}, there exists at least one non-valid configuration that satisfies ρ_i.
5. C_{rem} actually deletes some products: for each γ_i in C_{rem}, there exists at least one product that satisfies γ_i.
6. C_{rem} does not remove all the configurations that must be added by C_{relax}, i.e., for each ρ_i and γ_j there exists a configuration that satisfies ρ_i (it must be added) and it does not satisfy γ_j.
7. The update request should be achievable by a consistent feature model (i.e., accepting at least one product) without dead features (i.e., each feature is selected in at least one model).

Constraints 1, 2, and 3 can be easily checked directly at the syntactical level on the update request. If one of these constraints is not satisfied, the update request cannot be applied and we ask the user to fix it.

Checking constraints 4, 5, and 6, instead, requires to reason over the propositional representation of fm_{ren} (i.e., $\text{BOF}(fm_{ren})$) and the predicates in C_{relax} and C_{rem}; for example, checking constraint 4 consists in verifying that, for each $\rho_i \in C_{relax}$, $\neg\text{BOF}(fm_{ren}) \wedge \rho_i$ is satisfiable. If one of these constraints is not satisfied, the update request is still consistent, although the corresponding change requirement is useless; in case of constraint violation, we warn the user about the useless change requirement and, if (s)he confirms that the change requirement is indeed not necessary, we continue the updating process without that requirement.

Checking constraint 7, instead, requires to reason about the interaction of the different change requirements and can only be done after we define the target, as explained in the next section.

Example 2. Given the CAR SPL model shown in Fig. 1 and described in Ex. 1, an update request could be the following[7]:

- **Rename features** $F_{TBR} = \{\texttt{MonochromeRadioDisplay}\}$ and *ren* $(\texttt{MonochromeRadioDisplay}) = \texttt{RadioDisplay}$.
- **Add features** $F_{add} = \{\texttt{DVDEntertainment}\}$ and *parent*(DVDEnter = MultimediaDevices.
- **Remove features** $F_{rem} = \{\texttt{OtherFeatures}\}$.
- **Add products** $C_{relax} = \{\texttt{MultimediaDevices} \wedge \texttt{Navigation} \wedge (\texttt{MonochromeNavigationDisplay} \leftrightarrow \texttt{ColorNavigationDisplay})\}$. We also want to allow products with support for navigation, and having both displays or having no display at all.
- **Remove products** $C_{rem} = \{\texttt{Navigation} \wedge \neg\texttt{Radio}\}$. We want to exclude that navigation is present when the radio is not.

3.2. Update request semantics

We here precisely define the semantics of an update request *UR*. In order to do this, we define a formula that accepts and rejects configurations as the updated feature model should do.

[6] This is what we actually do in the experiments in Sect. 5 where update requests are computed as differences between different versions of feature models, and C_{relax} and C_{rem} can only be identified in terms of sets of configurations.

[7] Note that the change requirement F_{add} is the same reported in [5] for the evolution from the third version (shown in Fig. 1) to the fourth version of the CAR SPL; C_{rem}, instead, is taken from the evolution from the first to the second version of the same SPL. In order to have a complete example, the other three change requirements have been identified by us; F_{TBR} and F_{rem} resemble similar change requirements observed in literature, respectively in the SmartHome SPL [13] and in the CAR SPL [5].

In order to define the semantics of F_{rem} and C_{relax}, we need to be able to represent the feature model without some features. We exploit the approach used in [8] to remove features from feature models. To eliminate a feature f from a propositional formula, we substitute f by its possible values (true or false).

Definition 4 (Features removal)**.** *Given a feature model fm and a set of features $K = \{f_1, \ldots, f_k\}$ to remove, we recursively define filter(fm, K) as follows:*

$$filter(fm, K) = \begin{cases} \text{BOF}(fm) & \text{if } K = \emptyset \\ filter(fm, K')[f \leftarrow \top] \vee filter(fm, K')[f \leftarrow \bot] & \text{if } K = K' \cup \{f\} \end{cases}$$

The formula $filter(fm, K)$ has as logical models the same models as BOF(fm) except that all the features in K have been removed.

Exploiting Def. 4, the semantics of F_{rem} is captured by the formula

$$\phi_{rem} = filter(fm_{ren}, F_{rem})$$

whose logical models (i.e., products) are those of fm_{ren} without the removed features.

In order to capture the semantics of a $\rho_i \in C_{relax}$, we need to characterize the set of configurations to add. These are those that satisfy ρ_i but still respect the constraints of ϕ_{rem}, except for those involving the features of ρ_i. This is captured by

$$\phi^i_{relax} = filter(\phi_{rem}, features(\rho_i)) \wedge \rho_i$$

being *features* a function returning the features (i.e., propositional variables) contained in a formula. The *filter* on ϕ_{rem} has the effect of making the features in ρ_i unconstrained, i.e., it keeps only the constraints of ϕ_{rem} that do not *interfere* with ρ_i.

We can now build the formula that defines the semantics of the whole update request. We name the formula as *target* as it will be used as oracle to guide the proposed updating process (see Sect. 4).

Definition 5 (**Target**)**.** *The target t is a propositional formula whose models exactly correspond to the products of the desired updated feature model. Assume an update request $UR = \{ren, parent, F_{rem}, C_{relax}, C_{rem}\}$ defined over a feature model fm, with the functions ren defined over F_{TBR} and parent defined over F_{add}. Let fm_{ren} be the feature model renamed according to ren; the target is defined as:*

$$t = (\overbrace{\phi_{rem}}^{\text{remove } F_{rem} \text{ from products}} \vee \overbrace{\bigvee_{\rho_i \in C_{relax}} \phi^i_{relax}}^{\text{add products}}) \wedge \overbrace{\bigwedge_{f \in F_{add}} f \rightarrow parent(f)}^{\text{add } F_{add} \text{ features only when possible}} \wedge \overbrace{\bigwedge_{f \in F_{rem}} \neg f}^{\text{disable } F_{rem} \text{ features}} \wedge \overbrace{\bigwedge_{p \in C_{rem}} \neg p}^{\text{remove products}}$$

The target correctly rejects all the configurations of C_{rem} and those containing a removed feature in F_{rem}; the accepted configurations are those in C_{relax}, plus those of ϕ_{rem} (i.e., the original feature model without the removed features) possibly extended with each added feature f of F_{add} only when $parent(f)$ is present.

Note that the target correctly predicates over all the features $F_U = F' \cup F_{rem}$: those of the original feature model (after renaming), those added, and those removed.

Checking the target. On the target, we can finally check constraint 7 described in Sect. 3.1, requiring that the update request does not produce anomalies.

First of all, we check that the target accepts at least one product (i.e., it is satisfiable); if not, we warn the user that the update request cannot be applied.

Moreover, we also check that each feature $f \in F'$ can be actually selected in at least one product; it this is not possible, it means that f is required to be dead in the final model. If this is the case, we warn the user about this and, if this is the intended behavior, we move f in F_{rem}, so that we can directly try to remove it during the repairing process.

4. Evolutionary updating process

Modifying the initial feature model fm_i such that it satisfies the update request as specified by the target is a challenging task. Note that, in general, there could be no fm' that exactly adds and removes the specified configurations and features, unless *complex* cross-tree constraints are used. However, we claim that the usage of these constraints should be discouraged. Urli et al. [17] observe that "they make FMs complex to understand and maintain", Reinhartz-Berger et al. [18] experimentally assessed that they are more difficult to understand than parent-child relationships (at least, by modelers who are unfamiliar with feature modeling), and Berger et al. [19] report that "they are known to critically influence reasoners". Also the authors of FeatureIDE noted that cross-tree constraints "are harder to comprehend than simple tree constraints" and that "relations among features should be rather expressed using the tree structure if possible" [7]. In this paper, we therefore avoid the addition of complex cross-tree constraints, as we not only aim at correctness (i.e., full achievement of the update request), but also at readability of the final model. Some approaches that aim at simplifying complex constraints exist [9, 20], but they may diminish readability and other qualities, such as compactness, traceability, and maintainability.[8]

This paper proposes a heuristic approach that tries to achieve the update request *as much as possible*. In order to do this, we use the target as *oracle* to compute the *fault ratio*. The fault ratio tells how close we are to the correct solution. In Sect. 4.1, we give the definition of fault ratio and then, in Sect. 4.2, we describe the updating process we propose.

4.1. Correctness

We can use the target to evaluate whether a feature model fm' captures the desired change requirements, i.e., fm' is equivalent to the target ($\models t \leftrightarrow \text{BOF}(fm', F_U)$). In the following, we will only compare feature models fm' whose features $F_{fm'}$ are, at most, those in F_U, i.e., $F_{fm'} \subseteq F_U$.

Although a feature model could not fulfill all the change requirements, it could satisfy them partially. We give a measure

[8] In Sect. 8, we discuss how complex cross-tree constraints could be used to achieve all the change requirements and the issues that could be related to that approach.

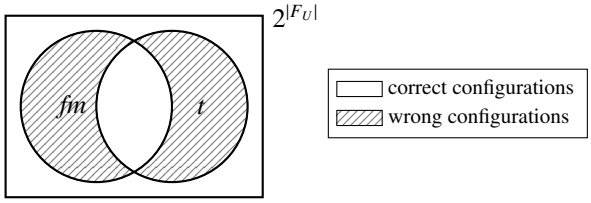

Figure 3: Faults

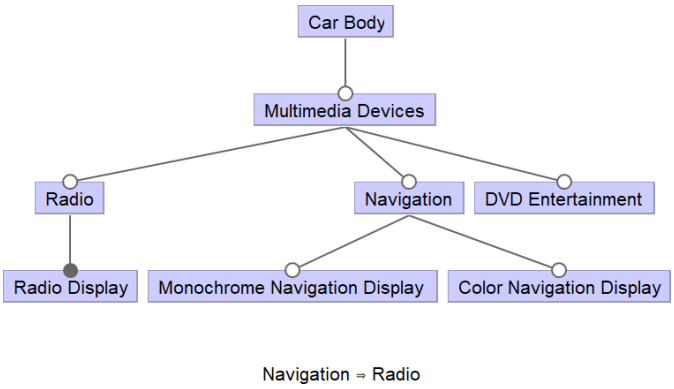

Figure 4: Updated feature model

of the *difference* between a feature model *fm* (either the initial one fm_i or a modified one fm') and the target as follows.

Definition 6 (Fault ratio). *Given a feature model fm and a target t, the* fault ratio *of fm w.r.t. t is defined as follows:*

$$FR(fm, t) = \frac{|AllConfs(\text{BOF}(fm, F_U) \neq t, F_U)|}{2^{|F_U|}}$$

where AllConfs(φ, V) *returns all the logical models of formula* φ*, i.e., all the truth assignments m to propositional variables V, such that* $m \models \varphi$.[9]

If the fault ratio is equal to 0, it means that *fm* accepts as products the same configurations that are logical models of *t*; otherwise, there are some configurations that are wrongly evaluated by *fm*, as shown in Fig. 3: *fm* could wrongly accept some configurations and/or wrongly refuse some others.

Example 3. Fig. 4 shows a possible updated model that exactly satisfies all the change requirements reported in Ex. 2 for the model in Fig. 1. Note that the fault ratio of the initial renamed feature model fm_i w.r.t. to the target *t* (that corresponds to the final feature model) is $\frac{20}{2^9}$, as fm_i wrongly accepts all its 7 products and wrongly rejects all the 13 products of the target.

4.2. Updating process

The process we propose to update an initial feature model fm_i, given an update request *UR*, is depicted in Fig. 5. As initial step, we generate the target *t* as a Binary Decision Diagram (BDD), as described in Def. 5. Then, we start the updating process, that is composed of two consecutive macro-phases. We first try to deal with the feature-based change requirements (see Sect. 4.2.1) and then with the product-based ones (see Sect. 4.2.2).

4.2.1. Dealing with feature-based change requirements

First of all, we apply F_{TBR} to rename features, obtaining the feature model fm_{ren}.

Then, we modify fm_{ren} in order to try to achieve the change requirements of F_{add} and F_{rem}. For each $f \in F_{add}$, we add f as child of $parent(f)$ as follows: if $parent(f)$ is the father of an *Or* or an *Alternative* group, f is added to the group; in all the other cases, it is added as *Optional* child of $parent(f)$. We name as fm_A the feature model obtained after this step. Then, for each feature $f \in F_{rem}$, f is removed from fm_A and replaced by its children Ch_f (if any). The relation of the moved children Ch_f of f with their new parent p is set according to the way FeatureIDE removes features [7]:

1. If f was the only child of p, p takes the group type of f.
2. If p has group type *And*, (a) if the children Ch_f are in *And* relationship, they keep their type (either *Mandatory* or *Optional*) (b) otherwise (they are in *Alternative* or *Or*), they are set to *Optional*.
3. Otherwise, if p has group type *Alternative* or *Or*, features in Ch_f are simply added to the group (regardless of their type).

We name as fm_{AR} the feature model obtained after this step.

Note that the model fm_{AR} could still be not equivalent to the target, i.e., $\not\models t = \text{BOF}(fm_{AR}, F_U)$. This could be due to two reasons. First of all, the update request could also require to add as products configurations described by constraints in C_{relax} and/or remove configurations described by constraints in C_{rem}. Moreover, the two previous transformations do not guarantee to precisely implement the required change requirements F_{add} and F_{rem}, and they could introduce some wrong configurations (either wrongly accepted or rejected). For example, in order to implement the addition of a feature f with $parent(f) = p$ and p father of an alternative group, we add f to the group; however, this is not the exact semantics of F_{add} that requires to duplicate the products containing p and adding f to them.

4.2.2. Dealing with product-based change requirements

Starting from fm_{AR}, we apply an evolutionary updating approach to try to obtain an updated feature model equivalent to the target. The process is an instance of classical evolutionary algorithms [21]; an evolutionary algorithm can be understood (in a metaphor-free language [22]) as an optimization problem in which different solutions are modified by random changes and their quality is checked by an objective function. Precisely, the steps are as follows:

1. **Initial population**: at the beginning, a population P is created. P is a set of candidate solutions.
2. **Iteration**: the following steps are repeatedly executed:

[9] In our approach, we represent formulas as Binary Decision Diagrams (BDDs) in JavaBDD that implements $|AllConfs|$ by means of the method `satCount` that directly computes the cardinality of the set without enumerating all the models.

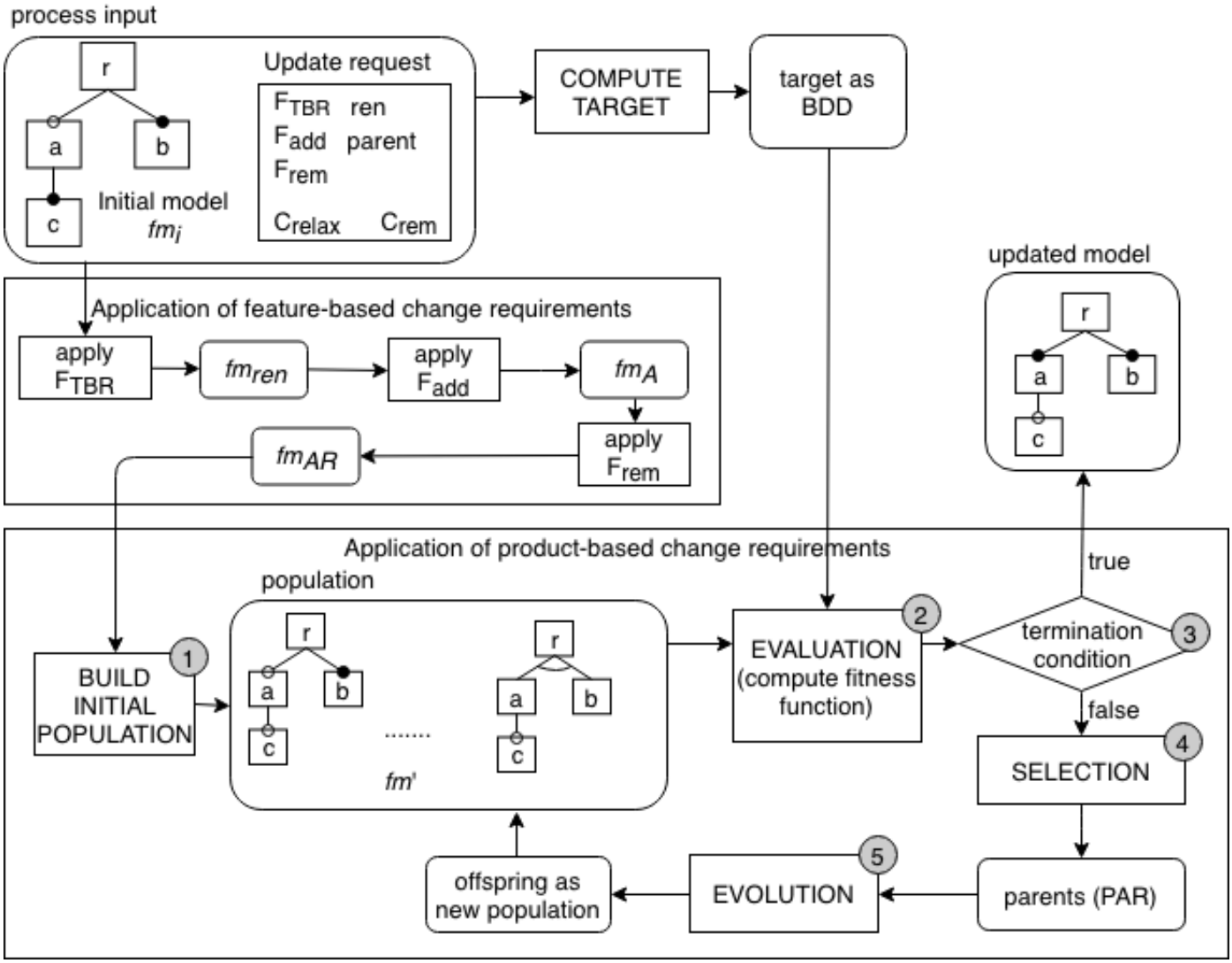

Figure 5: Proposed evolutionary approach

(a) **Evaluation**: each member of the population P is evaluated using a given *fitness function*, representing the objective function.

(b) **Termination**: a termination condition is checked in order to decide whether to start the next iteration. If the termination condition holds, the candidate with the best *fitness value* is returned as final model.

(c) **Selection** (*Survival of the Fittest*): some members of P having the best values of the fitness function are selected as parents of the next generation and collected in the set *PAR*.

(d) **Evolution**: parents *PAR* are mutated to obtain the *offspring* to be used as population in the next iteration. The mutation performs random changes suitable to improve the existing solutions.

In our approach, we assume that the population P is a multiset (i.e., possibly containing duplicated elements) with fixed size M equal to $H \cdot |F'|$, where H is a parameter of the process. In the following, we describe each step in details.

Initial population. As initial population, we generate the set P by cloning fm_{AR} M times (step 1 in Fig. 5). In this way, if fm_{AR} is already correct, it will be returned as final model in the *termination condition* phase.

Evaluation. As first step of each iteration (step 2 in Fig. 5), each candidate member fm' of the population P is evaluated using a *fitness function* that tells how *good* the member is in achieving the overall goal. We define the fitness function both in terms of fault ratio (see Def. 6) and of *complexity* of the model structure. Indeed, we would like to avoid that, during the updating process, the feature model becomes unreadable, unnecessarily complex, and difficult to maintain [19, 17, 18, 7]. We have decided to consider, at least initially, the number of cross-tree constraints as indicator of complexity, since the constraints among features should be expressed by structural relationships and cross-tree constraints should be used only when really necessary. We introduce the following fitness function:

$$fitness_t(fm') = 1 - FR(fm', t) - k \times ctc(fm') \quad (1)$$

where *ctc* is a function returning the number of cross-tree constraints of a feature model and k a constant. In our approach, the quality of a candidate must be mainly given by the percentage of configurations that it evaluates correctly, i.e., $1 - FR(fm', t)$; if a candidate c_1 evaluates correctly more configurations than a candidate c_2, its fitness should be guaranteed to be greater than that of c_2. However, for models having the same fault ratio, the fitness should penalize those that are structurally more complex. In order to obtain this effect, we use as k:

$$k = \frac{1}{2^{|F_U|} \times 2|F_U|^2} \quad (2)$$

Note that $2|F_U|^2$ is a safe strict upper bound on the number of cross-tree constraints among the features of the feature model. Indeed, among the possible $2|F_U|^2$ cross-tree constraints, some of them are not introduced because redundant (e.g., two *excludes* constraints between (a, b) and (b, a) are redundant and only one is necessary). Therefore, the term $\frac{ctc(fm')}{2^{|F_U|} \times 2|F_U|^2}$ is guaranteed to be less than $\frac{1}{2^{|F_U|}}$ that is the minimal possible variation of the fault ratio due to the change of evaluation of a single configuration. This means that the term can only affect the ranking of feature models having the same fault ratio.

Termination condition. In this step (step 3 in Fig. 5), the process checks whether at least one of the following conditions is met:

- a defined level of fitness Th_f is reached, i.e., there exists an fm' in P with $fitness_t(fm') \geq Th_f$. For example, $Th_f = 1$ means that we want to obtain a completely correct model without any cross-tree constraint; with $Th_f = 1 - \frac{1}{2^{|F_U|}+1}$, instead, we still want to have a correct model, but we allow to have any number of cross-tree constraints;
- in the previous Th_{NI} iterations there has been no improvement of the fitness value of the best candidate;
- a maximum number Th_i of iterations have been executed;
- a total time threshold Th_t has been reached.

If at least one of the previous conditions holds, the fm' in P with the highest fitness value is returned as final model.[10]

Selection. In the *selection* step (step 4 in Fig. 5), starting from population P, a multiset of *parents PAR* of size p is built, being p a parameter of the evolutionary process. Different selection strategies have been proposed in literature:

- *Truncation*: it selects the first $n = \lceil K \cdot |P| \rceil$ members of the population with the highest fitness value, where K is a parameter specifying a percentage of the population ($0 < K \leq 1$). Then, the first n elements are added to *PAR* as many times as necessary to reach the size p. Such strategy could result in premature convergence, as candidates with lower fitness values are not given the opportunity to improve their fitness.
- *Roulette wheel*: p members of the population are selected randomly; each member can be selected with a probability proportional to its fitness value. Note that one or more individuals could be selected multiple times.
- *Rank*: it is similar to roulette wheel, except that the selection probability is proportional to the relative fitness rather than the absolute fitness, i.e., the probability of selecting a member is inversely proportional to its ranking number (where the member with highest fitness has ranking number 1). This strategy tends to avoid premature convergence by mitigating the selection pressure that comes from large differences in fitness values (as it happens in truncation selection).

Evolution. In the *evolution* step (step 5 in Fig. 5), the parents *PAR* are used to generate the *offspring* that constitutes the population of the next generation.

The idea we assume here is that the feature model should be updated applying a limited number of mutations. Making updates through the use of mutation operators has the benefit of reducing the risk of loss of domain knowledge, by changing the feature model as less as possible. Note that this assumption is similar to the competent programmer hypothesis [23] that assumes that the user has defined the artifact close to the correct one. If our approach is used for removing faults, we can directly rely on the competent programmer hypothesis. On the other hand, if the approach is used to evolve the feature model to align it with the SPL, we can still assume that the mutation operators are sufficient to obtain the updated model; indeed, it is unlikely that the updated version of the feature model should be too different from the initial one.

In order to build the next population P, we mutate all the feature models in *PAR* using the operators presented in Table 1. We set an upper bound M to the size of the new population. If the mutation operators generate a maximum of M mutants, we take all of them as the new population, otherwise we randomly select M of them. In our approach, the offspring replaces the entire population.

Description of mutation operators In [11], we have proposed some mutation operators for feature models, divided in *feature-based* and *constraint-based* operators that are a subset of the edit operations identified in [12]. We use eight of the feature-based mutation operators proposed in [11], and introduce two new ones (`OrToAndOpt` and `AlToAndOpt`) that provide slightly different versions of `OrToAnd` and `AlToAnd`. Moreover, we also introduce two operators that permit to move a feature as sibling of the parent (`PullUp`) or as child of one its siblings (`PushDown`); we also introduce two versions of these operators that move all the children of a feature (`PullUpCh`) and all the siblings of a feature (`PushDownSibl`). Note that we do not allow the movement of a feature in any part of the feature model as this would produce too many mutants and could change too much the structure of the feature model. However, if in order to obtain the correct feature model a feature should be moved *far* from its current position, this is still obtainable by a suitable sequence of `PullUp` and `PushDown` mutations.

Finally, we use all the *constraint-based* operators[11] proposed in [11], and introduce two new ones (`AddReq`, `AddExc`)

[10] If there is more than one model with the highest fitness value, we randomly select one of these models.

[11] Note that the mutation operator `DelConstr` corresponds to operator `MC` described in [11].

Table 1: Mutation operators

Name	Description
`OptToMan`	an *optional* feature is changed to *mandatory*
`ManToOpt`	a *mandatory* feature is changed to *optional*
`OrToAl`	an *or* group is changed to *alternative*
`OrToAnd`	an *or* group is changed to *and* with all children *mandatory*
`OrToAndOpt`	an *or* group is changed to *and* with all children *optional*
`AlToOr`	an *alternative* group is changed to *or*
`AlToAnd`	an *alternative* group is changed to *and* with all children *mandatory*
`AlToAndOpt`	an *alternative* group is changed to *and* with all children *optional*
`AndToAl`	an *and* group is changed to *alternative*
`AndToOr`	an *and* group is changed to *or*
`PullUp`	a feature is moved as sibling of its parent
`PushDown`	a feature is moved as child of one of its siblings
`PullUpCh`	all children of a feature are moved as siblings of their parent
`PushDownSibl`	all siblings of a feature are moved as children of that feature
`DelConstr`	a cross-tree constraint (*requires* or *excludes*) is deleted
`ReqToExcl`	a *requires* constraint is changed to an *excludes* constraint
`ExclToReq`	an *excludes* constraint is changed to a *requires* constraint
`AddReq`	a *requires* constraint is created
`AddExc`	an *excludes* constraint is created

to create *requires* and *excludes* constraints; in order to limit the number of generated mutants, we only create constraints among features that belong to different sub-trees of the feature model, i.e., neither feature of the constraint is ancestor of the other. Moreover, we avoid to create constraints that would be redundant or that would introduce dead features: for instance, $A \rightarrow \neg B$ is not added if $A \rightarrow B$ is already present in the model, because A would become a dead feature.

Although we allow feature models with constraints also in general form, we decided to not modify them, nor introduce new ones, in order to avoid the introduction of too many mutants and to achieve better readability of the final model.

In general, we cannot evaluate a priori if a mutant introduces an anomaly; therefore, for each mutant, we check if it is infeasible, it contains redundant constraints, or it has dead features. If it has any of these anomalies, we do not select it.

Table 2: Benchmark properties

	SPL	model size input	model size target	*UR* size $\lvert F_{TRR}\rvert$	*UR* size $\lvert F_{add}\rvert$	*UR* size $\lvert F_{rem}\rvert$	*UR* size $\lvert C_{relax}\rvert$	*UR* size $\lvert C_{rem}\rvert$
BENCH$_{REAL}$	`MobileMedia` d1 (V5..8)	18.7 (15-23)	22.3 (18-26)	11.3 (1-17)	4 (3-5)	0.33 (0-1)	26.7 (0-48)	258 (24-560)
	`MobileMedia` d2 (V5..8)	16.5 (15-18)	24.5 (23-26)	12 (11-13)	8 (8-8)	0 (0-0)	64 (48-80)	536 (528-544)
	`MobileMedia` d3 (V5,V8)	15	26	9	11	0	80	1648
	`HelpSystem` (V1, V2)	25	26	0	1	0	672	2016
	`SmartHome` d1 (V1, V2)	48	58	12	23	2	1.92×10^9	2.31×10^{10}
	`ERP_SPL` (V1, V2)	48	58	0	15	0	0	1.51×10^7
	`PPU` d1 (V1..9)	13.9 (9-17)	14.9 (11-17)	0 (0-0)	1.13 (0-4)	0.125 (0-1)	3.75 (0-27)	27.8 (0-183)
	`PPU` d2 (V1..9)	13.9 (9-17)	15.9 (11-17)	0 (0-0)	2.14 (0-6)	0.142 (0-1)	9 (0-27)	54.4 (0-243)
	`PPU` d3 (V1..9)	13.9 (9-17)	15.9 (13-17)	0 (0-0)	3.5 (1-6)	0.167 (0-1)	16 (0-27)	77.5 (9-156)
	`CAR` d1 (V2009..2012)	7 (6-8)	9.33 (7-13)	0 (0-0)	2.67 (1-5)	0.333 (0-1)	5 (0-13)	16 (1-40)
	`CAR` d2 (V2009..2012)	6.5 (6-7)	10.5 (8-13)	0 (0-0)	4.5 (3-6)	0.5 (0-1)	6 (0-12)	34 (9-59)
	`CAR` (V2009..2012)	6	13	0	7	0	0	87
BENCH$_{HIST}$	`Register`	11	11	0	0	0	11.85 (0-40)	62.28 (0-210)
	`Graph`	6	6	0	0	0	12.71 (0-28)	0 (0-0)
	`Trafficlight`	13	13	0	0	0	196.86 (0-315)	53.53 (0-365)
	`Connector`	20	20	0	0	0	8.23 (0-18)	26.02 (0-336)

5. Experiments

The process [12] has been implemented in Java by using Watchmaker [13] as evolutionary framework, the Sat4j library [7] to represent and mutate feature models, and JavaBDD for BDDs manipulation.

We conducted a set of experiments to evaluate the proposed evolutionary approach; they have been executed on a Windows 10 system with an Intel i7-3770 3.40GHz processor, and 16 GB RAM.

5.1 Benchmarks

For the experiments, we used two sets of benchmarks, both shown in Table 2.

Real models. The first benchmark set BENCH$_{REAL}$ is constituted by SPLs described in literature for which different versions of their feature model have been developed. First, we have identified in the SPLOT repository[14] four SPLs that evolved over time:

- `MobileMedia`: a program to manipulate multimedia on mobile devices (four versions) [15];
- `HelpSystem`: a cyber-physical system with multiple sensors (two versions) [14];
- `SmartHome`: a set of smart house components (two versions) [13];
- `ERP_SPL`: an Enterprise Resource Planner (two versions).

Then, we have also considered the industrial case of a Pick-and-Place Unit (`PPU`) [12]; for this system, the feature model has been changed eight times to adapt to new requirements (therefore, there are nine feature models available [12]). Finally, we have also considered the product line model of a `CAR` [5], for which four different feature models have been produced.

For each SPL, we identified couples (fm_i, fm_t) of their feature models: the latest version was considered as target model[15] fm_t, and the oldest one as the initial model fm_i we want to update. For SPLs with more than two feature models, in addition to couples of feature models of consecutive versions, we also

[12] The code is available at `https://github.com/fmselab/eafmupdate`

[13] `https://watchmaker.uncommons.org/`

[14] `http://52.32.1.180:8080/SPLOT/feature_model_repository.html`

[15] Note that, in the real usage of our approach, we do not have a target feature model, but an update request *UR* from which we generate the target as propositional formula.

considered models with version distance 2 and 3 (in the table, d1, d2, and d3 indicate the distance). In this way, we attempt to reproduce update requests of different complexity. In total, $\texttt{BENCH}_{\texttt{REAL}}$ contains 36 couples of models.

Generated models. The second benchmark set $\texttt{BENCH}_{\texttt{MUT}}$ has been built with the aim of evaluating our approach under the assumption we did in Sect. 4.2.2 that mutation operators are sufficient to update the feature model. We selected four feature models developed for four SPLs (for which only one model is available):

- `Register`: a register of supermarkets, adapted from [24];
- `Graph`: a graph library;
- `Aircraft`: the configurations of the wing, the engine, and the materials of airplane models;
- `Connector`: IP connection configurations.

From these models (used as target models fm_t), we automatically generated other versions to be used as input models fm_i; we randomly mutated the target models (using 1 to 10 mutations), applying the operators described in Table 1. For each target model, we generated 100 input models. Therefore, $\texttt{BENCH}_{\texttt{MUT}}$ contains 400 couples.

5.1.1. Deriving the update request

In the devised usage of the approach, the user should specify the update request that must be provided as input to the evolutionary process; however, for our experiments, we do not have any update request available. Therefore, we automatically generated an update request *UR* from the initial feature model fm_i and the target feature model fm_t of each couple (fm_i, fm_t) of the benchmarks.

In order to detect renamed features in F_{TBR}, we manually inspected the two feature models and produced the renamed model fm_{ren}. Then, from fm_{ren} and fm_t, we automatically identified the differences of their features for building F_{add} and F_{rem}; moreover, using their BDD representation, we identified the configurations that are differently evaluated in order to build C_{relax} and C_{rem} (we built a predicate for each wrongly evaluated configuration[16]). Note that configurations that are added and removed by F_{add} and F_{rem} are not also specified in C_{relax} and C_{rem}.

Table 2 reports, for all the benchmarks, the size of the input and target models in terms of number of features, and the number of requirements of the update request. For the SPLs in $\texttt{BENCH}_{\texttt{REAL}}$ having more than one couple of feature models, the reported values are aggregated by distance; for each SPL in $\texttt{BENCH}_{\texttt{MUT}}$, the values are aggregated among its 100 input models. For these aggregated models, we report the average, minimum, and maximum number of elements in the update request. In $\texttt{BENCH}_{\texttt{MUT}}$, since we did not add or remove features for producing the input models, their size is the same of that of the target model and so F_{add} and F_{rem} are empty; moreover, we do not even rename features and so also F_{TBR} is empty.

[16] Note that, in this way, the sets C_{relax} and C_{rem} are very large because they contain a predicate for each added and removed configuration. However, in a real setting, the user should specify predicates capturing sets of configurations and so the sets should be much smaller.

Table 3: Performance of the updating process

	SPL	time (s)	initial FR (%)	final FR (%)	FR reduction
$\texttt{BENCH}_{\texttt{REAL}}$	`MobileMedia d1`	71.11	4.17e-03	1.10e-04	9
	`MobileMedia d2`	157.98	3.90e-03	4.41e-04	8
	`MobileMedia d3`	178.46	2.57e-03	3.57e-04	8
	`HelpSystem`	112.41	4.01e-03	1.24e-04	9
	`SmartHome`	1990.40	7.24e-07	8.39e-08	8
	`ERP_SPL`	2014.30	5.24e-09	8.86e-10	8
	`PPU d1`	5.98	0.09	2.27e-03	9
	`PPU d2`	10.23	0.10	0.54e-03	9
	`PPU d3`	15.90	0.09	0.01	8
	`CAR d1`	3.85	1.13	0.25	6
	`CAR d2`	4.21	1.31	0.17	8
	`CAR d3`	6.59	1.06	0.37	6
$\texttt{BENCH}_{\texttt{MUT}}$	`Register`	4.35	3.62	0.17	9
	`Graph`	0.12	19.86	0.00	10
	`Aircraft`	8.09	3.06	0.07	9
	`Connector`	28.68	3.27e-03	6.87e-05	9

5.2. Analysis

We now evaluate the proposed approach by a series of research questions. In these experiments, we set the parameters of the termination conditions as follows: Th_f to $1 - \frac{ctc(fm_i)}{2^{|F_U|} \times 2|F_U|^2}$, Th_{NI} to 15, Th_i to 25, and Th_t to 30 minutes. Note that the value chosen for Th_f requires to have a totally correct model, with at most the same number of cross-tree constraints of the starting feature model. The parameter H used to determine the maximum population size M (as defined in Sect. 4.2.2) has been set to 5, and the parameter p of the selection phase has been set to $M/2$. All the reported data are the averages of 30 runs.

In [6], we experimented the effect of the different selection policies of the evolutionary approach and we found that truncation with K=5% is the best policy in terms of fault ratio reduction, and the second best as execution time. Therefore, we here select it as selection policy and evaluate the approach using other research questions.

RQ1: *Is the proposed approach able to achieve the change requirements specified by the update request?*

For each benchmark SPL, Table 3 reports (among other things) the initial and final values of *FR*, and the *FR* reduction. For the models in $\texttt{BENCH}_{\texttt{REAL}}$, the table reports the averages among couples at the same distance; for the models in $\texttt{BENCH}_{\texttt{MUT}}$, instead, it reports the averages among the 100 input models of each SPL.[17] We observe that for all models we can reduce the fault ratio of at least 65%, with an average of

[17] Non-aggregated results are reported online at `http://foselab.unibg.it/eafmupdate/`

89.1%. Comparing the two benchmark sets, we notice that the reduction is higher in $\texttt{BENCH}_{\texttt{MUT}}$ (on average, 97.86%) than in $\texttt{BENCH}_{\texttt{REAL}}$ (on average, 86.15%); this means that, as expected, if the assumption that the models can be updated using the proposed mutation operators holds, the approach behaves very well. However, also for general models as those in $\texttt{BENCH}_{\texttt{REAL}}$, the performance of the approach is quite good.

RQ2: *Which is the computational effort of the proposed approach?*

We are here interested in the effort required by the proposed approach in terms of computation time and iterations of the evolutionary process. Table 3 also reports the total execution time of our process and the number of iterations of the evolutionary approach. For all but two models, the process takes at most 179 seconds; as expected, smaller models (as `CAR`, `Graph`, and `Register`) are updated faster than larger models (as `MobileMedia`, `HelpSystem`, `SmartHome`, and `ERP_SPL`). We observe that the process terminates when one of these three terminating conditions occurs: (i) Th_f, i.e., the model is completely updated (as in `HelpSystem` and `Graph`), (ii) Th_t, i.e., the timeout has occurred (as in `SmartHome` and `ERP_SPL`[18]), or (iii) Th_{NI}, i.e., no fitness improvement has been observed in the previous 15 iterations (as, for example, all the models of `MobileMedia d2` and `d3`, and those of `CAR d3`). Note that the process never terminates because of the maximum number of iterations Th_i.

RQ3: *Is there a relation between the initial fault ratio and its* updatability*?*

We are here interested in investigating whether there is a relation between the initial fault ratio and its reduction. Fig. 6 shows, for each benchmark model (a point in the plot), its initial fault ratio and the fault ratio reduction. It seems that there is no proper correlation: we reduce (or do not reduce) the fault ratio in the same proportion among models having different initial fault ratios. We checked the correlation with the Spearman rank-order correlation coefficient [25], and indeed we found a value of 0.23 indicating almost no correlation [26].

RQ4: *Are the final models similar to those produced by SPL designers?*

The main aim of the proposed approach is to obtain a feature model that satisfies all the change requirements; to this purpose, we use the fault ratio as measure of model correctness. However, the final model we obtain, although correct, may be not readable by and not useful for an SPL designer. Although we could not ask real SPL designers to validate our models in terms of usefulness and readability, we have access to models developed by SPL designers to achieve the same update requests we tackled in the experiments (models fm_t used as target in the experiments). We can assume that SPL designers are likely to find our models useful if models produced by our process are similar to their models. We therefore measure the *readability* of the final model of our process in terms of *distance* from the model fm_t developed by a designer for the same update request. We compute the *edit distance* [27, 28] of the final model fm_f from fm_t, defined as the number of *edits* (insertion, deletion, and rename of tree nodes) that we have to apply to fm_f in order to obtain fm_t.[19] Table 3 reports also the edit distance (ED) to the target model (average among the models), and the percentage of models that are syntactically equal and semantically equivalent to the target model. Of course, models not completely updated are not syntactically equal to the target and have edit distance greater than 0. Completely correct models (semantically equivalent) are often also syntactically equal (for example, 86.67% of the `PPU d1` models are semantically equivalent and 48.33% are also syntactically equal); however, there are some correct final models that are different from the target model fm_t (for example, `HelpSystem` is completely updated 86.67% of the times, but always in a different way than fm_t).

RQ5: *Does considering the number of cross-tree constraints in the fitness impact the final results?*

As explained in Sect. 4.2.2, our fitness function (see Eq. 1) can also take into consideration the number of cross-tree constraints (*ctc*); the value we selected for k (see Eq. 2) has the aim of penalizing models with higher *ctc* at the same fault ratio (in order to limit the insertion of such constraints and instead give precedence to changes of the parental relations). In order to assess the impact of this choice, we have executed the same experiments presented before with $k = 0$ in the fitness function (that becomes $fitness_t(fm') = 1 - FR(fm', t)$) and $Th_t = 1$. Table 4 reports the data in terms of average fault ratio reduction, percentage of semantically equivalent, percentage of syntactically equal models, and the average number of cross-tree constraints. For easing the comparison, we also report the same data of the results obtained with the previous experiment (already reported in Table 3). In order to check whether the consideration of cross-tree constraints has some effect on the final results, we have applied to the data the classical hypothesis testing by performing the Wilcoxon signed-rank test[20] [25] between the results with the two versions of fitness. The null hypothesis that considering *ctc* has no impact of the final model cannot be rejected for the percentage of totally updated models (i.e., semantically equivalent), but it is rejected for the syntactical equivalence with p-value equal to 0.0038. This confirms that penalizing the usage of cross-tree constraints in the fitness improves the quality (readability) of the final model without compromising the ability of the approach in achieving the update request.

[18] Note that the time for these two models is around 3.5 minutes above the threshold Th_t of 30 minutes; indeed, the terminating condition is checked at the end of an evolution step, but the threshold could be overcame during the step.

[19] Note that these edit operations are more fine-grained than our mutation operators and we are always able to compute the distance between two feature models.

[20] We performed a non-parametric test as we found, with the Shapiro-Wilk test, that the distributions are not normal.

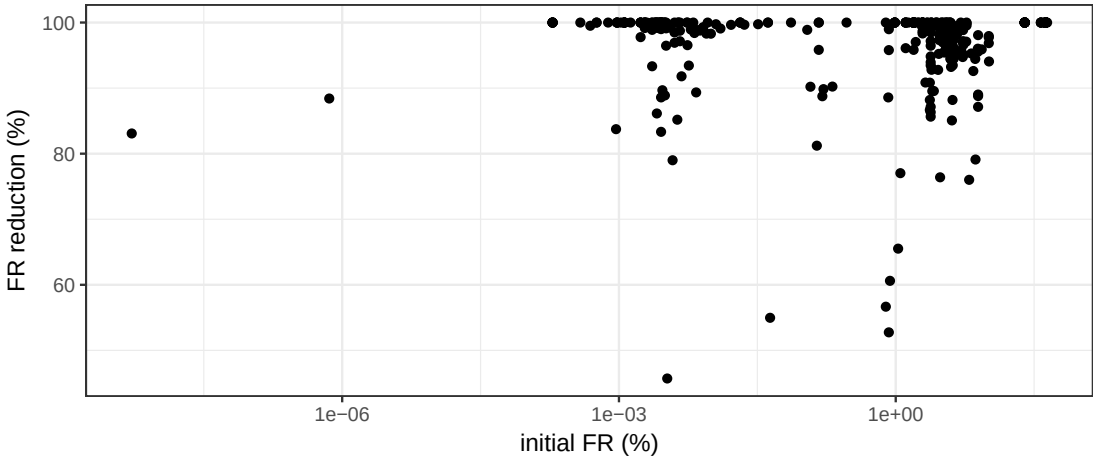

Figure 6: Relation between the initial fault ratio and the fault ratio reduction

Table 4: Performance of the updating process with the two versions of the fitness

	SPL	Fitness without constr. ($k = 0$)				Fitness with constr. (k as in Eq. 2)			
		FR reduction (%)	sem. eq. (%)	synt. eq. (%)	ctc.	FR reduction (%)	sem. eq. (%)	synt. eq. (%)	ctc.
$BENCH_{REAL}$	MobileMedia d1	94.10	61.11	17.78	0.69	96.03	64.44	17.78	0.23
	MobileMedia d2	84.83	0.00	4.00	1.85	86.54	0.00	1.00	0.63
	MobileMedia d3	85.16	0.00	0.00	2.43	86.13	0.00	0.00	0.80
	HelpSystem	97.22	86.67	0.00	5.57	96.92	86.67	0.00	5.57
	SmartHome	88.42	0.00	0.00	0.53	88.42	0.00	0.00	0.43
	ERP_SPL	82.96	0.00	0.00	0.40	83.08	0.00	0.00	0.43
	PPU d1	98.10	87.50	45.83	1.62	97.44	86.67	48.33	1.23
	PPU d2	95.89	68.57	37.14	2.11	96.48	68.57	36.67	1.39
	PPU d3	87.25	50.00	27.22	2.93	87.43	48.89	28.89	1.94
	CAR d1	73.61	55.56	3.33	1.89	69.81	57.78	6.67	0.77
	CAR d2	78.06	45.00	0.00	3.22	80.31	50.00	0.00	1.83
	CAR d3	62.80	0.00	0.00	4.63	65.52	0.00	0.00	2.37
$BENCH_{MUT}$	Register	96.11	82.70	64.03	0.60	95.84	80.27	65.20	0.22
	Graph	100.00	100.00	99.37	0.00	100.00	100.00	99.03	0.00
	Aircraft	97.43	85.97	68.73	0.43	97.49	84.53	68.63	0.29
	Connector	98.05	93.70	63.00	0.48	98.11	94.00	68.47	0.20

6. Threats to validity

We discuss the threats to the validity of our results along two dimensions, *external* and *internal* validity [25].

6.1. External validity

Regarding external validity, a threat is that the obtained results could be not generalizable to real-world (industrial) feature models having specific update requests. However, as first benchmark, we have selected 9 couples of models showing the evolution of real SPLs [15, 14, 13] taken from the SPLOT repository, and other 27 couples from the evolution of other two real SPLs described in literature [12, 5] (see Sect. 5.1); moreover, in order to enlarge the set of evaluated models, we generated 4000 pull models by randomly mutating other 4 feature models (acting as target). We believe that this way of selecting the benchmarks reduces the bias w.r.t. other real models this process may be applied to in the future.

Another threat to the external validity could be that the proposed process does not scale well to models larger than those considered in the experiments. In particular, the time for computing the fitness function grows exponentially with the number of features, as it is based on BDD construction. In order to address this problem, as future work we plan to devise a technique that, given a single change requirement c, identifies the sub-tree st of the feature model *affected* by c, i.e., c can be achieved only modifying st. Considering st would allow to improve the process performance, as the mutations would be more targeted and the fitness computation would be performed on smaller BDD.

Another threat to the external validity is that the update requests that we use in the experiments could be different by those written by SPL designers. However, update requests have been obtained by computing the difference of consecutive versions of feature models written by SPL designers. While change requirements F_{TBR}, F_{add}, and F_{rem} are guaranteed to be the same as those specified by SPL designers, C_{relax} and C_{rem} are only semantically equivalent. However, this is not a threat, as the evolutionary approach only considers the semantics of the C_{relax} and C_{rem}, not their structure.

6.2. Internal validity

Regarding internal validity, a threat is that the obtained results could depend on the values chosen for the parameters of

the evolutionary process (parameters of termination conditions, and parameters of the selection and evolution phases) and that, with some other values, the results would have been different (e.g., a given selection strategy could perform better); although we kept all the parameters fixed, we believe that the overall result that our approach is able to actually update the feature model is not affected. However, as future work, we plan to perform a wider set of experiments in which the effect of each single parameter is evaluated. For finding the best parameter setting, we could use a parameter-tuning framework as irace [29].

7. Related work

Different approaches have been proposed for updating and/or repairing feature models.

In a previous work, we proposed a technique to generate *fault-detecting configurations* (tests) able to show *conformance faults* (i.e., configurations wrongly accepted or wrongly rejected) in feature models [11]; in [30], we then presented an iterative process based on mutation that first shows these fault-detecting configurations to the user who must assess their correct evaluation, and then modifies the feature model to remove the faults (if any). The approach proposed here is different, since it is based on an evolutionary approach, it is completely automatic, and does not require the interaction with the user who must only provide the initial update request. Moreover, in the current approach we consider update requests not only coming from failing tests but also from the normal evolution of the SPL.

Another approach trying to remove faults from feature models is presented in [31]: it starts from a feature model and, through a cycle of *test-and-fix*, improves it by removing its wrong constraints; the approach uses configurations derived both from the model and from the real system and checks whether these are correctly evaluated by the feature model. The approach is similar to ours in considering wrong configurations, but does not allow to add and remove features. The main differences with our approach are that we have a precise definition of target we need to reach, we rely on an evolutionary approach, and we assume that the model evolution can be obtained through mutation.

In [32], we proposed an approach to repair variability models by modifying the constraints of the model using some *repairs*; that approach differs from the one presented in this work in different aspects. First of all, the oracle (similar to our target) in [32] is given by the implementation constraints, while here the target comes from update requests. Then, the aim of [32] is only to remove faults from the model, while here we also support the evolution of the model. Finally, the approach in [32] always improves the conformity index (similar to our fitness function) during the process, with the risk of obtaining local optima; in the current approach, instead, we maintain a set of candidate solutions in which some of them may decrease the fitness function in some iteration, but that could obtain a better result at the end.

Regarding the use of evolutionary algorithms for feature models, the work in [33] proposes a process to reverse engineer feature models starting from a set of products: the process starts from a population of randomly generated models and evolves it using as fitness function the number of correctly evaluated products. The approach is similar to ours in using an evolutionary approach based on mutation (some used mutation operators are similar to ours), but differs in the aim and in the starting point: we start from an existing feature model that we want to update to achieve some change requirements (removing faults or business requirements), while the approach in [33] wants to build a new feature model starting from some known products.

Evolutionary approaches have been widely used also for testing and repairing programs. For example, GenProg [34] is a repair tool based on genetic programming. It uses mutation and crossover operators to search for a program variant that passes all tests.

8. Conclusion

We proposed an evolutionary approach that, given a set of change requirements in terms of features to rename, features to add/remove to/from existing products, and products to add/remove from a feature model, through a sequence of mutations, tries to obtain another feature model that exactly captures the requested changes. The approach tries to achieve all the change requirements without producing unnecessary complex models, by limiting the number of cross-tree constraints.

We evaluated the approach on feature models of ten SPLs. For some of these SPLs, different versions of their feature models have been produced during the evolution of the SPL: this allowed us to assess the approach on update requests of different complexity. Experiments showed that the process is indeed able to update the feature model, although the update could be partial when the model is particularly big. On the other hand, it seems that the initial fault ratio does not influence the percentage of fault ratio that can be reduced.

Some models cannot be completely updated by the proposed approach. One of the reasons could be that we produce too many mutants and we lose time in evaluating all of them (as for `SmartHome` in the experiments); as future work, we plan to devise more tailored ways to select the mutants, maybe reasoning on the structure of the update request. In a similar way, we could also reduce the size of the initial feature model by keeping only the part that is interested by the change requirements; this should permit to obtain a better process performance. Moreover, in this work we did not consider abstract features [8] that, however, could be necessary to achieve some change requirements. As future work, we plan to use also abstract features in the updating process, but only when really needed, since too many abstract features could reduce readability.

Experiments show that the models we obtain at the end of the updating process, although semantically correct, could be not so readable. The fitness function already tries to reduce the complexity (and so preserve the readability) of the model by penalizing the addition of cross-tree constraints; as future work, we plan to integrate additional syntactical measures in the fitness function.

Other approaches could be devised to achieve the change requirements of the update request. One possible way could be

to add the update request as cross-tree constraints and then simplify them using approaches as those described in [20, 9]. As future work, we plan to compare our approach with this alternative approach in terms of readability and other qualities of the obtained final models, such as compactness, traceability, and maintainability.

Moreover, in the experiments we used a given setting for the parameters of the evolutionary approach. As future work, we plan to use a framework such as irace to find the best parameters setting.

References

References

[1] D. Benavides, P. Trinidad, A. Ruiz-Cortés, Automated reasoning on feature models, in: Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 491–503. doi:10.1007/11431855_34.

[2] D. Batory, Feature models, grammars, and propositional formulas, in: Proceedings of the 9th International Conference on Software Product Lines, SPLC'05, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 7–20. doi:10.1007/11554844_3.

[3] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, Y. Bontemps, Generic semantics of feature diagrams, Computer Networks 51 (2) (2007) 456 – 479. doi:10.1016/j.comnet.2006.08.008.

[4] D. Benavides, S. Segura, A. Ruiz-Cortés, Automated analysis of feature models 20 years later: A literature review, Information Systems 35 (6) (2010) 615–636. doi:10.1016/j.is.2010.01.001.

[5] A. Pleuss, G. Botterweck, D. Dhungana, A. Polzer, S. Kowalewski, Model-driven support for product line evolution on feature level, Journal of Systems and Software 85 (10) (2012) 2261–2274. doi:10.1016/j.jss.2011.08.008.

[6] P. Arcaini, A. Gargantini, M. Radavelli, An evolutionary process for product-driven updates of feature models, in: Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems, VAMOS 2018, ACM, New York, NY, USA, 2018, pp. 67–74. doi:10.1145/3168365.3168374.

[7] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, G. Saake, Mastering Software Variability with FeatureIDE, Springer, 2017. doi:10.1007/978-3-319-61443-4.

[8] T. Thüm, C. Kastner, S. Erdweg, N. Siegmund, Abstract features in feature modeling, in: Proceedings of the 2011 15th International Software Product Line Conference, SPLC '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 191–200. doi:10.1109/SPLC.2011.53.

[9] A. Knüppel, T. Thüm, S. Mennicke, J. Meinicke, I. Schaefer, Is there a mismatch between real-world feature models and product-line research?, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, ACM, New York, NY, USA, 2017, pp. 291–302. doi:10.1145/3106237.3106252.

[10] T. Thüm, D. Batory, C. Kastner, Reasoning about edits to feature models, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 254–264. doi:10.1109/ICSE.2009.5070526.

[11] P. Arcaini, A. Gargantini, P. Vavassori, Generating tests for detecting faults in feature models, in: Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on, 2015, pp. 1–10. doi:10.1109/ICST.2015.7102591.

[12] J. Bürdek, T. Kehrer, M. Lochau, D. Reuling, U. Kelter, A. Schürr, Reasoning about product-line evolution using complex feature model differences, Automated Software Engineering 23 (4) (2016) 687–733. doi:10.1007/s10515-015-0185-3.

[13] A. R. Santos, R. P. de Oliveira, E. S. de Almeida, Strategies for consistency checking on software product lines: A mapping study, in: Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering, EASE '15, ACM, New York, NY, USA, 2015, pp. 5:1–5:14. doi:10.1145/2745802.2745806.

[14] V. Štuikys, R. Burbaitė, K. Bespalova, G. Ziberkas, Model-driven processes and tools to design robot-based generative learning objects for computer science education, Science of Computer Programming 129 (2016) 48–71, special issue on eLearning Software Architectures. doi:10.1016/j.scico.2016.03.009.

[15] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, F. Dantas, Evolving software product lines with aspects: An empirical study on design stability, in: Proceedings of the 30th International Conference on Software Engineering, ICSE '08, ACM, New York, NY, USA, 2008, pp. 261–270. doi:10.1145/1368088.1368124.

[16] A. Durán, D. Benavides, S. Segura, P. Trinidad, A. Ruiz-Cortés, 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. doi:10.1007/s10270-015-0503-z.

[17] S. Urli, A. Bergel, M. Blay-Fornarino, P. Collet, S. Mosser, A visual support for decomposing complex feature models, in: 2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT), 2015, pp. 76–85. doi:10.1109/VISSOFT.2015.7332417.

[18] I. Reinhartz-Berger, K. Figl, Ø. Haugen, Comprehending feature models expressed in CVL, in: J. Dingel, W. Schulte, I. Ramos, S. Abrahão, E. Insfran (Eds.), Model-Driven Engineering Languages and Systems, Springer International Publishing, Cham, 2014, pp. 501–517. doi:10.1007/978-3-319-11653-2_31.

[19] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, A. Wąsowski, A survey of variability modeling in industrial practice, in: Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '13, ACM, New York, NY, USA, 2013, pp. 7:1–7:8. doi:10.1145/2430502.2430513.

[20] A. von Rhein, A. Grebhahn, S. Apel, N. Siegmund, D. Beyer, T. Berger, Presence-condition simplification in highly configurable systems, in: Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 178–188.

[21] A. E. Eiben, J. E. Smith, Introduction to Evolutionary Computing, Springer Verlag, 2003.

[22] K. Sörensen, Metaheuristics – the metaphor exposed, International Transactions in Operational Research 22 (1) (2013) 3–18. doi:10.1111/itor.12001.

[23] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Trans. Softw. Eng. 37 (5) (2011) 649–678. doi:10.1109/TSE.2010.62.

[24] D. Shimbara, Ø. Haugen, Generating configurations for system testing with common variability language, in: Proceedings of the 17th International SDL Forum on SDL 2015: Model-Driven Engineering for Smart Cities - Volume 9369, Springer-Verlag New York, Inc., New York, NY, USA, 2015, pp. 221–237. doi:10.1007/978-3-319-24912-4_16.

[25] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, A. Wessln, Experimentation in Software Engineering, Springer Publishing Company, Incorporated, 2012.

[26] D. Hinkle, W. Wiersma, S. Jurs, Applied Statistics for the Behavioral Sciences, no. 663 in Applied Statistics for the Behavioral Sciences, Houghton Mifflin, 2003.

[27] M. Pawlik, N. Augsten, Efficient Computation of the Tree Edit Distance, ACM Transactions on Database Systems 40 (1) (2015) 1–40. doi:10.1145/2699485.

[28] M. Pawlik, N. Augsten, Tree edit distance: Robust and memory-efficient, Information Systems 56 (2016) 157–173. doi:10.1016/j.is.2015.08.004.

[29] M. López-Ibáñez, J. Dubois-Lacoste, L. P. Cáceres, M. Birattari, T. Stützle, The irace package: Iterated racing for automatic algorithm configuration, Operations Research Perspectives 3 (2016) 43–58. doi:10.1016/j.orp.2016.09.002.

[30] P. Arcaini, A. Gargantini, P. Vavassori, Automatic detection and removal of conformance faults in feature models, in: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), 2016, pp. 102–112. doi:10.1109/ICST.2016.10.

[31] C. Henard, M. Papadakis, G. Perrouin, J. Klein, Y. Le Traon, Towards automated testing and fixing of re-engineered feature models, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 1245–1248.

doi:10.1109/ICSE.2013.6606689.

[32] P. Arcaini, A. Gargantini, P. Vavassori, Automated repairing of variability models, in: Proceedings of the 21st International Systems and Software Product Line Conference - Volume A, SPLC '17, ACM, New York, NY, USA, 2017, pp. 9–18. doi:10.1145/3106195.3106206.

[33] R. E. Lopez-Herrejon, L. Linsbauer, J. A. Galindo, J. A. Parejo, D. Benavides, S. Segura, A. Egyed, An assessment of search-based techniques for reverse engineering feature models, Journal of Systems and Software 103 (2015) 353–369. doi:http://dx.doi.org/10.1016/j.jss.2014.10.037.

[34] C. Le Goues, M. Dewey-Vogt, S. Forrest, W. Weimer, A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each, in: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 3–13. doi:10.1109/ICSE.2012.6227211.