Automatic Detection and Removal of Conformance Faults in Feature Models

Paolo Arcaini
Charles University in Prague
Faculty of Mathematics and Physics, Czech Republic
Email: arcaini@d3s.mff.cuni.cz

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

Paolo Vavassori
DIGIP
University of Bergamo, Italy
Email: paolo.vavassori@unibg.it

***Abstract*—Building a feature model for an existing SPL can improve the automatic analysis of the SPL and reduce the effort in maintenance. However, developing a feature model can be error prone, and checking that it correctly identifies each actual product of the SPL may be unfeasible due to the huge number of possible configurations. We apply mutation analysis and propose a method to detect and remove *conformance* faults by selecting special configurations that distinguish a feature model from its mutants. We propose a technique that, by iterating this process, is able to repair a faulty model. We devise several variations of a simple hill climbing algorithm for automatic fault removal and we compare them by a series of experiments on three different sets of feature models. We find that our technique is able to improve the conformance of around 90% of the models and find the correct model in around 40% of the cases.**

I. Introduction

Feature models (FMs) allow designers to specify families of products, generally called Software Product Lines (SPLs), in a simple way. A feature model lists the features in a product line together with their possible values and 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 product line consistency, automatic product configuration, interaction testing among features on the products, and similar actions [1].

Building the correct feature model for an SPL is not an easy task: while identifying the features may be easy and performed in a semiautomatic way [2], modeling the relationships among features and their constraints can be very complex and time consuming [3]. There exist numerous attempts to help the designer to reverse engineer a feature model from an SPL [4]. These techniques generally require the set of valid products, normally given as a *list*, as in [5], or symbolically as a *propositional formula* specifying dependencies among features, as in [6]. Some also require a starting feature model [7], while others promise to synthesize a model from scratch [8].

In many cases, though, modeling the entire set of valid products may be impossible or too costly. Consider, for example, the Linux kernel: with its thousands of features, the list of valid configurations is intractable. In many cases, also a representation of the constraints among the features can be difficult to obtain. In general, discovering if a configuration is valid or not may require a great effort in terms of time (for example, if a compilation and build are necessary) and in terms of human effort (if checking whether a configuration is correct or not requires some human intervention). In these cases, traditional reverse engineering techniques are not efficient.

In this paper, we assume to have an SPL and already a feature model describing it. We assume that the model correctly identifies the features of the SPL (although some of them may be not necessary), but the relations among features may be wrong. Moreover, we assume that static anomalies like dead features, redundant constraints and so on [1] have already been found and removed (if any) with, for example, FeatureIDE [9]. However, the designer could be not sure that the feature model exactly captures the SPL: correct configurations over the SPL may be judged wrong by the feature model or the other way around. We call these discrepancies *conformance faults* (that are different from feature model anomalies, like dead features and so on, that generally indicate non-minimal models). We want to automatically discover the conformance faults and remove them in order to obtain a model which is *closer* to the SPL. Moreover, we assume that checking whether a configuration is a valid product or not is costly and we want to keep the number of checks limited.

In order to do this, we apply a mutation-based approach that, through a sequence of mutations, obtains a feature model having a *better* quality. Quality is defined in terms of number of configurations that are correctly evaluated. The approach consists in generating some mutants of the starting model, computing configurations that permit to distinguish the model from its mutants, checking the quality of the mutants and the original model over the generated configurations, and choosing the *best* model. We propose several versions of the previous general process which differ in the number (and kind) of considered mutants, in the number of generated configurations, in the order in which the phases are executed, etc.

In Sect. II we give background on feature models, mutation of them, and generation of configurations able to distinguish a model from a mutant. Sect. III presents the notion of conformance fault and conformance index. Sect. IV presents our approach for repairing a possible faulty feature model, and gives different versions of the general approach with a preliminary evaluation. A more throughout evaluation is done in Sect. V. Sect. VI identifies possible threats to the validity of the proposed approach. Finally, Sect. VII reviews some related

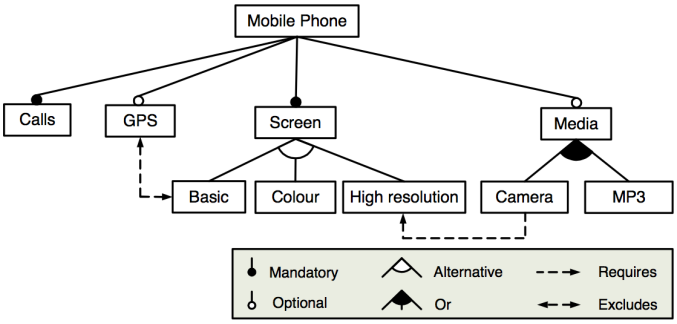


Fig. 1: Example of a Feature Model

work and Sect. VIII concludes the paper.

II. Background

A. Feature models

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 one of the following types (each having a suitable graphical notation as shown in Fig. 1):

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

In addition to the parental relations, it is possible to add *extra-constraints*, i.e., cross-tree relations that specify incompatibility between features:

- A *requires* B – The selection of feature A in a product implies the selection of feature B.
- A *excludes* B – A and B cannot be part of the same product.
- It is also possible to specify a constraint in *general* form through a propositional formula (using the usual Boolean operators $\vee, \wedge, \rightarrow, \neg, \dots$) representing the features as propositional variables. Not all the frameworks support the general form of constraints. Allowing general constraints guarantees logical completeness [10].

Feature models can be visually represented by means of feature diagrams. In order to present the visual notation commonly adopted for feature modeling, Fig. 1 depicts a simplified example model presented in [1] and inspired by the mobile phone industry. The example also shows how a model can be used to specify a product family, i.e., to determine the features that will be supported (selected) in a particular phone configuration of the considered family. According to the model, all mobile phones must include support for calls, and must display information in either a basic, color or high resolution screen. Furthermore, the software for mobile phones may optionally include support for GPS and multimedia devices such as the camera, the MP3 player, or both of them. An extra-constraint (*excludes*) specifies that the GPS and the basic screen are incompatible, and another constraint asserts that the camera *requires* a high resolution screen.

Extensions to the basic feature model notation have been proposed in literature, e.g., for specifying the cardinality of the features and/or additional types of information. However, in this paper we only consider basic feature models with general form constraints.

Several languages/tools for specifying/analyzing feature models are currently available, some of them already mature enough to be part of a software production IDE. In this work, FeatureIDE [9] has been used to design, import, analyze, mutate, and validate the models used in the evaluation section.

Feature Model semantics: Feature models semantics can be rather simply expressed by using propositional logic as already done in [11], [1]. Every feature becomes a propositional letter, and every relation among features becomes a propositional formula modeling the constraints about them.

B. Configurations and Validity

Definition 1: A *configuration* of a feature model fm is a subset of the features in fm that must include the root.

If fm has n features (including the root), there are 2^{n-1} possible configurations, which, however, are not all valid. For instance, a configuration may miss a mandatory feature. In the following, let C_{fm} be the set of configurations of a feature model fm.

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

C. Mutating feature models

Our idea is that conformance faults can be discovered through mutation analysis. In [12], we have proposed some mutation operators for feature models, divided in *feature-based* and *constraint-based* operators. We here consider all of them and introduce new ones.

Feature-based mutation operators are:

- **AlToOr**: an Alternative is changed to an Or;
- **AlToAnd**: an Alternative is changed to an And;
- **OrToAl**: an Or is changed to an Alternative;
- **OrToAnd**: an Or is changed to an And;
- **AndToOr**: an And is changed to an Or;
- **AndToAl**: an And is changed to an Alternative;
- **ManToOpt**: a mandatory relation is changed to optional;
- **OptToMan**: an optional relation is changed to mandatory;

(a) Original model fm (b) Mutant fm'

Fig. 2: A model and a **ManToOpt** mutant

- **MF**: a feature f is removed and it is replaced by its sub-features which inherit the same relation the removed feature had with its parent. f is replaced by false in any constraint containing it;
- **MoveF**: a feature f is moved (with its descendants) as child of another feature (not belonging to its descendants) in the feature model.

When a relation is changed to And, all the children in the relation are set to mandatory (if the parent is selected, all the children must be selected). We never remove the root, otherwise we would obtain a void model.

Constraint-based mutation operators are:

- **MC**: an extra-constraint is removed;
- **ReqToExcl**: a *requires* constraint is transformed into an *excludes* constraint;
- **ExclToReq**: an *excludes* constraint is transformed into a *requires* constraint;
- A general constraint is modified by inserting a new feature, changing a logical operator (e.g., *and* becomes *or*), or removing part of it. These mutation operators are borrowed from the classical logical mutation operators of Boolean expressions in general form [13].

Example 1: Fig. 2 shows a model and one of its possible faulty implementations, namely the model obtained by applying the **ManToOpt** operator to the relation between the root node a and its child b.

Definition 3 (Mutants set): Given a feature model fm, we identify with $\texttt{mut}(fm)$ the mutants obtained by applying the mutation function mut.

Function mut applies (a subset of) the operators defined previously, once (first order mutation) or several times in sequence (higher order mutation).

Note that mutation operators can produce *equivalent mutants*, i.e., mutants representing the same set of products of the original feature model.

D. Distinguishing configurations

Definition 4 (Distinguishing configuration): We say that a configuration c distinguishes a feature model fm from a mutant fm', if c is valid in fm and not in fm' (i.e., $val(fm, c)$ and $\neg val(fm', c)$) or vice versa. We call this a *distinguishing* configuration.

A distinguishing configuration is able to find the difference between fm and fm'. It could be either a product or an invalid configuration for the original model, but it is a product for only either fm or fm', never for both. Note that equivalent mutants do not have distinguishing configurations. In the following, we identify with $DCs(fm, fm')$ the set of distinguishing configurations between fm and fm'.

Example 2: Consider the model in Fig. 2a and its mutant in Fig. 2b. The configuration $\{a, c\}$ is valid in fm' but it is invalid in fm: therefore, it is a distinguishing configuration. The configuration $\{a, b\}$, instead, is valid in both feature models fm and fm' and so it is not a distinguishing configuration.

In [12], we described an approach for generating compact distinguishing configuration sets based on the computation (through a SAT/SMT solver) of a model for the exclusive or between fm and fm'. In this work, we generate distinguishing configurations by exploiting a feature of FeatureIDE [9] that directly calls Sat4j.

III. Conformance faults and conformance index

In this section, we show how we compare a feature model with the SPL it is supposed to represent. We do this by comparing the validity evaluations given to some specific configurations by the feature model and the SPL (acting as oracle). Since we suppose to have only one software product line under representation, we directly use *SPL* for indicating it, whereas we give different names to the different feature models allegedly describing SPL.

We assume that there is an oracle able to tell us whether a given configuration is a product in the SPL or not.

Definition 5 (Oracle): An *oracle* o, given a configuration c, gives the validity value of c over the SPL. An oracle is defined as follows

$$o\colon C \rightarrow Boolean$$

A feature model may wrongly assess the validity of some configurations. In this case, we say that it is *killed* by the configuration.

Definition 6 (Killed feature model): A feature model fm is *killed* by a configuration c if the validity value of c over fm is different from the expected value given by the oracle o, i.e.,

$$killed(fm, c, o) \quad \text{iff} \quad o(c) \neq val(fm, c)$$

Definition 7 (Conformance fault): A feature model fm for SPL contains a *conformance fault* if there exists a configuration c that is a product in fm but not in the SPL or vice versa (i.e., fm is killed by c). Formally,

$$\exists c \in C\colon killed(fm, c, o)$$

A conformance fault is a discrepancy between the feature model and the SPL under representation represented by the oracle o. Such faults are due to wrong relations among features or because of wrong constraints.

We give an index of the *quality* of the feature model as conformance w.r.t. the oracle.

Definition 8 (Conformance index): Given a feature model fm and an oracle o, we define the *conformance index* ($confIndex$) of fm as the percentage of configurations of the oracle (C_o) whose validity is correctly judged by fm:

$$confIndex(fm, o) = \frac{|\{c \in C_o\colon \neg killed(fm, c, o)\}|}{|C_o|}$$

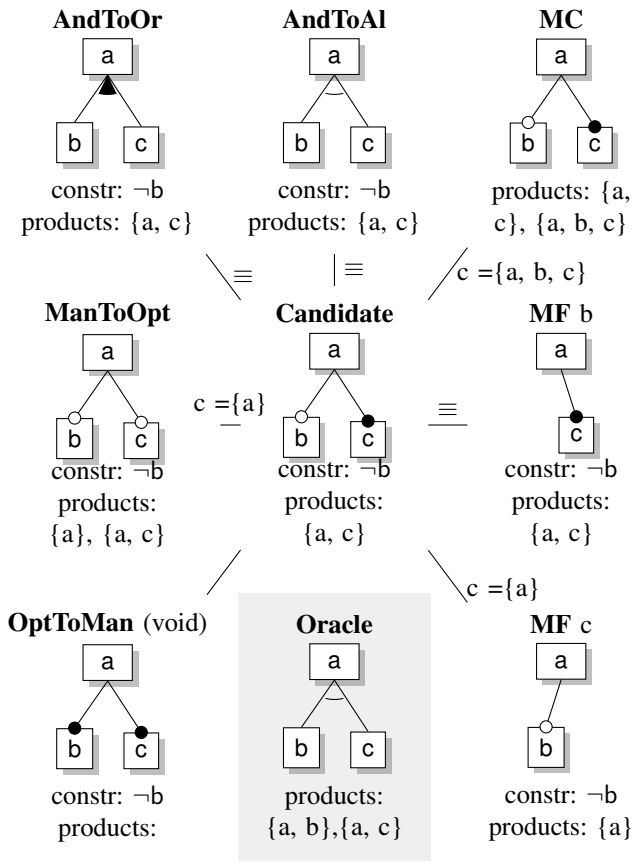


Fig. 3: Mutation-conformance

A correct model will never be killed, and therefore has 100% conformance index: in that case, we say that it is *conformant* to the oracle.

Since computing the conformance can be expensive and is sometimes impossible, as all the configurations of the oracle are required, in the following we give a weaker notion of conformance that will be used in our approach.

Definition 9 (Mutation-conformance): Given a feature model fm and a function `mut` producing mutants, fm is *mutation-conformant* ($mutConf$) if every non-equivalent mutant $fm' \in \texttt{mut}(fm)$ is killed by a configuration which confirms fm instead.

$$mutConf(fm, o) = \forall fm' \in \texttt{mut}(fm): \left(\begin{array}{ll} fm \not\equiv fm' & \Longrightarrow \\ \exists c \in C_{fm}: & \neg killed(fm, c, o) \wedge \\ & killed(fm', c, o) \end{array} \right)$$

Mutation-conformance does not guarantee that fm is correct, and neither that it has a conformance index better than its mutants. However, it guarantees that the mutations introduced in the mutants fm' are conformance faults that are not present in fm (that, however, may contain other faults).

Example 3: Fig. 3 shows an example in which mutation-conformance does not imply to have a model equivalent to the oracle. The candidate is in the middle, while its mutations obtained by applying the mutation operators `mut` ={**OptToMan**, **MF**, **ManToOpt**, **AndToAl**, **AndToOr**, **MC**} are displayed around it. The candidate is mutation-conformant w.r.t. the mutation function `mut` because each mutant is either equivalent or killed by at least a configuration c (shown on the line connecting the candidate with the mutant). However, the candidate is not correct w.r.t. the oracle in grey in the bottom of the figure and its conformance index is only ¾.

A. *Using distinguishing configurations to prove* mutConf

We have already observed that the set C_{fm} may be too big to compute, but we can exploit distinguish configurations to operatively prove the weak form of conformance, thanks to the following theorem.

Theorem 1: Given a feature model fm, if for every non-equivalent mutant $fm' \in \texttt{mut}(fm)$ there exists a distinguishing configuration $dc \in DCs(fm, fm')$ such that $killed(fm', dc, o)$, then fm is *mutation-conformant*.

The advantage of the previous theorem is that it gives us a way for assessing mutation-conformance, since we know how to generate distinguishing configurations. A further advantage is that if we find a configuration *dc* that does not kill a mutant fm' (i.e., we fail in assessing mutation-conformance), we can say that, in that particular case, fm' is better than fm because fm is killed instead (by Def. 4). This gives us a way to *repair* the original model.

In the following, we make two classical assumptions: coupling effect and competent programmer hypothesis [14], which are adapted for feature models.

a) Coupling effect: According to Offutt, the Coupling Effect Hypothesis is that "complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect a high percentage of the complex faults?" [15]. In our case, we hypothesize that distinguishing configurations that detect all simple mutants in a feature models will also detect a large percentage of the complex faults. Unfortunately, this may be not always true, as shown in Example 3, where each distinguish configuration was unable to find the faults in the candidate model.

b) Competent designer: In principle, to prove that fm is conformant to the oracle, one would need to check all the configurations, since it is always possible to build a feature model that behaves as the oracle on all the configurations but one. This is not feasible because the number of configurations grows exponentially with the number of features. However, we can assume that the designer is competent enough and (s)he may insert a small number of local mistakes that can be individually found and fixed by our distinguishing configurations. We refer to this hypothesis as *competent designer*.

IV. Algorithms for Detecting and Removing Conformance Faults

Fig. 4 graphically depicts the situation in which we are operating. We start with a feature model fm that, however, may be not totally correct. In order to obtain a correct feature model for the SPL, we can iteratively apply some mutation

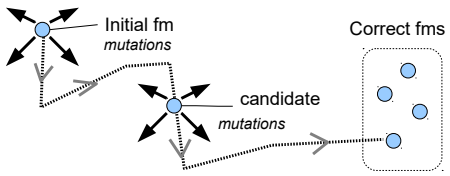


Fig. 4: Distance from the correct feature model

Algorithm 1 Meta-approach

Require: fm initial feature model
Require: o oracle
Require: `mut` mutation function
Ensure: a repaired feature model
  while *stoppingCondition* **do**
    $MUTs \leftarrow$ `mut`(fm)
    $DCs \leftarrow$ `buildDCs`$(fm, MUTs)$
    `evalMUTsAndCandidate`$(fm, MUTs, DCs)$
    `updateCandidate`$(fm, MUTs)$
  end while
  return fm

operators (i.e., we try to remove some faults from the feature model), and so traverse a path (i.e., a sequence of mutations) that brings to a feature model conformant with the oracle (or, at least, having a better conformance index than the initial model). Note that two feature models fm_1 and fm_2 may be connected by different paths, i.e., the correct feature model may be obtained from the original one with different sequences of mutations.

Alg. 1 shows an approach that, based on the previously mentioned idea, tries to repair a (possibly) faulty feature model. It is a classical greedy algorithm for traversing a graph. We choose a greedy approach because we assume that the exhaustive exploration of the graph is unfeasible. The algorithm consists in generating some mutants (using a given mutation function `mut`) for the current candidate, computing some distinguishing configurations between the current candidate and (a subset of) its generated mutants, checking how the candidate and the mutants evaluate the distinguishing configurations, and finally decide whether to change the candidate or not. These activities are repeatedly executed till the current candidate is not updated anymore.

This algorithm can be classified as a hill climbing [16] exploration of the graph, since it makes a small modification (mutation) at a time (local search) and it takes the new candidate if it is a better model. It is well known that hill climbing is very good in finding local optimum, but it may fail to find the global optimum.

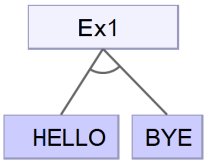
Such a meta-algorithm is very general, since it does not specify which mutation function `mut` is used (i.e., which are the used mutation operators and which is the order of mutations), how many mutants are considered at a time, how the comparison is done between the candidate and the mutants, when the candidate is changed, and which is the stopping condition. Different implementations of the previous phases are given in the algorithms we propose in Sects. IV-B–IV-E.

TABLE I: SPLOT models properties

model	#features	model	#features
aircraft	13	car	9
connector	20	fame_dbms	21
MarketPlace	26	monitor_engine	20
movies_app	13	REAL-FM-13	12
REAL-FM-14	18	REAL-FM-20	44
ReferenceManagement	31	SmartPhone	36
stack_fm	17	VolkswagenUp	35

```
#ifdef HELLO
char* msg = "Hello!\n";
#endif
#ifdef BYE
char* msg = "Bye bye!\n";
#endif
main() {
        printf(msg);
}
```

(a) The source code



(b) Its feature model

Fig. 5: Synthesis of feature models

A. Benchmarks

In order to compare the different algorithms, we have gathered the following three sets of benchmarks[1].

1) SPLOT: The first set consists of 14 SPLOT models which are often used as benchmarks[2]. They include models of aircraft configurations, car configurations, DBMS settings, configurations of a smartphone application, configurations of a monitor engine, etc. Table I reports the data about the 14 models in terms of number of features (recall that the number of configurations is equal to $2^{\#\text{features}-1}$). For each of these 14 models, we have built 10 faulty versions of the original model by randomly selecting by hand from 1 to 3 operations from the possible actions allowed by the FeatureIDE editor. We prefer to modify the original models not programmatically (i.e., by a simple Java program) in order to be more faithful to possible human errors and to avoid the use of the same mutation operators to insert and remove faults. For this benchmark set, the oracle is the original model.

2) CPP: A typical use of feature models is to model SPLs represented by preprocessor directives. For instance, Fig. 5a shows a C code fragment using preprocessor directives to build a small SPL; Fig. 5b shows the feature model obtained from the source code. The CPP set contains feature models developed by 6 of our students for 6 small programs taken from the literature regarding the synthesis of feature models from preprocessor directives of C/C++ source code. Although this technique is programming language specific, it can be easily extended in order to deal with more abstract and general concepts [17]. Each program has from 19 to 38 lines of code

[1]The tool, benchmarks, and results are available at https://github.com/fmselab/fmautorepair

[2]http://www.splot-research.org/

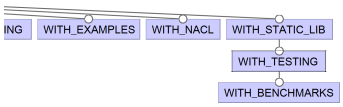


Fig. 6: Fragment of the LibSSH feature model

containing from 2 to 5 `#ifdef` or equivalent directives. We have instructed the students and asked them to build the 6 feature models. The resulting models contain from 2 to 6 features and maximum 2 constraints. In this case, we use the compiler as oracle: a configuration is valid if and only if it compiles successfully (this was explained to the students).

3) LibSSH: As real-life case study, we have analyzed `libssh`, a multi-platform library implementing SSHv1 and SSHv2 written in C[3]. The library consists of around 100 KLOC and can be configured by several options and several modules (like an SFTP server and so on) can be activated during compile time. We have analyzed the `cmake` files and identified 16 features (besides the root) and the relations among them. We have built a feature model for the case study which has 57344 products. Fig. 6 shows a fragment of the feature model in which `WITH_BENCHMARKS` can be activated only when `WITH_TESTING` is selected. In this case, the `cmake` and the `gcc` compiler act as oracles: a configuration is valid if the building system is able to produce an executable (without changing the options). A deeper analysis could consider successful only the builds that produce an executable that passes the tests, but this would require a deeper knowledge of the system (and some tricks when `WITH_TESTING` is not selected). Starting from the initial model, we have built 9 variations by introducing small errors. In total, we have 10 models for libssh.

Comparison of the techniques: In order to compare the different algorithms that we will present, we have executed each technique and computed the following measures:

AvgTime the average time taken (in secs) by the algorithm over all the models. In case we use as oracle external programs (like `gcc` and `cmake`), the time includes both the generation of distinguishing configurations and the execution of such programs.

MaxTime the maximum time taken by the algorithm over all the models.

#DCs the average number of generated distinguishing configurations over all the models.

ΔC ratio between the average of the differences between the final and the initial conformance[4] over the average gap distance (in terms of conformance) between the initial model and the oracle. Formally:

$$\frac{\texttt{AVG}(confIndex_{final} - confIndex_{init})}{1 - \texttt{AVG}(confIndex_{init})}$$

ΔC measures how much the initial unconformity has been repaired.

[3]https://www.libssh.org/

[4]We have computed $confIndex$ only for evaluation purposes. However, it cannot be used in the algorithms since it may very expensive to calculate.

Algorithm 2 Naïve approach

Require: fm initial feature model
Require: o oracle
Ensure: a feature model being mutation-conformant

```
1:  MUTs ← mut1(fm)
2:  genDCs ← ∅
3:  while MUTs ≠ ∅ do
4:      fm' ← pickone(MUTs)
5:      if ∃c ∈ genDCs : killed(fm', c, o) then
6:          MUTs ← MUTs \ {fm'}
7:      else
8:          dc ← generateDc(fm, fm')
9:          genDCs ← genDCs ∪ {dc}
10:         if killed(fm', dc, o) ∨ fm ≡ fm' then
11:             MUTs ← MUTs \ {fm'}
12:         else
13:             fm ← fm'
14:             MUTs ← mut1(fm)
15:         end if
16:     end if
17: end while
18: return fm
```

%F the percentage of models that are improved (some faults are repaired). It counts how many times ΔC is strictly positive.

%C the percentage of models that are transformed in a correct model (i.e., a model equivalent to the oracle). It counts how many times ΔC is equal to 100%.

%W the percentage of models whose conformance has been lowered by the algorithm. It counts the number of cases in which ΔC is negative.

All the experiments have been executed on a Linux PC with two Intel(R) Xeon(R) CPU E5-2630 (2.30GHz) and 64 GB of RAM. All the reported results are the average of 20 runs with a timeout for a single model of 3600 secs.

B. `Naïve` approach

In this first approach, we try to repair the feature model by looking for a model being mutation-conformant. Alg. 2 shows the proposed approach (called `Naïve`). It takes in input a feature model fm describing the SPL and an oracle (the SPL itself). It first generates the set of first-order mutants for fm (line 1) and initializes the set $genDCs$ of distinguishing configurations with the empty set. Then, for each mutant fm' of fm, it checks if there is a (previously computed) configuration able to kill fm' (line 5), and, if this is the case, it removes fm' from the set of mutants (line 6). Otherwise, it computes a new distinguishing configuration for fm and fm' (line 8), and adds it to the set $genDCs$. If the new distinguishing configuration dc is able to kill fm' or if fm and fm' are equivalent (i.e., there is no configuration that distinguishes them) (line 10), fm' is removed from the set of mutants (line 11). Instead, if fm' is a non-equivalent mutant not killed by dc, it means that fm contains a fault that has

TABLE II: Naïve approach – Results

benchmark	AvgTime	MaxTime	ΔC	#DCs	%F	%C	%W
SPLOT	54.24	1979.56	65.66	32.99	90.71	27.92	4.29
CPP	0.67	3.24	25.78	3.20	83.18	73.18	0
LibSSH	209.70	627.45	67.76	28.92	94	56	3
All	43.41	1979.56	48.91	28.21	87.4	34.35	3.56

been removed in fm'; therefore, fm' is now considered as the new best candidate for correctly representing the SPL (i.e., fm is substituted by fm') (line 13) and a new set of mutants is computed for the new candidate (line 14). The algorithm iterates until the set of mutants to be considered is empty, i.e., when it finds a feature model whose mutants are all killed: this means that the algorithm guarantees to return a mutation-conformant feature model (see Def. 9).

Note that the algorithm does not guarantee to return a feature model with a better conformance (see Def. 8) than the original one.

Theorem 2: Under the assumption that no features are added in the mutated models (i.e., the space of configurations does not increase during the computation), the naïve algorithm terminates.

Proof: At each iteration of the algorithm, the set of mutated models $MUTs$ can be reduced of one unit or reinitialized with the mutants of the new feature model. Every time $MUTs$ is reinitialized, a new configuration has been added to $genDCs$. Since the number of configurations is finite, when all the configurations have been added to $genDCs$, any new considered mutated model is removed, either because killed or because equivalent. Therefore, the algorithm terminates. ■

Results are reported in Table II. It reports the measures for the three benchmarks taken individually and for all the models considered together. We will use these results as comparison for the following approaches. We will always refer to the data regarding all the models, if not stated otherwise.

C. MultiDCs approach: Incrementing the number of DCs

In the naïve algorithm, the current candidate fm is updated with the mutant fm' if fm is killed by the configuration generated for the comparison with fm'. However, the mutant may be killed by other distinguishing configurations and changing the candidate may be not a good choice since it would bring us towards models with lower conformance. Based on this observation, the MultiDCs approach consists in generating n distinguishing configurations (with $n \geq 2$) at line 8 of Alg. 2 and keeping the model (either fm or fm') that survives more times. Alg. 3 shows the modifications w.r.t. the naïve algorithm. Note that there may be not n distinguishing configurations between fm and fm': in these cases, we will generate less than n configurations.

We have executed the algorithm with values of n, from 2 to 7. We here report the results related to $n = 3$ that has been experimentally proved to be the best option; indeed, it seems that considering too much configurations is not advantageous:

Algorithm 3 MultiDCs approach

Require: n # of distinguishing configurations to generate

8: $DCs \leftarrow$ `generateDCs`(fm, fm', n)
9: $genDCs \leftarrow genDCs \cup DCs$
10: **if** $|\{c \in DCs | killed(fm', c, o)\}| \geq |\{c \in DCs | killed(fm, c, o)\}|$ **then**
11: ...
15: **end if**

TABLE III: MultiDCs approach ($n = 3$) – Results

benchmark	AvgTime	MaxTime	ΔC	#DCs	%F	%C	%W
SPLOT	104.56	2853.81	76.86	75.61	91.21	31.71	4.0
CPP	0.87	4.62	28.22	4.19	84.01	72.73	0
LibSSH	536.04	1124.72	69.10	74.61	93	60	6
All	83.61	2853.81	52.4	64.67	87.85	37.51	3.50

TABLE IV: 2nd-order approach – Results

benchmark	AvgTime	MaxTime	ΔC	#DCs	%F	%C	%W
SPLOT	1424.76	3600	61.48	62.04	91.00	21.57	4.79
CPP	0.91	5.236	39.33	3.99	88.18	80.45	0
LibSSH	490.26	1126.57	72.60	87.27	100	58	0
All	1143.83	3600	56.07	51.24	88.47	30.38	3.78

it increases the time spent in generation and evaluation without producing any increase in the final conformance.

Results are reported in Table III. They show that requiring more distinguishing configurations permits to obtain better results in terms of final conformance (ΔC is 3.5% higher than in the naïve approach). This is due to the fact that the algorithm is less greedy and changes the candidate only if the mutant correctly evaluates more distinguishing configurations than the current candidate. This permits to reduce the probability of diverging from the oracle. A disadvantage of the approach is that it produces more configurations (more than the double of the naïve approach) and, therefore, the execution time increases; this is particular concerning for SPLs in which the verification of a configuration over the oracle is expensive (e.g., if some code must be compiled). The percentages of fixed, and worsened models are similar to the naïve approach, while 3% more models are corrected completely.

D. nth-order approach: Incrementing the order of mutations

This approach consists in increasing the order of mutations in Alg. 2 (at lines 1 and 14). In our experiments, we have considered second order mutation (i.e., mutation function mut2). The approach is called 2nd-order from now on.

Results are reported in Table IV. They show that increasing the order of mutation can give some advantages, but at the price of an increase of one order of magnitude of the execution times w.r.t. the previous approaches. Indeed, second order mutation produces a high number of mutants (the square of the number of mutants produced with the other approaches) and this has a huge effect on the computation times. Moreover, we can note that, although ΔC is higher (the approach can find more faults), the percentage of models corrected completely

Algorithm 4 Breadth approach

Require: fm initial feature model
Require: o oracle
Ensure: a feature model with a possibly better $confIndex$

```
1:  genDCs ← ∅
2:  bestFM ← null
3:  while bestFM ≠ fm do
4:      if bestFM ≠ null then
5:          fm ← bestFM
6:      end if
7:      MUTs ← mut1(fm)
8:      for all fm' ∈ MUTs do
9:          if ¬∃c ∈ genDCs : val(fm, c) ≠ val(fm', c) then
10:             dc ← generateDc(fm, fm')
11:             if fm ≢ fm' then
12:                 genDCs ← genDCs ∪ {dc}
13:             end if
14:         end if
15:     end for
16:     bestConf ← partialConfIndex(fm, o, genDCs)
17:     bestFM ← fm
18:     for all fm' ∈ MUTs do
19:         mutConf ← partialConfIndex(fm', o, genDCs)
20:         if mutConf > bestConf then
21:             bestConf ← mutConf
22:             bestFM ← fm'
23:         end if
24:     end for
25: end while
26: return fm
```

is lower: indeed, approach `2nd-order` is very greedy and it can quickly find a candidate that is a local maximum but that it may be far from the oracle.

E. `Breadth` approach

This approach starts from the observation that the previous algorithms may change the candidate before checking all the mutants of the current candidate. In this way, we may not evaluate some feature models that could give good results. The idea of the approach is to generate a distinguishing configuration for all the mutants and then to choose the mutant that correctly evaluates most of the generated configurations. The approach is called `Breadth` because it executes a kind of breadth exploration of the graph near the current candidate, instead of quickly going in depth as the previous approaches.

Alg. 4 shows the proposed approach. Given a candidate fm, it first generates all the first order mutants of fm (line 7) and generates a distinguishing configuration for each non-equivalent mutant (line 10); we avoid generating a configuration if there is a previously generated configuration that is distinguishing for the mutant (control at line 9). Then it computes how many generated configurations are evaluated correctly (called `partialConfIndex`) by the current candidate (line 16) that is marked as *best* feature model over the configurations in $genDCs$ (line 17). Then, it checks if there is a mutant better than the candidate and all the other mutants (loop at line 18). If the current candidate is still the best model, the algorithm terminates returning the current candidate as repaired model; otherwise, the while loop continues using as current candidate the found best model.

Note that this approach does not guarantee to obtain a feature model that is mutation-conformant.

Results are reported in Table V. The approach is expensive in terms of execution time because all the mutants of the current candidate are always considered; however, it still takes half of the time taken by `2nd-order`. Regarding ΔC, it is only slightly better than `naïve`, but worst than the other two approaches: this is due to the fact that the current candidate may also be killed by some configurations in $genDCs$. However, it is the approach that completely repairs the greater percentage of models (9% more models than `MultiDCs`) and worsens the lower percentage of models.

TABLE V: `Breadth` approach – Results

benchmark	AvgTime	MaxTime	ΔC	#DCs	%F	%C	%W
SPLOT	732.35	3600	71.42	72.75	91.57	42.79	2.42
CPP	0.73	3.218	28.89	3.43	82.27	73.18	0
LibSSH	311.86	912.35	71.0	32.97	100	61	0
All	583.95	3600	51.68	59.92	88.30	46.39	1.92

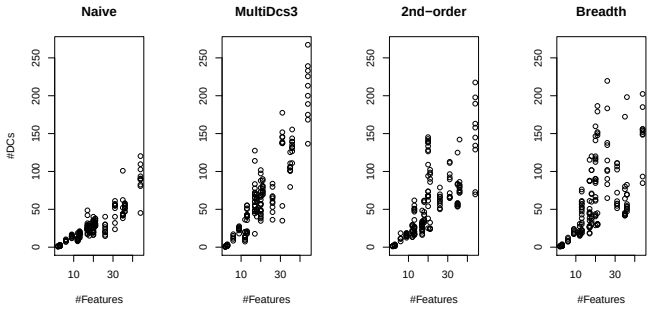


Fig. 7: # Features vs # generated DCs

V. Evaluation

We have already presented a preliminary evaluation of the algorithms in Tables II, III, IV, and V. In this section, we deepen our analysis guided by the following research questions.

RQ1 How does the number of generated distinguishing configurations grow w.r.t. the number of features of the starting feature model?

Such question permits to evaluate if our techniques scale well, i.e., if the number of generated configurations does not grow too much with the size of the model. Fig. 7 shows the number of configurations generated by each technique depending on the number of the features. We can see that number of generated configurations is linear with the number of features of the model. Indeed, the techniques usually

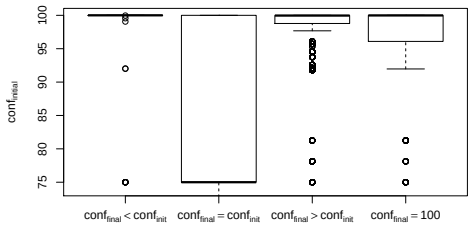
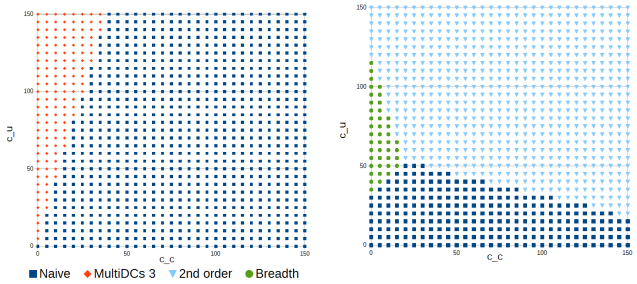


Fig. 8: Initial conformance vs reparability



(a) SPLOT (b) LibSSH

Fig. 9: Cost model

generate a configuration for each mutant of the model and the number of mutants is linear with the number of features [12]. In case of `2nd-order`, the number of mutants should grow in a quadratic way, but a distinguishing configuration may kill many mutants at once. Overall, even for the bigger model with 2^{44} configurations, we generate at most only around 300 configurations. Our approach has therefore the advantage that a small number of configurations must be considered, differently from approaches that do synthesis starting from the configuration set (like, for instance, [18]).

RQ2 Is the initial conformity a good indicator of the *reparability* of the feature model?

We can hypothesize that models with an initial conformity near to 100% are more easily repaired. Fig. 8 shows the correlation between the initial conformity and the final one in four cases: the conformity is lowered, it remains as the initial one, the model has been repaired, and it has been totally fixed. It is apparent that the hypothesis cannot be confirmed with enough confidence. Our techniques are able to repair even models with lower conformity than models for which the techniques do not complete the repairs or even introduce new faults. There is evidence (second boxplot) that if the model has a relatively low conformance (around 75%), it is very likely that the algorithms will leave it as it is. A deeper investigation suggests that the conformance index is not a good indicator of the number of faults contained in the model, because there are single faults that are able to significantly reduce the conformity (for examples faults near the root) but can be easily removed by a single mutation.

RQ3 Which is the best technique? How to choose it?

From the experiments, it is apparent that there is no *one* best technique: algorithms that are fast produce fewer distinguishing configurations but may miss some faults, while techniques that are slow may produce better results in terms of fault detection. In order to compare the algorithms, we introduce a cost model, which takes into account the following main cost factors of our process:

- c_t the cost of the single time unit during generation of the distinguishing configurations.
- c_c the cost of checking a single configuration. It can be the time required to run some external tool or to manually verify the validity of a configuration.
- c_u the cost of the missing conformity. It can be the time needed to find and fix unexpected faults.

We can therefore compute the cost of an algorithm by the following schema:

$$Cost = time_{gen} \times c_t + \#DCs \times c_c + (100 - \Delta\mathrm{C}) \times c_u$$

We can compare the four algorithms using the statistical data we have collected for $time_{gen}$, $\#DCs$, and ΔC. For simplicity, we assume the cost $c_t = 1$ and we compute for each technique the *Cost* at the variation of parameters c_c and c_u. The technique that gives the least cost is displayed in Fig. 9. Fig. 9a shows the best techniques for the SPLOT models: only the `naïve` and the `multiDCs` approaches are convenient. However, the results also depend on the considered feature model. If we check the results in Fig. 9b for LibSSH, we can see that the `multiDCs` approach is never convenient, and instead `2nd-order` and `breadth` approaches are convenient for some parameter values. For example, if the cost c_u of an inconformity is high (higher than 100), using `2nd-order` is worthy, regardless the cost of checking a single configuration.

Since the constants $time_{gen}$, $\#DCs$, and ΔC may depend on the single model under repair, we plan to study other techniques able to predict such quantities starting from the characteristics of the SPL. In absence of such a predictive model, the user could hypothesize that the starting feature model is the oracle, build a couple of mutations and run the different algorithms on such mutations using the original model as oracle instead of the real system, and then use the measured $time_{gen}$, $\#DCs$, and ΔC in the cost model in order to decide which technique to apply. Although this may be not very precise and more experiments are needed, we believe that this should give a good estimation of such quantities.

VI. Threats to validity

Regarding internal validity, the first threat regards the choice of the framework and the feature models on which we performed the experiments. We have chosen a public repository (SPLOT) and a good number of models, besides our own SPLs (CPP and LibSSH). FeatureIDE is a mature framework and, whenever possible, we have checked the results using also the SPLOT Java library (for example, to count the valid configurations). Therefore, we believe that our experiments report accurate data. Regarding mutation for feature models, there is a good level of consensus on which operators to

use [12], [5], [19]. We believe that other approaches, like mutating the propositional formulas representing the feature model [20], do not suit our goals of finding conformance errors. Another threat regards the type of faults we have artificially inserted in the original models for the SPLOT benchmark set and for the LibSSH set. In order to avoid biased modifications, each author has generated around one third of the mutations and he has used only the graphical editor of FeatureIDE. Unfortunately, the limited knowledge of the domain the feature models refer to, makes our faults only plausible. We plan to work with real users and real faults (as done for the CPP benchmark set).

In our experiments, all the mutation operators (and therefore the faults they try to fix) are treated equally, while in the reality a type of fault may occur more often than others and the designer may have an idea on which faults to target. One could introduce a probabilistic model on faults and filter or order the mutations accordingly. For instance, if one suspects that the designer may have confused the use of *Or* with *And* among features, the technique should try to find that type of fault by using the mutation operators **AndToOr** and **OrToAnd**. We believe that by using such fault models, our techniques can only improve, but further studies are necessary.

VII. Related work

The problem of reverse engineering a feature model is well studied in the literature. A direction of research tries to synthesize a feature model from its feature sets. Generally, these techniques identify patterns among features in products and in invalid configurations and build hierarchies and constraints (in limited form) among them. For instance, Davril et al. apply feature mining and feature associations mining to informal product descriptions [21]. There exist several papers that apply search based techniques, which generally give better results [22], [23], [24], [25]. They are similar to our technique since they use evolutionary algorithms which mutate feature models using mutation and crossover operators (as in [5]). However, their starting hypotheses are much different, since they assume that the list of products is available and therefore they can check a mutant with all the configurations (valid and invalid) and build a very precise fitness function (which is similar to our *conformance index*).

Feature models can be synthesized also from propositional formulas describing the relations among features [6], [26]. These approaches employ ad hoc algorithms that extract logical formulas from configuration, documentation, and source files. Once the formulas are extracted, the construction of the feature model begins (for example with the identification of the parents). No technique is proposed to validate the starting formula, and we believe that our technique could be used also to check and repair constraints among features.

The new idea presented in [7] is very close to our work. They start from a feature model given as a set of constraints over the features and, through a continuous cycle of test-and-fix, they improve the original constraints in order to reduce the number of wrong constraints. At every step, three possible operations on the constraints are considered (altering, removing, and inserting). Configurations both derived from the model and from the real system are checked in order to discover errors in the model. Besides the fact that they use feature models represented exclusively by constraints, they assume the availability of a great number of configurations (the evaluation has been done with 1000 configurations), while we assume that the number of checked configurations is kept small. Moreover, while they produce products randomly using a SAT solver (from the model) or using an oracle from the actual system (like *make*), we generate configurations solely from the model and not randomly, but guided by fault classes in a way that precise behavioral faults can be detected. Moreover, we do not generate only products, but any possible configuration.

In order to kill mutants we use a symbolic approach, based on the definition of distinguishing configurations, the logical representation of feature models, and a SAT solver. There are other approaches that instead use a search based Evolutionary Algorithm (EA) also for mutating the set of configurations [27]. It would be interesting to extend their test generation technique also for repairing feature models.

The use of constraint solvers to fix an invalid configuration is also proposed in [28]. However, in that case it is used to mediate conflicts among several versions of the same SPL, which may be produced from different actors in the feature modeling process. Our distinguishing configurations could be reused to pinpoint configuration errors and identifying corrective actions.

VIII. Conclusions and Future work

We have presented a methodology that can either detect a *conformance* fault in the feature model candidate to represent the SPL under test or prove its mutation-conformance. We have devised a family of techniques that are able to iteratively remove conformance faults from the candidate. Experiments show that our techniques are efficient, and can repair most of the faults with a reasonable number of configurations to check against the oracle.

As future work we plan to explore the use of well known techniques able to free the search from being stuck in local optima when using hill climbing. For instance, a direction would be to apply simulated annealing, in which we could change the candidate even if its mutation is not better than it (to also accept equivalent mutants, for instance). Another direction is using random restarts. This would increase the number of configurations to be used against the oracle, but it would increase the success of the search.

Although we have applied our technique to a real case study like libssh, we would like to test our framework on big feature models like that of the Linux kernel, for which there exists a feature model which is expressed in terms of constraints over features (and not as feature diagram) [29].

With our surprise, generating second order mutants did not improve the results, also because many times the algorithm was stopped by the timeout. We plan to work on improving

the mutation generation in order to speedup and filter the generation of *better* higher order mutants.

ACKNOWLEDGMENT

The work was partially supported by Charles University research funds PRVOUK.

REFERENCES

[1] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems*, vol. 35, no. 6, pp. 615–636, 2010.

[2] C. Kästner, A. Dreiling, and K. Ostermann, "Variability mining: Consistent semi-automatic detection of product-line features," *Software Engineering, IEEE Transactions on*, vol. 40, no. 1, pp. 67–82, 2014.

[3] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining configuration constraints: Static analyses and empirical results," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 140–151.

[4] W. Fenske, T. Thüm, and G. Saake, "A taxonomy of software product line reengineering," in *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, ser. VaMoS '14. New York, NY, USA: ACM, 2013, pp. 4:1–4:8.

[5] 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*, vol. 103, pp. 353–369, 2015.

[6] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki, "Reverse engineering feature models," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 461–470.

[7] 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*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 1245–1248.

[8] M. Acher, A. Cleve, G. Perrouin, P. Heymans, C. Vanbeneden, P. Collet, and P. Lahire, "On extracting feature models from product descriptions," in *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, ser. VaMoS '12. New York, NY, USA: ACM, 2012, pp. 45–54.

[9] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "FeatureIDE: An extensible framework for feature-oriented software development," *Science of Computer Programming*, vol. 79, no. 0, pp. 70–85, 2014.

[10] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps, "Generic semantics of feature diagrams," *Computer Networks*, vol. 51, no. 2, pp. 456 – 479, 2007.

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

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

[13] A. Gargantini and G. Fraser, "Generating minimal fault detecting test suites for general boolean specifications," *Information and Software Technology, Elsevier*, vol. 53, pp. 1263–1273, 2011.

[14] R. Demillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, April 1978.

[15] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Trans. Softw. Eng. Methodol.*, vol. 1, no. 1, pp. 5–20, Jan. 1992.

[16] D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis, "How easy is local search?" *Journal of Computer and System Sciences*, vol. 37, no. 1, pp. 79–100, 1988.

[17] T. Ziadi, C. Henard, M. Papadakis, M. Ziane, and Y. Le Traon, "Towards a language-independent approach for reverse-engineering of software product lines," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, ser. SAC '14. New York, NY, USA: ACM, 2014, pp. 1064–1071.

[18] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed, "On extracting feature models from sets of valid feature combinations," in *FASE*, ser. Lecture Notes in Computer Science, vol. 7793. Springer, 2013, pp. 53–67.

[19] D. Reuling, J. Bürdek, S. Rotärmel, M. Lochau, and U. Kelter, "Fault-based product-line testing: Effective sample generation based on feature-diagram mutation," in *Proceedings of the 19th International Conference on Software Product Line*, ser. SPLC '15. New York, NY, USA: ACM, 2015, pp. 131–140.

[20] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. Le Traon, "Assessing software product line testing via model-based mutation: An application to similarity testing," in *9th Workshop on Advances in Model Based Testing (A-MOST) ICST Workshop*. IEEE, 2013, pp. 188–197.

[21] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang, and P. Heymans, "Feature model extraction from large collections of informal product descriptions," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 290–300.

[22] M. Harman, Y. Jia, J. Krinke, W. B. Langdon, J. Petke, and Y. Zhang, "Search based software engineering for software product line engineering: A survey and directions for future work," in *Proceedings of the 18th International Software Product Line Conference - Volume 1*, ser. SPLC '14. New York, NY, USA: ACM, 2014, pp. 5–18.

[23] R. E. Lopez-Herrejon, L. Linsbauer, and A. Egyed, "A systematic mapping study of search-based software engineering for software product lines," *Information and Software Technology*, vol. 61, pp. 33 – 51, 2015.

[24] J. M. Ferreira, S. R. Vergilio, and M. A. Quináia, "A mutation approach to feature testing of software product lines," in *The 25th International Conference on Software Engineering and Knowledge Engineering, Boston, MA, USA, June 27-29, 2013*. Knowledge Systems Institute Graduate School, 2013, pp. 232–237.

[25] R. E. Lopez-Herrejon, J. A. Galindo, D. Benavides, S. Segura, and A. Egyed, "Reverse engineering feature models with evolutionary algorithms: An exploratory study," in *Search Based Software Engineering*, ser. Lecture Notes in Computer Science, G. Fraser and J. Teixeira de Souza, Eds. Springer Berlin Heidelberg, 2012, vol. 7515, pp. 168–182.

[26] S. She, U. Ryssel, N. Andersen, A. Wąsowski, and K. Czarnecki, "Efficient synthesis of feature models," *Information and Software Technology*, vol. 56, no. 9, pp. 1122–1143, 2014, special Sections from "Asia-Pacific Software Engineering Conference (APSEC), 2012" and "Software Product Line conference (SPLC), 2012".

[27] C. Henard, M. Papadakis, and Y. Le Traon, "Mutation-based generation of software product line test configurations," in *Search-Based Software Engineering*, ser. Lecture Notes in Computer Science, C. Le Goues and S. Yoo, Eds. Springer International Publishing, 2014, vol. 8636, pp. 92–106.

[28] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortés, "Automated diagnosis of product-line configuration errors in feature models," in *Proceedings of the 2008 12th International Software Product Line Conference*, ser. SPLC '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 225–234.

[29] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "Variability modeling in the real: a perspective from the operating systems domain," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 73–82, 00106.