An evolutionary process for product-driven updates of feature models

Paolo Arcaini[*]
Charles University
Faculty of Mathematics and Physics,
Czech Republic
arcaini@d3s.mff.cuni.cz

Angelo Gargantini
University of Bergamo
Bergamo, Italy
angelo.gargantini@unibg.it

Marco Radavelli
University of Bergamo
Bergamo, Italy
marco.radavelli@unibg.it

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 able 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. Our method is based on an evolutionary algorithm and it iteratively applies structure-preserving mutations to the original model, until the model is completely updated. We evaluate the process on real-world feature models. Although our approach does not guarantee to completely update all possible feature models, empirical analysis shows that, on average, more than 80% of requested changes are applied.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; *Search-based software engineering*; • **Theory of computation → ** *Evolutionary algorithms*;

KEYWORDS

feature models, software product lines, mutation, search-based software engineering

ACM Reference Format:
Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. 2018. An evolutionary process for product-driven updates of feature models. In *VAMOS 2018: 12th International Workshop on Variability Modelling of Software-Intensive Systems, February 7–9, 2018, Madrid, Spain.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3168365.3168374

[*]The research reported in this paper has been partially supported by the Czech Science Foundation project number 17-12465S.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
VAMOS 2018, February 7–9, 2018, Madrid, Spain
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5398-4/18/02…$15.00
https://doi.org/10.1145/3168365.3168374

1 INTRODUCTION

Software Product Lines (SPLs) are families of products that share some common characteristics, and differ on some others [4, 6]. 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* [17]; 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 representing 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 [5].

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 *change requirements* can be identified: either the model is incorrect (it excludes/includes some products that should be included/excluded), or the SPL is 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 [16]). Manually updating a feature model to achieve all change requirements could 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 evolutionary 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): some features (s)he wants to add to and remove from the original feature model, some configurations that should be no more accepted as products by the final model, and some configurations that should instead be accepted as new products. Then, starting from the update request, the proposed evolutionary process tries to obtain a feature model that exactly captures all the specified changes; 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 the percentage of correctly evaluated configurations.

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.

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 basic feature model is a hierarchically arranged set of features, where each parent-child relation between them is a constraint of the following types[1]:

- *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.

Feature models can be visually represented by means of feature diagrams.

Feature models semantics can be expressed by using propositional logic [4, 5]: features are represented by propositional letters, and relations among features by propositional formulae. We identify with $\textsc{bof}(fm)$ the logic representation of 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., $c \subseteq F$)*[2].

If *fm* has n features, there are 2^n possible configurations, which, however, are not all valid.

Definition 2 (**Validity**). *Given a feature model fm, a configuration* c *is* valid *if it respects the constraints of fm. A valid configuration is called* product. *We represent the validity of a configuration* c *over a feature model fm by the predicate $val(fm, c)$.*

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 $c = \{l_1, \ldots, l_n\}$ as 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 [1, 19].

3 SPECIFYING AN UPDATE REQUEST

We suppose that the product line engineer wants to update an existing feature model *fm*; although (s)he knows which are the desired updates in terms of products and features to add or remove, (s)he doesn't 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.

Definition 3 (**Update request**). *Given a 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 the following four sets:*

- *$UR_{F_{add}} = \{(f_1, p_1), \ldots, (f_m, p_m)\}$ where each f_i is a new feature (i.e., $f_i \notin F$) and $p_i \in F$ is an existing feature that must become the parent of f_i. We collect the new features in the set $F_{add} = \{f_1, \ldots, f_m\}$. By adding a feature f_i with parent p_i, the user wants to duplicate all the products that contain p_i by adding also f_i;*
- *F_{rem} is a subset of the features of F to remove; by removing a feature f, the user wants to remove f from all the products. The only constraint on F_{rem} is that it is not possible to remove the parents p of features added in F_{add}, i.e., $F_{rem} \subseteq F \setminus (\bigcup_{(f,p) \in UR_{F_{add}}} \{p\})$.*
- *CF_{add} is a set of configurations that must be valid in the updated feature model. Each c in CF_{add} must not be a product of fm (i.e., $\neg val(fm, c)$); moreover, it cannot contain features removed in F_{rem}, but can contain features added in F_{add} (i.e., $c \subseteq (F \cup F_{add}) \setminus F_{rem}$).*
- *CF_{rem} is a set of configurations defined over the set $c \subseteq (F \cup F_{add}) \setminus F_{rem}$ that must not be valid in the updated feature model. The only constraint on CF_{rem} (and on CF_{add}) is that it is not possible to remove configurations that are also added in CF_{add} (i.e., $CF_{add} \cap CF_{rem} = \emptyset$).*

Note that CF_{add} and CF_{rem} usually don't have to be explicitly enumerated by the designer, but they could be derived by failing tests that are, respectively, wrongly rejected and wrongly accepted. Moreover, the designer could specify them in a compact way by providing logic formulae that symbolically identify sets of configurations that must be accepted and rejected.

Modifying the feature model such that it satisfies the update request is a challenging task. Note that, in general, there could be no fm' that exactly adds and removes the specified configurations and features, unless general constraints are used. However, the usage of these constraints is usually not recommended, as they make the model less readable and less maintainable [11].

[1] In addition to the parental relations, it is possible to add *extra-constraints*, i.e., cross-tree relations that specify incompatibility between features; in this work, we do not consider extra-constraints.

[2] Note that all feature models must contain a `root` feature that must be present in all the configurations. In this paper, we assume that the root is included in all configurations but, for presentation purposes, we never report it and do not consider it in the set of features F.

We now want to obtain a formula that accepts and rejects configurations as the updated feature model (i.e., the one obtained after the update request) should do according to an update request *UR*. First of all, in order to specify the semantics of F_{rem}, we exploit the approach used in [20] to remove features from feature models.

Definition 4 (F_{rem} **semantics**). *Given a feature model fm and a set of features* $F_{rem} = \{f_1, \ldots, f_h\}$ *to remove, we recursively define* ϕ_i *as follows:*

$$\phi_i = \begin{cases} \textsc{bof}(fm) & i = 0 \\ \phi_{i-1}[f_i \leftarrow \texttt{true}] \vee \phi_{i-1}[f_i \leftarrow \texttt{false}] & 0 < i \leq h \end{cases}$$

The final formula $\phi_{rem} = \phi_h$ *has the same products of fm, except for the features in* F_{rem}.

We can now build the *target* formula that will be used as oracle to guide the proposed updating process.

Definition 5 (**Target**). *The target t is a propositional formula whose models exactly correspond to the products of the desired updated feature model. Let fm be the initial feature model and* $UR = \{UR_{F_{add}}, F_{rem}, CF_{add}, CF_{rem}\}$ *be the update request; the target is defined as:*

$$t = \left(\phi_{rem} \vee \bigvee_{c \in CF_{add}} \textsc{bof}(c)\right) \wedge \bigwedge_{(f,p) \in UR_{F_{add}}} f \rightarrow p \wedge \bigwedge_{f \in F_{rem}} \neg f \wedge \bigwedge_{c \in CF_{rem}} \neg \textsc{bof}(c)$$

The target correctly rejects all the configurations of CF_{rem} and those containing a removed feature; the accepted configurations are those in CF_{add}, and those of ϕ_{rem} (i.e., the original feature model without the removed features) possibly extended with the added features of F_{add} (when the constraints of $UR_{F_{add}}$ are satisfied).

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

We can use the target to evaluate whether a feature model fm' captures the desired change requirements (i.e., $\models t = \textsc{bof}(fm', F_U)$). 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 of the *difference* between a feature model fm^x (either the initial one *fm* or a modified one fm') and the target as follows.

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

$$FR(fm^x, t) = \frac{|AllConfs(\textsc{bof}(fm^x, F_U) \neq t)|}{2^{|F_U|}}$$

where AllConfs(φ) *gets all the configurations satisfying* φ[3].

If the fault ratio is equal to 0, it means that fm^x accepts as products the same configurations that are logical models of *t*; otherwise, there are some configurations that are wrongly evaluated by fm^x,

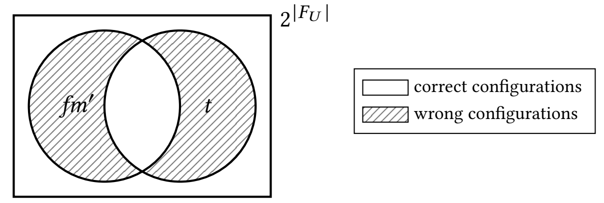
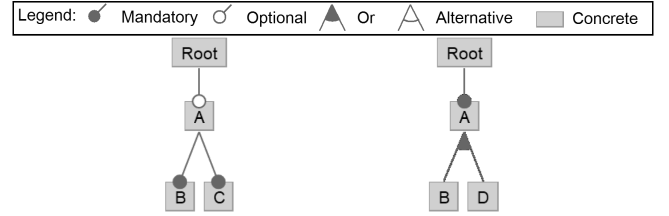


Figure 1: Faults



(a) Original feature model (b) Updated feature model

Figure 2: Example of feature model update

as shown in Fig. 1: fm^x could wrongly accept some configurations and/or wrongly refuse some others.

Example 1. Let's consider the feature model shown in Fig. 2a, made of three features $F = \{A, B, C\}$ (as said before, we do not report the root feature in *F*). Feature A is optional, whereas B and C are mandatory (i.e., they must be selected if A is present). The only products of *fm* are $\{\neg A, \neg B, \neg C\}$ and $\{A, B, C\}$. Let's now suppose to have an update request defined by the following sets:

- $UR_{F_{add}} = \{(D, A)\}$ requires to add *D* as child of *A*; the effect of this update is that, for each product containing *A*, a new product containing *D* is added, i.e., the product $\{A, B, C, D\}$ is added;
- $F_{rem} = \{C\}$ requires to remove *C* from all the products, i.e., products $\{A, B, C, \neg D\}$ and $\{A, B, C, D\}$ become $\{A, B, \neg D\}$ and $\{A, B, D\}$;
- $CF_{add} = \{\{A, \neg B, D\}\}$ requires to add the specified product;
- $CF_{rem} = \{\{\neg A, \neg B, \neg C\}\}$ requires to remove the specified product.

A possible updated model that satisfies all the required changes is shown in Fig. 2b, in which A is mandatory, C has been removed, D has been added, and B and D are in OR.

The fault ratio of the initial feature model w.r.t. to the target (that corresponds to the final feature model) is $\frac{5}{16}$; for example, $\{A, \neg B, \neg C, D\}$ is a configuration that is wrongly rejected by *fm*, and $\{A, B, C, \neg D\}$ a configuration that is wrongly accepted as product.

4 EVOLUTIONARY UPDATING PROCESS

In this section, we introduce the process (depicted in Fig. 3) we propose to update an initial feature model *fm*, given an update request $UR = \{UR_{F_{add}}, F_{rem}, CF_{add}, CF_{rem}\}$.

As initial step, we generate the target *t* as described in Def. 5.

Then, we start the updating process. First of all, we modify *fm* in order to try to achieve the change requirements of $UR_{F_{add}}$ and F_{rem}. For each $(f, p) \in UR_{F_{add}}$, we add f as child of p as follows: if p is

[3] In our approach, we represent formulas as BDDs in JavaBDD that implements *AllConfs* by means of method `AllSat`. JavaBDD also provides the method `satCount` that directly computes the cardinality of the set without enumerating all the models.

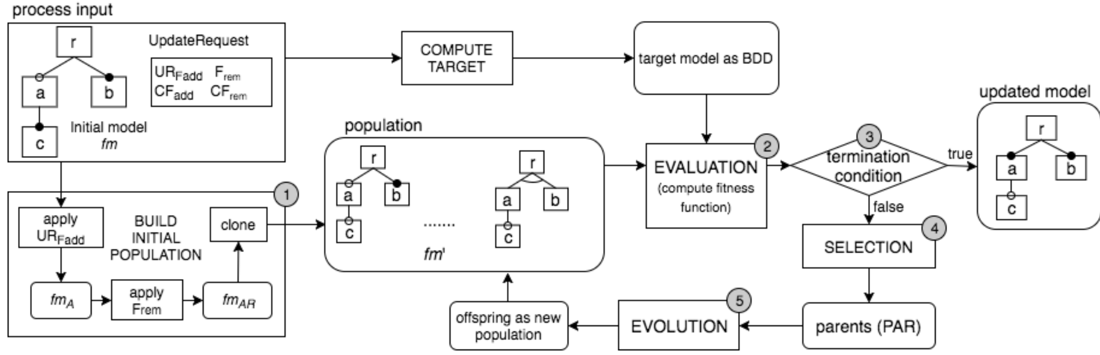


Figure 3: Proposed evolutionary approach

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. 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 (if any) which inherit the same relation that f had with its parent. 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 CF_{add} and/or remove configurations CF_{rem}. Moreover, the two previous transformations do not guarantee to precisely implement the required change requests $UR_{F_{add}}$ and F_{rem}, and they could introduce some wrong configurations (either wrongly accepted or rejected).

Starting from fm_{AR}, we apply an evolutionary updating approach to try to obtain an updated feature model equivalent to the target. The process follows these classical steps of evolutionary algorithms [8]:

(1) **Initial population**: at the beginning, a population P of candidate solutions is created.
(2) **Iteration**: the following steps are repeatedly executed:
  (a) **Evaluation**: each member of the population P is evaluated using a given *fitness 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 are selected as parents PAR of the next generation.
  (d) **Evolution**: parents PAR are mutated to obtain the *offspring* to be used as population in the next iteration.

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.

4.1 Initial population

As initial population, we generate the set P by cloning fm_{AR} M times (step 1 in Fig. 3). The feature model fm_{AR} is obtained by the application of the first two updating steps (i.e., those for $UR_{F_{add}}$ and F_{rem}). In this way, if fm_{AR} is already correct, it will be returned as final model in the *termination condition* phase (see Sect. 4.3).

4.2 Evaluation

As first step of each iteration (step 2 in Fig. 3), 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. In our context, the fitness function is defined in terms of fault ratio (see Def. 6), i.e.,

$$fitness_t(fm') = 1 - FR(fm', t)$$

Therefore, the quality of a candidate is given by the percentage of configurations that it evaluates correctly.

4.3 Termination condition

In this step (step 3 in Fig. 3), 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 $fitness_t(fm') \geq Th_f$; $Th_f = 1$ means that we want to obtain a completely correct model;
- 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[4].

4.4 Selection

In the *selection* step (step 4 in Fig. 3), starting from population P, a multiset of *parents PAR* of size p is built, being p is 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.

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

Roulette wheel: for each member of the population, a section of an imaginary roulette wheel is allocated; a section is proportional to the member fitness, such that the fittest candidate has the biggest slice and the weakest candidate has the smallest one. The wheel is then spun p times to select the candidates to add to *PAR*. 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 absolute fitness, i.e., members of the population receive an area of the wheel proportional to their ranking. 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).

4.5 Evolution

In the *evolution* step (step 5 in Fig. 3), 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 [10] that states 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 [1], we have proposed some mutation operators for feature models, divided in *feature-based* and *constraint-based* operators that are a subset of edit operations identified in [7]. We here consider only the *feature-based* ones, as we only consider basic feature models (i.e., without cross-tree constraints [11]), but the process could be extended also to arbitrary constraints. In particular, we use eight of the feature-based mutation operators proposed in [1], and introduce two new ones (`OrToAndOpt` and `AlToAndOpt`). The operators are described in Table 1.

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 maximum 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.

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

Table 1: Mutation operators

	SPL	model size input	model size target	*UR* size $\lvert UR_{F_{add}}\rvert$	$\lvert F_{rem}\rvert$	$\lvert CF_{add}\rvert$	$\lvert CF_{rem}\rvert$
$\text{BENCH}_{\text{SPLOT}}$	`MobileMedia` (V5, V6)	15	18	4	1	0	64
	`MobileMedia` (V7, V8)	23	26	19	16	32	1.53×10^5
	`HelpSystem` (V1, V2)	25	26	1	0	672	2016
	`SmartHome` (V2.0, V2.2)	39	60	35	14	1.25×10^9	3.3×10^{11}
	`ERP_SPL` (V1, V2)	43	58	15	0	0	1.51×10^7
$\text{BENCH}_{\text{MUT}}$	`Example`	4	4	0	0	0.41 (0-2)	0.68 (0-2)
	`Register`	11	11	0	0	11.85 (0-40)	62.28 (0-210)
	`Graph`	6	6	0	0	12.71 (0-28)	0 (0-0)
	`Aircraft`	13	13	0	0	196.86 (0-315)	53.53 (0-365)
	`Connector`	20	20	0	0	8.23 (0-18)	26.02 (0-336)

Table 2: Benchmark properties

5 EXPERIMENTS

The process[5] is developed in Java and uses JavaBDD for BDD manipulation. To represent and mutate feature models we use FeatureIDE [14], and as evolutionary framework we use Watchmaker[6].

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.

The first benchmark set $\text{BENCH}_{\text{SPLOT}}$ is constituted by feature models published in literature. We identified in the SPLOT repository[7] four SPLs that evolved over time and, therefore, are described by different versions of the same feature model: a program to manipulate multimedia on mobile devices[8] (`MobileMedia`), a cyber-physical system with multiple sensors (`HelpSystem`), a set of smart house components (`SmartHome`), and an Enterprise Resource Planner (`ERP_SPL`). For each SPL, we identified couples (fm_i, fm_t) of their feature models (two couples for `MobileMedia`, and one couple for the other three SPLs): the latest version was considered as target model[9] fm_t, and the oldest one as the initial model fm_i we want to update. Table 2 reports the considered versions.

The second benchmark set $\text{BENCH}_{\text{MUT}}$ contains, as target models, five feature models developed for five SPLs:

- `Example`: the example used in Fig. 2b;
- `Register`: a register of supermarkets, adapted from [18];

[5] The code is available at https://github.com/fmselab/eafmupdate.
[6] https://watchmaker.uncommons.org/

[7] http://52.32.1.180:8080/SPLOT/feature_model_repository.html
[8] http://mobilemedia.cvs.sf.net/
[9] Note that, in the real usage of our approach, we don't have a target feature model, but an update request *UR* from which we generate the target as propositional formula.

- `Graph`: a graph library;
- `Aircraft`: the configurations of the wing, the engine and the materials of airplane models;
- `Connector`: IP connection configurations.

For these five SPLs, only one model has been proposed in literature. Therefore, we had to automatically generate other versions to be used as input models. To build them, we rely on the assumption of our process that the evolution of a feature model can be obtained using few mutation operators (see Sect. 4.5). 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. In this way, $\text{BENCH}_{\text{MUT}}$ contains 500 couples.

In the normal usage of the approach, the user should specify the update request to give as input to the evolutionary process. For the experiments, we automatically generated the update request UR from the initial feature model fm_i and the target feature model fm_t, by computing the differences of their features and retrieving, using their BDD representation, the configurations that are differently evaluated; configurations that are added and removed by $UR_{F_{add}}$ and CF_{rem} are not in F_{add} and F_{rem}. Deriving the update requests from real evolutions of feature models (as for $\text{BENCH}_{\text{SPLOT}}$) has the advantage that we are experimenting the approach on real change requirements; on the other hand, also the update requests of $\text{BENCH}_{\text{MUT}}$ should represent plausible change requirements, assuming that the mutation operators mimic real edits done by designers on feature models.

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 $\text{BENCH}_{\text{MUT}}$, the reported values are aggregated by SPL; since we did not add or remove features for producing the input models of $\text{BENCH}_{\text{MUT}}$, the size of all the input models is the same of that of the target model and so $UR_{F_{add}}$ and F_{rem} are empty; for CF_{add} and CF_{rem}, instead, we report the average, minimum, and maximum number of configurations.

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 condition as follows: Th_f to 1, Th_{NI} to 15, Th_i to 25, and no time limit. The parameter H used to determine the maximum population size M (as defined in Sect. 4) 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 10 runs.

RQ1: *Which is the selection strategy of the evolutionary process that achieves the best results?*

The most critical parameter of the proposed process is the selection strategy, and we here evaluate which strategy obtains the best results. We run the updating process over all the models using truncation in three versions (with $K = 2\%, 5\%, 10\%$), roulette wheel, and rank. As baseline, we add a *random* selection strategy that randomly selects p members of the population. Table 3 reports the results in terms of FR reduction (defined as $1 - FR(fm_f, t)/FR(fm_i, t)$, being fm_i and fm_f the initial and the final feature models) and process time. The values are the averages among all the benchmarks. All the three truncation strategies perform similarly and better than the other strategies both in terms of *FR* reduction and execution time. Also the rank strategy performs very well. The random and the roulette wheel selection strategies underperform both in terms of time and *FR* reduction.

Selection strategy	avg time (s)	*FR* reduction (%)
truncation K=2%	13.3	83.9%
truncation K=5%	13.8	84.4%
truncation K=10%	14.0	84.2%
roulette wheel	16.8	69.1%
rank	14.3	82.2%
random	26.6	71.4%

Table 3: Selection strategy comparison

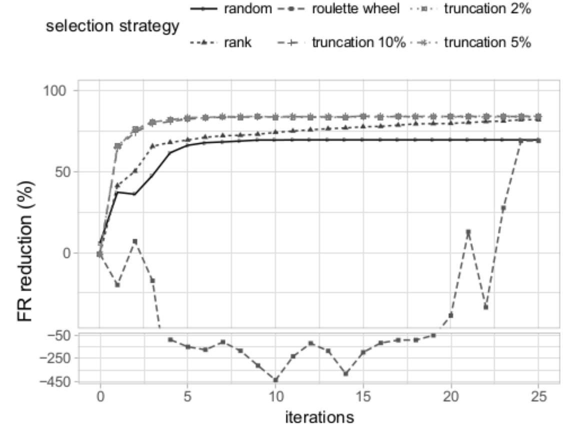


Figure 4: Reduction of *FR* during the evolutionary process

RQ2: *How does the fault ratio decrease during the process?*

We now observe how the different selection strategies affect the evolution of the *FR*. To perform this observation, for each feature model in the population, we keep track of the sequence of all its ancestors (i.e., the models from which it has been obtained by mutation in the previous iterations). For the model returned at the end, we compute the *FR* in the sequence of its ancestors (called *best sequence*). Fig. 4 reports the average *FR* reduction of the best sequences over all the benchmarks. We can notice that the three truncation strategies progress similarly: at the beginning they are very efficient in reducing *FR*, and after few mutations (around 5) *FR* has been almost completely reduced. After that, the progress is very slow. The rank selection, instead, has a slower progress at the beginning but, after a while, it converges towards the best values. This is a strength of the rank strategy, that tries to avoid early convergence and reduces the risk of being stuck into local minimum after few iterations. The roulette wheel risks, especially at the beginning, to increase the *FR* instead of reducing it; at 10 iterations, it reaches an *FR* reduction of -450%. In the random strategy, the best model is obtained after few iterations and the algorithm is no longer able to find a better model. These two latter strategies are unable to reduce *FR* as the others.

RQ3: *How are the performances of the approach affected by the initial model and the update request?*

	SPL	time (s)	initial FR (%)	final FR (%)	FR reduction	# iterations	$\equiv fm_t$ (%)
$\mathsf{BENCH_{SPLOT}}$	MobileMedia 5-6	2.72	0.031	0.007	76.7%	19	0
	MobileMedia 7-8	7.28	0.098	0.0009	99.1%	24.5	0
	HelpSystem	5.63	0.008	0.002	74.4%	17	0
	SmartHome	58.1	2.33×10^{-8}	2.33×10^{-8}	0%	15	0
	ERP_SPL	62.0	4.60×10^{-8}	1.79×10^{-9}	96.1%	25	0
$\mathsf{BENCH_{MUT}}$	Example	0.06	13.63	0	100%	2.07	100
	Register	0.46	7.24	0.175	97.6%	5.39	96
	Graph	0.07	39.7	0	100%	1.74	100
	Aircraft	0.45	6.11	0	100%	4.33	100
	Connector	0.96	0.006	3.81×10^{-5}	99.4%	4.85	99

Table 4: Performance of the evolutionary process with the most promising selection strategy (truncation K=5%)

We now perform a more detailed analysis to assess how the process performs over the different benchmarks (having different sizes and different update requests); in order to do this, we select the selection strategy that proved to be the best in RQ1 (i.e., truncation K=5%). Table 4 reports, for each benchmark SPL, the process time, the evolution of *FR* (initial, final, and reduction), and the number of performed iterations. For each SPL of $\mathsf{BENCH_{MUT}}$, the table reports the averages among its 100 input models.

We observe that the size of the initial model and the numbers of change requirements in *UR* affect the *FR* reduction and the generation time. Small models as those in $\mathsf{BENCH_{MUT}}$ are almost all completely updated in less than 1 second; instead, bigger models as those in $\mathsf{BENCH_{SPLOT}}$ are partially updated: for two of them we get a reduction of around 75%, for two others the reduction is more than 95%, while for SmartHome we are not able to update the feature model at all. We manually inspected the initial model fm_i and the target model fm_t of SmartHome and we observed that their difference is so high that our mutation operators (used to produce the offspring in the evolution phase) are not able to modify the initial model V2.0 in a way that converges towards the target model V2.2; as discussed in Sect. 4.5, our approach assumes that a feature model is updated using simple syntactic variations as those described by our mutation operators, and, for SmartHome, this is not the case.

From Table 4, we also observe that small models in $\mathsf{BENCH_{MUT}}$ need few iterations to reach the final model (no more than 6), while big models in $\mathsf{BENCH_{SPLOT}}$ always need at least 15 iterations. SmartHome terminates after 15 iterations because of the termination condition on the fitness improvement (Th_{NI} is set to 15); also MobileMedia 5-6 and HelpSystem terminate because of Th_{NI}, but they are able to improve in the initial iterations; ERP_SPL, instead, terminates after 25 iterations because of the termination condition on the number of iterations (Th_i is set to 25); also MobileMedia 7-8 (in most of its runs) terminates because of Th_i.

RQ4: *Does the approach mimic the edits done by designers?*

The aim of the proposed approach is to build a feature model that satisfies the update request, and, in order to define the quality of a solution, we introduced the fault ratio *FR* measuring the percentage of configurations evaluated correctly; if $FR = 0$, the feature model is correct. However, the correct model we obtain could be very different from the one a user would obtain by updating the feature model manually; since we assume that users tend to write readable models, a different final model would mean that we reduced the model *readability*. Since in our benchmarks we have the correct feature model fm_t (used to compute the update request) written by a designer of the SPL, we can check whether a final model fm_f is also syntactically equal to fm_f. Table 4 reports, for each benchmark, the percentage of times that the final model fm_f is syntactically equal ($\equiv$) to fm_t. We can see that all the models of $\mathsf{BENCH_{MUT}}$ that are completely updated by the process are also identical to the target model fm_t (the semantically correct final models of Register and Connector are exactly 96% and 99%, as the identical ones). For $\mathsf{BENCH_{SPLOT}}$, instead, we never obtain a model identical to fm_t, as the process never completely updates the initial model.

Note that we employ a quite draconian approach to evaluate the readability of the final model fm_f by checking whether it is identical to the target model fm_t; however, it could be that fm_f, although not identical, is not too different from fm_t. A more precise approach could use some distance measure such as the *edit distance* [15] in order to assess the readability of the produced solutions.

6 THREATS TO VALIDITY

We discuss the threats to the validity of our results along two dimensions, *external* and *internal* validity [21].

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 five couples of models showing the evolution of real SPLs taken from the SPLOT repository; moreover, in order to enlarge the set of evaluated models, we generated 500 input models by randomly mutating other 5 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.

Regarding internal validity, a threat could be that the obtained results 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 (by varying only the selection strategy), 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.

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 [1]; in [2], 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 [9]: 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 [3], 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 [3] is given by the implementation constraints, while here the target comes from update requests. Then, the aim of [3] is only to remove faults from the model, while here we also support the evolution of the model. Finally, the approach in [3] 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 [13] 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 [13] 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 [12] 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 and products to add/remove from a feature model, through a sequence of mutations, tries to obtain a feature model that exactly captures the specified requirements.

We evaluated the approach on feature models of ten SPLs and we found that it is indeed able to apply the specified change requirements, although the update could be partial when the model is particularly big.

Some models cannot be completely updated by the proposed approach; as future work, we plan to study whether other mutation operators (e.g., moving a feature in a different part of the feature model [1]), mimicking the edits done by designers, would allow to obtain a better *FR* reduction. We did not consider any measure of model readability in the approach; as future work, we plan to assess whether the proposed approach preserves the readability and, eventually, integrate a measure of readability also in the fitness function. The goal would be avoiding the generation of correct models that, however, are not readable and maintainable. Moreover, since the computation of the fitness function is the most time consuming operation of the process, we plan to parallelize it over the population members; however, this would need to modify JavaBDD that currently does not support parallelization.

REFERENCES

[1] P. Arcaini, A. Gargantini, and P. Vavassori. Generating tests for detecting faults in feature models. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10, April 2015.

[2] P. Arcaini, A. Gargantini, and P. Vavassori. Automatic detection and removal of conformance faults in feature models. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 102–112, April 2016.

[3] P. Arcaini, A. Gargantini, and P. Vavassori. Automated repairing of variability models. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A*, SPLC '17, pages 9–18, New York, NY, USA, 2017. ACM.

[4] D. Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Conference on Software Product Lines*, SPLC'05, pages 7–20, Berlin, Heidelberg, 2005. Springer-Verlag.

[5] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[6] D. Benavides, P. Trinidad, and 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*, pages 491–503, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[7] J. Bürdek, T. Kehrer, M. Lochau, D. Reuling, U. Kelter, and A. Schürr. Reasoning about product-line evolution using complex feature model differences. *Automated Software Engineering*, 23(4):687–733, Dec 2016.

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

[9] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and 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, pages 1245–1248, Piscataway, NJ, USA, 2013. IEEE Press.

[10] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, Sept. 2011.

[11] A. Knüppel, T. Thüm, S. Mennicke, J. Meinicke, and 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, pages 291–302, New York, NY, USA, 2017. ACM.

[12] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 3–13. IEEE, 2012.

[13] R. E. Lopez-Herrejon, L. Linsbauer, J. A. Galindo, J. A. Parejo, D. Benavides, S. Segura, and A. Egyed. An assessment of search-based techniques for reverse engineering feature models. *Journal of Systems and Software*, 103:353–369, 2015.

[14] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017.

[15] M. Pawlik and N. Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157–173, Mar. 2016.

[16] A. Pleuss, G. Botterweck, D. Dhungana, A. Polzer, and S. Kowalewski. Model-driven support for product line evolution on feature level. *Journal of Systems and Software*, 85(10):2261–2274, 2012. Automated Software Evolution.

[17] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456 – 479, 2007.

[18] D. Shimbara and 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*, pages 221–237, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[19] T. Thum, D. Batory, and C. Kastner. Reasoning about edits to feature models. In *Proceedings of the 31st International Conference on Software Engineering*, pages 254–264. IEEE Computer Society, 2009. 00267.

[20] T. Thum, C. Kastner, S. Erdweg, and N. Siegmund. Abstract features in feature modeling. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 191–200. IEEE, 2011. 00039.

[21] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln. *Experimentation in Software Engineering*. Springer, 2012.