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...
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 types2:

- 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 → B.
- A excludes B: A and B cannot be part of the same product. We also indicate it as A → ¬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 xor(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 ⊆ 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, ..., 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 = ¬f_i means that f_i does not belong to the configuration. We will also represent a configuration as a BOolean Formula as follows: xor(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 xor(fm, U) = xor(fm) ∧ \bigwedge_{f∈U} ¬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...
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, (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, defined over a set of features F, we call update request UR the modifications a user wants to apply to fm 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: \( F_{\text{TRR}} = \{ f_1, \ldots, f_m \} \) is a subset of features of \( F \) that must be renamed and \( \text{ren} \) a renaming function; \( F_R \) is the set obtained by replacing every feature \( f_i \in F_{\text{TRR}} \) with \( \text{ren}(f_i) \), i.e., \( F_R = (F \setminus F_{\text{TRR}}) \cup \bigcup_{f_i \in F_{\text{TRR}}} \{ \text{ren}(f_i) \} \). We identify with \( fm_{\text{ren}} \) the feature model obtained by renaming the features according to \( F_{\text{TRR}} \) and \( \text{ren} \).\(^4\)

The other two feature-based change requirements over the features of \( fm_{\text{ren}} \) are:

- Add features: \( F_{\text{add}} \) is a set of features the user wants to add to \( fm_{\text{ren}} \). For each \( e \in F_{\text{add}} \), the user has to define in \( F_R \cup F_{\text{add}} \) 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: \( F_{\text{rem}} \) 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 \( (F_R \cup F_{\text{add}}) \setminus F_{\text{rem}} \).

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

\(^4\)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.
quests are computed as di-
elts, and C

3.1. Well-formedness of an update request

The meaning of C

and C

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

, but deprived of the products identified by C

; note that C

could also remove some products added by C

.

If the user wants to include/exclude a specific configuration c, then (s)he simply adds \(\text{rem}(c)\) to C

/C

.

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

must have an ancestor in F

\(- F

\). This implies that it is not possible to remove by F

the parents p of features added in F

, i.e., F

\(\cap (\cup_{f \in F

\text{add}}(\text{parent}(f))) = \emptyset\).

2. It is not possible to remove features that have been re-
named, i.e., F

\(\cap (\cup_{f \in F

\text{rem}}(\text{rem}(f))) = \emptyset\).

3. Predicates in C

cannot predicate over features that have been removed in F

.

4. C

actually increases the set of products: for each \(p_i\) in C

, there exists at least one non-valid configuration that satisfies \(p_i\).

5. C

actually deletes some products: for each \(\gamma_i\) in C

, there exists at least one product that satisfies \(\gamma_i\).

6. C

does not remove all the configurations that must be added by C

, i.e., for each \(p_i\) and \(\gamma_j\) there exists a configuration that satisfies \(p_i\) (it must be added) and it does not satisfy \(\gamma_j\).

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 F

(i.e., \(\text{rem}(F

\text{rem})\)) and the predicates in C

and C

; for example, checking constraint 4 consists in verifying that, for each \(p_i \in C

\), \(\neg \text{rem}(F

\text{rem}) \land p_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\(^\dagger\):

- **Rename features** F

= [MonochromeRadioDisplay] and ren (MonochromeRadioDisplay) = RadioDisplay.
- **Add features** F

= {DVDEntertainment} and parent(DVDEntertainment) = MultimediaDevices.
- **Remove features** F

= {[OtherFeatures]}.
- **Add products** C

= [MultimediaDevices \(\land\) Navigation \(\land\) MonochromeNavigationDisplay\(\leftrightarrow\) Color-NavigationDisplay]).

We also want to allow products with support for navigation, and having both displays or having no display at all.
- **Remove products** C

= {Navigation \(\land\) 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.

In order to define the semantics of F

and C

, 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).

\(^\dagger\)Note that the change requirement F

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

, 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

and F

resemble similar change requirements observed in literature, respectively in the SmartHome SPL [13] and in the CAR SPL [5].
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 \( \text{filter}(fm, K) \) as follows:

\[
\text{filter}(fm, K) = \begin{cases} 
\text{bor}(fm) & \text{if } K = \emptyset \\
\text{filter}(fm, K') & \text{if } K = K' \cup \{ f \}
\end{cases}
\]

The formula \( \text{filter}(fm, K) \) has as logical models the same models as \( \text{bor}(fm) \) except that all the features in \( K \) have been removed.

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

\[
\phi_{\text{rem}} = \text{filter}(fm_{\text{rem}}, F_{\text{rem}})
\]

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

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

\[
\phi_{\text{relax}} = \text{filter}(\phi_{\text{rem}}, \text{features}(\rho_i)) \land \rho_i
\]

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

We can now build the formula that defines the semantics of the whole update request. We name the formula as \( \text{target} \) 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 \( UR = \{ \text{remove}, \text{add products}, C_{\text{relax}}, C_{\text{rem}} \} \) defined over a feature model \( fm \), with the functions \( \text{ren} \) defined over \( F_{\text{filter}} \) and parent defined over \( F_{\text{add}} \). Let \( fm_{\text{rem}} \) be the feature model renamed according to \( \text{ren} \); the target is defined as:

\[
t = ( \text{remove } F_{\text{rem}} \text{ from products} \vee \text{add products} ) \land \phi_{\text{relax}}
\]

\[
\text{add } F_{\text{add}} \text{ features} \land \text{disable } F_{\text{rem}} \text{ features} \land \text{remove products}
\]

\[
\bigwedge_{f \in F_{\text{add}}} f \rightarrow \text{parent}(f) \land \bigwedge_{f \in F_{\text{rem}}} \neg f \land \bigwedge_{\gamma \in C_{\text{rem}}} \neg \gamma
\]

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

Note that the target correctly predicates over all the features \( F_U = F' \cup F_{\text{rem}} \): 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; if 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 \( F_{\text{rem}} \), 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 (\( \models t \leftrightarrow \text{bor}(fm', F_U) \)). In the following, we will only compare feature models \( fm' \) whose features \( F_{\text{filter}} \) are, at most, those in \( F_U \), i.e., \( F_{\text{filter}} \subseteq F_U \).

Although a feature model could not fulfill all the change requirements, it could satisfy them partially. We give a measure
of the difference between a feature model \( fm \) (either the initial one \( fm \) 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:

\[
FR(fm, t) = \frac{|AllConfs(\varphi) \cap (fm, F_U) \neq t, F_U)|}{2^{|F_U|}}
\]

where \( AllConfs(\varphi, V) \) returns all the logical models of formula \( \varphi \), 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 \( fm_1 \) w.r.t. to the target \( t \) (that corresponds to the final feature model) is \( \frac{20}{2^{|F_U|}} \), as \( fm_1 \) 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 \( fm \), 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., \( \not \equiv t = \text{sat}(fm_{AR}, F_U) \). 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:

---

In our approach, we represent formulas as Binary Decision Diagrams (BDDs) in JavaBDD that implements \(|AllConfs|\) by means of the method \texttt{satCount} that directly computes the cardinality of the set without enumerating all the models.
(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:

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

(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 $k$:

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

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 $\frac{1}{2|F_U|^2}$ is guaranteed to be less than $\frac{1}{2|F_U|^2}$ 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 $fitness(fm') \geq Th_f$. For example, $Th_f = 1$ means that we want to obtain a completely correct model without any cross-tree constraint; with $Th_f = 1 - \frac{1}{2^{|F_U|^2}}$, instead, we still want to have a correct model, but we allow to have any number of cross-tree constraints;
- in the previous $Th_h$ iterations there has been an 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 $n = \lceil K \times |P| \rceil$ 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 ($OrToAnd$ and $AndToOr$) that provide slightly different versions of $OrToAnd$ and $AndToOr$.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OptToMan</td>
<td>an optional feature is changed to mandatory</td>
</tr>
<tr>
<td>ManToopt</td>
<td>a mandatory feature is changed to optional</td>
</tr>
<tr>
<td>ExclToReq</td>
<td>an or group is changed to alternative</td>
</tr>
<tr>
<td>OrToAnd</td>
<td>an or group is changed to and with all children</td>
</tr>
<tr>
<td>OrToAndopt</td>
<td>an or group is changed to and with all children</td>
</tr>
<tr>
<td>AndToExcl</td>
<td>an alternative group is changed to or</td>
</tr>
<tr>
<td>AltToAnd</td>
<td>an alternative group is changed to and with all children</td>
</tr>
<tr>
<td>AltToAndopt</td>
<td>an alternative group is changed to and with all children</td>
</tr>
<tr>
<td>PullUpSibl</td>
<td>a feature is moved as sibling of its parent</td>
</tr>
<tr>
<td>PushDown</td>
<td>a feature is moved as child of one of its siblings</td>
</tr>
<tr>
<td>PullUpCh</td>
<td>all children of a feature are moved as siblings of their parent</td>
</tr>
<tr>
<td>PushDownSibl</td>
<td>all siblings of a feature are moved as children of that feature</td>
</tr>
<tr>
<td>DelConstr</td>
<td>a cross-tree constraint (requires or excludes) is deleted</td>
</tr>
<tr>
<td>ReqToExcl</td>
<td>a requires constraint is changed to an excludes constraint</td>
</tr>
<tr>
<td>ExclToReq</td>
<td>an excludes constraint is changed to a requires constraint</td>
</tr>
<tr>
<td>AddReq</td>
<td>a requires constraint is created</td>
</tr>
<tr>
<td>AddExc</td>
<td>an excludes constraint is created</td>
</tr>
</tbody>
</table>

\(^{10}\)If there is more than one model with the highest fitness value, we randomly select one of these models.
over, we also introduce two operators that permit to move a feature as sibling of the parent (PullUpSib1) or as child of one of 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 (PushDownSib1). 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, 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) 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 \(\rightarrow \sim B\) 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.

5. Experiments

The process\(^ {12}\) has been implemented in Java by using Watchmaker\(^ {13}\) as evolutionary framework, FeatureIDE [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:

\(^{11}\)Note that the mutation operator DelConstr corresponds to operator MC described in [11].

\(^{12}\)The code is available at https://github.com/fmselab/eaftupdate

\(^{13}\)https://watchmaker.uncommons.org/

\(^{14}\)http://52.32.1.180:8080/SPLOT/feature_model_repository.html

- 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 (\(f_m, f_m\)) of their feature models: the latest version was considered as target model\(^ {15}\) \(f_m\), and the oldest one as the initial model \(f_m\) we want to update. For SPLs with more than two feature models, in addition to couples of feature models of consecutive versions, we also 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, BENCH\(_{REAL}\) contains 36 couples of models.

**Generated models.** The second benchmark set BENCH\(_{GEN}\) 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 \(f_m\)), we automatically generated other versions to be used as input models \(f_m\); 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, BENCH\(_{GEN}\) contains 400 couples.

\(^{15}\)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.
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 the benchmarks.

In order to detect renamed features in \( F_{TBR} \), we manually inspected the two feature models and produced the renamed model \( fm_r \). Then, from \( fm_r \) 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 \( C_{ren} \) (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 \( C_{ren} \).

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 \( \text{BENCH}_{\text{REAL}} \) having more than one couple of feature models, the reported values are aggregated by distance; for each SPL in \( \text{BENCH}_{\text{BENCH}} \), 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 \( \text{BENCH}_{\text{BENCH}} \), 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.

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 - \( c\text{ic}(fm_r) \) from Table 3, \( Th_{N\text{f}} \) to 15, \( Th_h \) to 25, and \( Th_{res} \) 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 \( \text{BENCH}_{\text{REAL}} \), the table reports the averages among couples at the same distance; for the models in \( \text{BENCH}_{\text{BENCH}} \), 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 \( 89.1\% \). Comparing the two benchmark sets, we notice that the reduction is higher in \( \text{BENCH}_{\text{BENCH}} \) (on average, \( 97.86\% \)) than in \( \text{BENCH}_{\text{REAL}} \) (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 \( \text{BENCH}_{\text{REAL}} \), 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 process. For all but two models, the process takes at most 179 seconds; as expected, smaller models (as \( \text{CAR}, \text{Graph}, \text{Register} \) and \( \text{Register} \)) are faster than larger models (as \( \text{MobileMedia}, \text{HelpSystem}, \text{SmartHome}, \text{ERP}_{\text{SPL}} \)). 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 \( \text{HelpSystem} \) and \( \text{Graph} \)), (ii) \( Th_h \), i.e., the timeout has occurred (as in \( \text{SmartHome} \) and \( \text{ERP}_{\text{SPL}} \)), or (iii) \( Th_{N\text{f}} \), i.e., no fitness improvement has been observed in the previous 15 iterations (as, for example, all the models of \( \text{MobileMedia} \) and \( \text{d3} \), and those of \( \text{CAR} \) \( \text{d3} \)). Note that the process never terminates because of the maximum number of iterations \( Th_h \).

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

\(^{16}\)Note that, in this way, the sets \( C_{relax} \) and \( C_{ren} \) 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.

\(^{17}\)Non-aggregated results are reported online at http://fotolab.unibg. it/eafmapdata/
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 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 developed by a designer for the same update request. We compute the edit distance [27, 28] of the final model fm_t from fm_t, defined as the number of edits (insertion, deletion, and renaming of tree nodes) that we have to apply to fm_t 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 syntactically equivalent and 48.33% are also syntactically equal); however, there are some correct final models that are different from the target model fm, (for example, HelpSystem is completely updated 86.67% of the times, but always in a different way than fm).

**RQ5:** Does considering the number of cross-tree constraints in the fitness function 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 \(\text{fitness}(fm') = 1 - FR(fm', i)\) and \(TH_i = 1\). 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.

### 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

---

19Note 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.

20We performed a non-parametric test as we found, with the Shapiro-Wilk test, that the distributions are not normal.
400 input 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 by only modifying $st$; only considering $st$ would allow to improve the process performance, as the mutations would be more targeted and the fitness computation would be performed on a 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