Generating Tests for Detecting Faults in Feature Models

Paolo Arcaini
Department of Engineering
University of Bergamo, Italy
Email: paolo.arcaini@unibg.it

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

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

***Abstract*—We present a novel fault-based approach for testing feature models (FMs). We identify several fault classes that represent possible mistakes one can make during feature modeling. We introduce the concept of *distinguishing configuration*, i.e., a configuration that is able to detect a given fault. Starting from this definition, we devise a technique, based on the use of a logic solver, able either to find distinguishing configurations to be used as tests or to prove that a mutation produces an equivalent feature model. Compact test suites can be produced by exploiting an SMT solver. The experiments show that our methodology is viable and produces reasonable sized test suites in a short time. W.r.t. the approaches that use only the products, our approach has a better fault detection capability and requires fewer tests.**

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. There exist several tools for designing feature models, like FeatureIDE [1], and repositories and libraries like SPLOT [2]. Test generation for feature models and SPLs has attracted a lot of recent attention in research [3]. In the context of SPLs, a test is a configuration which may be valid, called *product*, (and accepted by the SPL) or even invalid (that must be rejected by the SPL). For real life SPLs, exhaustive testing (i.e., the generation of all the possible configurations) is infeasible, even if one limits to the testing of products. For this reason, researchers look for test generation methodologies from feature models which try to combine manageable sizes, practicality, and usefulness of the generated test suites. A classical test generation approach consists in applying combinatorial interaction testing (or T-wise testing) [4], [5], [6].

In this paper, we try to apply a *fault-based testing approach* [7] to feature model testing with the goal of finding defects in the models. Fault-based testing consists in generating test suites that try to demonstrate that prescribed faults are not in the artifact under test. It assumes that the program or the model can only be incorrect in a limited fashion specified by relatively small type of common mistakes. It is often used in conjunction with *mutation analysis* [8]. In mutation analysis, faults are deliberately inserted (aka *seeded*) in the software artifacts by simple syntactic changes, in order to create a set of faulty programs called *mutants*, each containing a different syntactic change. Mutations can be used to drive test generation with the goal of building tests able to distinguish an artifact from its mutants.

Our approach consists in mutating the feature model by seeding faults and using these mutations to obtain tests able to detect those faults. We present a relatively small set of typical fault classes that can be found in feature models, and we model them as mutation operators (we derive the terminology from mutation testing). Then we devise a method, based on the use of a logic solver, that generates configurations that are able to distinguish a feature model from its faulty version. We call these *distinguishing configurations*. We adapt an existing greedy method in order to build a test suite able to detect all the seeded faults. Moreover, since a mutated model can be equivalent to the original model (i.e., it describes the same set of products), our technique can also be exploited for checking models equivalence. The experiments indicate that the number of generated tests is still manageable (and grows linearly with the number of features and constraints) and that our approach is efficient in finding user faults although the mutation operators set is rather simple.

Sect. II presents the notation of feature models, their semantics, and the use of mutation for testing. In Sect. III, we present several fault classes for FMs, the concept of equivalent mutants and the definition of distinguishing configuration. How to automatically generate distinguishing configurations is presented in Sect. IV. Experiments are reported in Sect. V and Sect. VI reports some related work. Sect. VII 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 a 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 and in Table I):

- *Or* – at least one of the sub-features must be selected if the parent is selected.

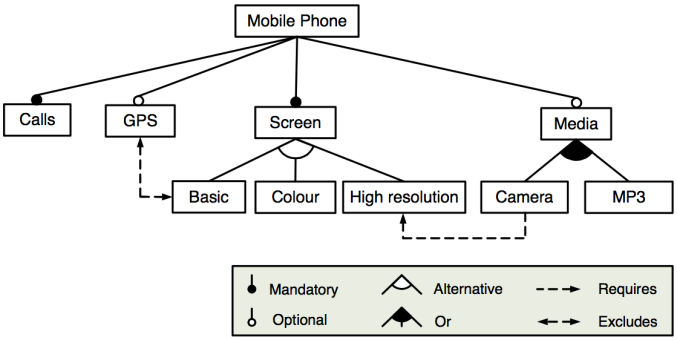


Figure 1: Example of a Feature Model

- *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.

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 [9] 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.

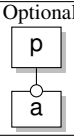
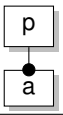
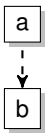
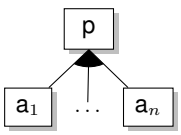
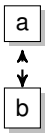
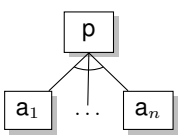
Notation	Propositional formula	Notation	Propositional formula
Optional 	$a \rightarrow p$	Mandatory 	$(p \rightarrow a) \wedge (a \rightarrow p)$
Requires 	$a \rightarrow b$	Or 	$(p \rightarrow (a_1 \vee \dots \vee a_n)) \wedge (a_1 \rightarrow p) \wedge \dots \wedge (a_n \rightarrow p)$
Excludes 	$a \rightarrow \neg b$	Alternative 	$(p \rightarrow alt(a_1, \dots, a_n)) \wedge (a_1 \rightarrow p) \wedge \dots \wedge (a_n \rightarrow p)$
Root 	p	General form	*gfConstr* / *gfConstr*

Table I: Conventional translation in propositional formulae

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 [1] has been used to design, import, analyze, mutate, and validate the models used in the evaluation section.

1) Feature Model semantics: Feature models semantics can be rather simply expressed by using propositional logic as already done in [10], [9]. Every feature becomes a propositional letter, and every relation among features becomes a propositional formula modeling the constraints about them as reported in Table I[1]. In the following, we will freely use a feature name also to identify its corresponding propositional variable.

Definition 1: Let BOF be a function that, given a feature model, returns its representation as propositional formula.

Let $r_1, \dots, r_n$ be all the parent-child relations, the extra-constraints, and the root of a feature model $\mathcal{M}$. BOF($\mathcal{M}$) is given by the conjunction of the propositional formula of each r_i (with $i = 1, \dots, n$) as shown in Table I.

Example 2: The BOF for the feature model of Fig. 1 is MobilePhone $\wedge$ (MobilePhone $\rightarrow$ Calls) $\wedge$ (Calls $\rightarrow$ MobilePhone) $\wedge$ $\dots \wedge$ (Media $\rightarrow$ (Camera $\vee$ MP3)) $\wedge$ (Camera $\rightarrow$ Media)$\wedge$ (MP3 $\rightarrow$ Media).

2) Configuration and Products:

Definition 3: A *configuration* of a feature model $\mathcal{M}$ is a subset of the features in $\mathcal{M}$ that must include the root.

If $\mathcal{M}$ has n features (including the root), there are 2^{n-1} possible configurations, which, however, are not all valid. A configuration is *valid* if it respects all the constraints of $\mathcal{M}$, derived from the parental relations and by the extra-constraints.

[1]In Table I the *alt* operator represents the exclusive or among all its arguments and it is defined as $alt(a_1, a_2, \dots, a_n) = (a_1 \wedge \neg a_2 \wedge \dots \wedge \neg a_n) \vee \dots \vee (\neg a_1 \wedge \neg a_2 \wedge \dots \wedge a_n)$.

A valid configuration is called a *product*, since it represents a possible instance of the feature model.

Note that configurations can contain also abstract and non-terminal features, differently from [11], since we do not explicitly focus only on testing artifacts of a SPL.

There exists a clear relation between products of $\mathcal{M}$ and $\text{BOF}(\mathcal{M})$: a product is a model of $\text{BOF}(\mathcal{M})$. So, checking whether a configuration is a product corresponds to checking whether the configuration makes $\text{BOF}(\mathcal{M})$ true. Moreover, finding a product for $\mathcal{M}$ is equivalent to finding a model of $\text{BOF}(\mathcal{M})$. This fact can be used to generate products as in [12].

B. Mutation testing

Mutation is a well known technique in the context of software code and formal modeling. Program mutation consists in introducing small modifications into program code such that these simple syntactic changes, called mutations, represent typical mistakes that programmers often make. These faults are deliberately seeded into the original program in order to obtain a set of faulty programs called mutants. The use of mutations is twofold. First, they can be used to guide the test generation. Second, mutants are classically used to evaluate other testing approaches (*mutation analysis*) and, for this reason, program mutation is almost always used in combination with testing. High quality test suites should be able to distinguish the original program from its mutants, i.e., to detect the seeded faults. The history of mutation testing can be traced back to the 70s [8]. Mutation testing has been applied to many programming languages, to formal notations [13] (as in this work), and to several application domains. However, as far as we know, mutation analysis has never been directly applied to feature models. Preliminary attempts to join mutation analysis and feature models are presented in [14], [12].

III. Fault models for FMs

The first contribution of this paper regards the mutation of feature models. Given a feature model $\mathcal{M}$, we are interested in finding possible faulty version of $\mathcal{M}$ called *mutants*. In order to generate mutated models, some mutation operators must be defined, i.e., rules that specify syntactic variations of the model to be mutated. Mutation operators are derived from fault classes, i.e., from the families of errors that can be introduced by the developer in the model.

We have devised the following fault classes and corresponding mutation operators, divided in *feature-based* and *constraint-based* mutation operators. 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 $\mathcal{M}$ (b) Mutant $\mathcal{M}'$

Figure 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.

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.

In this paper, we focus only on simple faults for the constraints and we leave as future work complex faults that can be borrowed from the classical logical mutation operators of Boolean expressions in general form [15].

The mutation operators we introduce in this paper are inspired by the feature model edits presented in [11] and by the refactoring actions of [16]. However, none of the proposed mutation operators can extend the feature set, since in this work we assume that a user may forget a feature but not add a new unexpected one; we leave this as future work, together with the addition of new constraints.

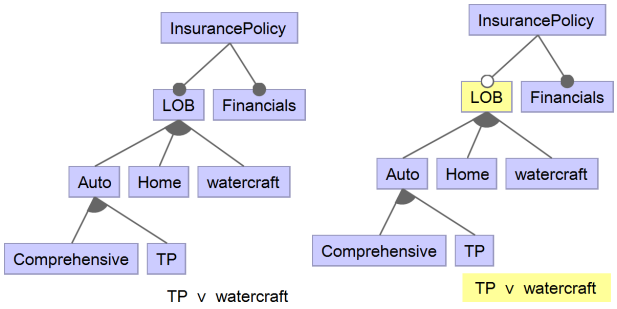
Example 4: Fig. 2 shows a model and one of its possible faulty implementations, namely the model obtained by applying the **ManToOpt** operator to the relation between the root node a and its child b.

Mutants can be obtained by applying one mutation operator at a time (*first order* mutants) or several mutation operators in sequence (*higher order* mutants). Given a feature model $\mathcal{M}$ and one of its mutants $\mathcal{M}'$, we want to find a configuration that distinguishes $\mathcal{M}$ from $\mathcal{M}'$.

Definition 5: We say that a configuration c distinguishes $\mathcal{M}$ from $\mathcal{M}'$ if c is valid in $\mathcal{M}$ and not in $\mathcal{M}'$ or vice versa. We call this a **distinguishing** configuration.

We borrow the term *distinguishing* from the context of conformance testing between Finite State Machines. A distinguishing configuration is able to find the difference between $\mathcal{M}$ and $\mathcal{M}'$, and, in terms of mutation analysis, it *kills* the mutant. Note that a distinguishing configuration could be either a product or an invalid configuration for the original model. The validity information attached to a configuration is used as *oracle* when configurations are used as tests.

Example 6: Consider the model in Fig. 2a and its mutant in Fig. 2b. The configuration $\{a, c\}$ is valid in $\mathcal{M}'$ but it is invalid



(a) Insurance Policy ($\mathcal{M}$) (b) $\mathcal{M}'$, with LOB optional

Figure 3: An equivalent mutant

in $\mathcal{M}$: therefore, it is a distinguishing configuration. The configuration $\{a, b\}$, instead, is valid in both feature models $\mathcal{M}$ and $\mathcal{M}'$ and so it is not a distinguishing configuration.

We do not require distinguishing configurations to be products for $\mathcal{M}$ because some faults cannot be revealed by products, as shown in the following example.

Example 7: Consider the model and its mutant in Fig. 2. All the products of $\mathcal{M}$ are also products for $\mathcal{M}'$, although there exists the distinguishing configuration $\{a, c\}$. In this example, it is not possible to find a distinguishing configuration that is also a product for $\mathcal{M}$.

In our approach, distinguishing configurations can be used to detect faults in the feature model under test.

A. Equivalent mutants

Not all the mutants represent actual semantic changes in the feature model. A mutated model $\mathcal{M}'$ is said to be *equivalent* to the original model $\mathcal{M}$ if there is no distinguishing configuration for it. In this case, the mutation introduced in $\mathcal{M}'$ is not a real fault since it represents a syntactic mutation without any real change in the feature model semantics. Equivalent models represent exactly the same set of products.

Example 8: Consider the FM and one of it mutants shown in Fig. 3. The original model $\mathcal{M}$ (Fig. 3a) is a simplified version of a model taken from the SPLOT [2] website. In $\mathcal{M}$, LOB is mandatory and there is an extra-constraint stating that at least one between TP and watercraft must be present in any product. If we modify $\mathcal{M}$ by making LOB optional, we obtain the model $\mathcal{M}'$ of Fig. 3b. However, due to the effect of the extra-constraint, also in $\mathcal{M}'$ LOB must be selected: since TP or watercraft must be selected, their common ancestor LOB will be selected in any case in any product. In $\mathcal{M}'$, LOB is a *false optional* and there is no configuration that can distinguish $\mathcal{M}$ from $\mathcal{M}'$: if a configuration does not contain LOB, it is invalid for both models, otherwise, if it contains LOB, the validity result is the same in both models.

Equivalent mutants pose a challenge to any mutation analysis, since they cannot be detected by any configuration. The problem of equivalent mutants is well-known in mutation testing: an equivalent mutant is a mutant that does not change the semantics of the program; therefore, it is impossible to write a test that captures it. In general, detecting equivalent mutants is a time-consuming activity. Our approach is able to automatically identify equivalent mutants. However, since the search for equivalent mutants consume resources without producing useful tests, we try to avoid the generation of a mutant when the feature model topology guarantees that it will be equivalent. For instance, we never apply the **AlToOr** operator to an alternative feature having only one child, since it would always produce an equivalent mutant.

Note that in this paper we focus only on faults that can be detected by a configuration. There exist anomalies that cannot be detected by configurations since they produce equivalent mutants. In these cases, finding equivalent mutants is useful in order to expose this kind of anomalies. For instance, the problem of false optional features, which are declared as optional but actually required in all products (for example, due to a cross-tree constraint like in Fig. 3), would be exposed by an **OptToMan** equivalent mutation. This kind of problems is well studied in literature [9], considered outside the scope of this paper, and left as future work.

B. Use cases for distinguishing configurations

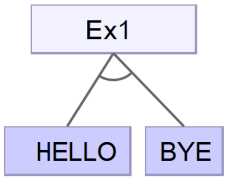
Distinguishing configurations are useful in many scenarios. The principal use case we foresee is to identify faults in feature models developed by the user or obtained using some automatic technique which, however, could make some mistake or require further user interactions.

The first usage scenario is when $\mathcal{M}$ is the feature model for a SPL as it has been designed, and the user is interested in validating $\mathcal{M}$, i.e., he/she wants to be sure that $\mathcal{M}$ captures the SPL he/she had in mind. This can be performed in the following way. All the distinguishing configurations can be generated together with their validity value (oracle), i.e., together with the fact that a distinguishing configuration dc is a product or not for $\mathcal{M}$. For every dc, the user must assess if the (computed) validity is what he/she expected, i.e., to decide those that are products or not as expected by the informal requirements. If every dc has a validity value judged correct by the user, we can conclude that $\mathcal{M}$ does not contain any of the faults we have seeded in order to obtain the distinguishing configurations. On the contrary, if a dc, that should be a product for the intended SPL, is not for the current model $\mathcal{M}$ (or the other way around), then a fault has been discovered. In case of design validation, it is important that the number of generated distinguishing configurations is small, since their evaluation is a human activity. Note that companies may find identifying products easier than building the feature model from scratch [17].

A similar problem is when a feature model is synthesized for a SPL containing a great quantity of features but without a precise model for them. In this case, a feature model is automatically reverse engineered starting from the set of products, as proposed in [18]. Sometimes feature models are synthesized starting from a given set of dependencies among features. These constraints can be either specified by engineers, or automatically mined from the source code using static analysis [19]. For instance, Fig. 4a shows a C

```
#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

Figure 4: Synthesis of feature models

code fragment using preprocessor directives to build a small SPL; Fig. 4b shows the feature model obtained from the source code. In these cases, distinguishing configurations are useful to validate both the synthesis process and the extracted dependencies among features.

As future work, we plan to investigate the use of distinguishing configurations for further uses. A first extension consists in using distinguishing configurations instead of products (as in [18]) to synthesize feature models. Feature models and corresponding distinguishing configurations can be used to test feature model analysis tools, especially components that check products validity (as done in [20]).

IV. Generation of Distinguishing Configurations

Fault-based testing can be successfully applied to Boolean expressions [21], [15]. The basic principle is the following one. The erroneous implementation φ' of a Boolean expression φ can be discovered only when the expression $\varphi \oplus \varphi'$, called *detection condition* (also called *boolean difference* or *derivative* [22]), is evaluated to true, where $\oplus$ denotes the logical *exclusive or* (xor) operator. Indeed, $\varphi \oplus \varphi'$ is true only if φ' and the correct predicate φ evaluate to two different values. This technique has been exploited in test generation for Boolean expressions [15]. The same principle can be applied to test generation for feature models.

Let $\mathcal{M}$ be a feature model and $\mathcal{M}'$ a mutated version of $\mathcal{M}$ due to the mutation operator F. Let φ be the Boolean proposition formula for $\mathcal{M}$, i.e., $\varphi = \text{BOF}(\mathcal{M})$. Let φ' be the Boolean proposition formula for $\mathcal{M}'$ *over all the literals defined in* φ. We have to be careful when we build φ'. Indeed, in case the operator F is **MF** (missing feature), $\text{BOF}(\mathcal{M}')$ will contain fewer literals than φ: every missing feature m will result in a missing literal m in $\text{BOF}(\mathcal{M}')$. In this case, we add, for every missing feature m, a further conjoint $\neg m$ in order to preserve the semantics of $\mathcal{M}'$ (a feature that has been removed with the fault **MF** cannot appear in a product for the obtained mutant $\mathcal{M}'$) and keep the same set of literals. A similar technique is proposed in [11].

Definition 9: Given a model $\mathcal{M}$ and a mutant $\mathcal{M}'$, the propositional formula φ' is defined as

$$\varphi' = \text{BOF}(\mathcal{M}') \wedge \bigwedge_{m \in fea(\mathcal{M}) - fea(\mathcal{M}')} \neg m$$

where *fea* returns the set of features of a feature model.

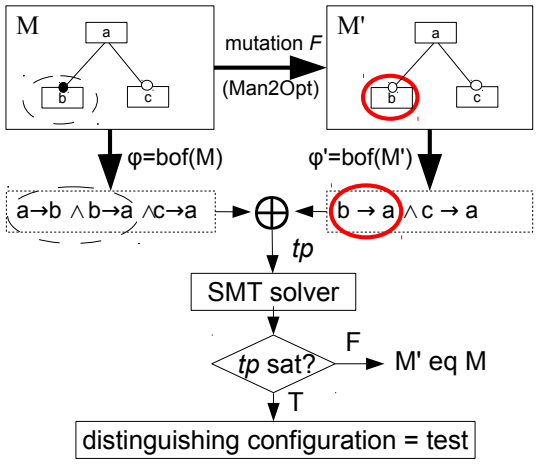


Figure 5: Generating a distinguishing configuration for a fault F by an SMT solver

Definition 10: The predicate $\varphi \oplus \varphi'$ is the detection condition of F and it is called *test predicate* (or *test goal*).

Theorem 11: A model of the test predicate is a distinguishing configuration.

Proof: For the semantics of the exclusive or, the test predicate $\varphi \oplus \varphi'$ is true if either φ or φ' is true. So a model of the test predicate makes true either φ or φ', that is what required by the definition of distinguishing configuration (see Def. 5). Note that adding the negation for possible missing features guarantees that every distinguishing configuration which is a model of φ' is also a product for $\mathcal{M}'$. Otherwise, a configuration could be a model of $\text{BOF}(\mathcal{M}')$ and contain a feature not available in $\mathcal{M}'$. ■

Theorem 12: If a mutation operator produces an equivalent mutant, its test goal is unsatisfiable.

A. Generation of Distinguishing Configurations

In order to find the distinguishing configurations, one can use a constraint solver like SAT, CSP, or SMT, as long as the solver is able to find a model of a formula or to prove that a formula is unsatisfiable. In this paper, we use the SMT solver Yices [23]. Indeed, SMT solvers are more flexible tools than SAT solvers: they have a richer command interface – allowing, for instance, adding and retracting assertions –, they are as powerful as SAT solvers when applied to satisfiability problems, and they accept as input generic Boolean formulae. Fig. 5 depicts the process of finding a single distinguishing configuration for a model $\mathcal{M}$ and its mutated version $\mathcal{M}'$.

In case the mutant is equivalent, the solver returns that the test goal is unsatisfiable. As already noted, equivalent mutants pose a problem to the test generation process: they require computational time (generally proving that a formula is unsatisfiable requires more resources than finding a model) and they do not contribute to the final test suite since they do not produce any distinguishing configuration.

Example 13: Given the model $\mathcal{M}$ and its mutant $\mathcal{M}'$ shown in Fig. 2, the test predicate is

$$\begin{aligned} tp &= \varphi \oplus \varphi' = \text{BOF}(\mathcal{M}) \oplus \text{BOF}(\mathcal{M}') \\ &= (a \to b \wedge b \to a \wedge c \to a) \oplus (b \to a \wedge c \to a) \end{aligned}$$

The models of tp (i.e., $\{a = true, b = false, c = true\}$ and $\{a = true,\ b = false,\ c = false\}$) are distinguishing configurations.

A set of configurations able to detect all the (first order) faults for a given feature model can be generated as follows:

1) generate all the mutants for the original feature model by applying one mutation at a time;
2) generate a test predicate for every mutant;
3) use the solver to get the model for every test predicate (or to prove that the mutant is equivalent).

The process can be even extended in case one wants to find higher order mutants (it suffices to apply the mutation operators several times). However, the naïve algorithm above produces many tests and can be very time consuming. Indeed, it generates a distinguishing configuration for every test predicate and this may be unnecessary, since a configuration may act as distinguishing configuration for two faults and we are in general interested to find a *small* test suite killing all the faults. To the basic process, we can apply several techniques [21], including:

- *monitoring*: it checks whether a test produced for a test predicate is also a model for other test predicates;
- *collecting*: it looks for a model of a conjunction of test predicates, instead of a model for each test predicate. The use of an SMT solver allows to incrementally add test predicates to the context;
- *prioritizing*: it considers the test predicates in a particular order; such technique is useful only if used in combination with monitoring and/or collecting;
- *post reduction*: after the test generation, it removes *unnecessary* tests, i.e., tests that only cover test predicates that are also covered by other tests.

From now on, we assume to use the collecting technique, since, although it requires much computational resources, it is able to produce very compact test suites [15]. Moreover, we devise the following syntactic optimization.

B. Logical XOR simplification

Considering that our test predicates have the form $\varphi \oplus \varphi'$ and that φ and φ' are conjunctions of many common sub-expressions (those that refer to parts of the feature model which are not mutated), we can simplify the test predicate before running the solver in order to reduce the number of conditions (i.e., occurrences of literals). We have used the following equivalence that allows to factor a part of the formula and to push the $\oplus$ operator near the literals:

$$(\alpha \wedge \beta) \oplus (\alpha \wedge \gamma) \equiv \alpha \wedge (\beta \oplus \gamma)$$

where α, β, and γ are predicates.

Example 14 (Simplification of xor expressions): Consider the test predicate shown in Ex. 13 for the models in Fig. 2. The test predicate can be simplified as follows:

$$\begin{aligned} &(a \to b \wedge b \to a \wedge c \to a) \oplus (b \to a \wedge c \to a) \\ &\equiv (b \to a \wedge c \to a) \wedge (a \to b \oplus true) \\ &\equiv (b \to a \wedge c \to a) \wedge \neg(a \to b) \\ &\equiv (b \to a \wedge c \to a) \wedge (a \wedge \neg b) \end{aligned}$$

While the original test predicate has 10 conditions (i.e., occurrences of literals), the simplified version contains only 6 conditions.

This simplification is similar to the *simplified reasoning* introduced in [11] in which CNF clauses are grouped in order to simplify the SAT-checking formulas for feature model edits.

C. Mutants classifications

Following the terminology given in [11], we can classify every mutant in one of four groups, depending on the satisfiability of its test goal.

1) **Refactoring**: if an operator F does not add new products and no existing products are removed, then $\mathcal{M}'$ is an equivalent mutant and the test goal is unsatisfiable. Formally, in a refactoring, $\varphi \leftrightarrow \varphi'$ holds, therefore $\varphi \oplus \varphi'$ is unsatisfiable.
2) **Specialization**: if an operator F removes some existing products but no new products are added, then every distinguishing configuration is a product for $\mathcal{M}$ but not for $\mathcal{M}'$. Formally, $\varphi' \to \varphi$ holds and $\varphi \to \varphi'$ does not hold, therefore $\neg\varphi \wedge \varphi'$ is unsatisfiable while $\varphi \wedge \neg\varphi'$ is satisfiable.
3) **Generalization**: if an operator F adds some products but no existing products are removed, then every distinguishing configuration cannot be a product for $\mathcal{M}$, while it is a product for $\mathcal{M}'$. Formally, $\varphi \to \varphi'$ holds and $\varphi' \to \varphi$ does not hold, therefore $\varphi \wedge \neg\varphi'$ is unsatisfiable while $\neg\varphi \wedge \varphi'$ is satisfiable.
4) **Arbitrary edit**: if an operator F adds some products and removes some other products, then there exists a distinguishing configuration that is a product of $\mathcal{M}$ and another one that is not a product. Formally, neither $\varphi \to \varphi'$ nor $\varphi' \to \varphi$ hold, and therefore both $\neg\varphi \wedge \varphi'$ and $\varphi \wedge \neg\varphi'$ are satisfiable.

Note that, in general, it is not possible to associate a mutation operator with a given mutant class. However, for some particular mutation operators, this is possible: the **OptToMan** operator, for example, will always produce a mutant that is a specialization of the original model (unless there are constraints that produce a false optional).

V. Experiments

As case studies we have taken 53 SPLOT models which are often used as benchmarks. Table II reports the data about the 53 models in terms of number of features and total number of products. We have also included 1600 artificially generated models that are available for download on the FeatureIDE official site; we took 200 models for eight different groups

Table II: SPLOT models properties

	Min	Max	Mean	Median	Sum	Stdev
Features	9	287	31.79	20	1685	43.99
Products	2	4.52×10^{49}	8.53×10^{47}	6.3	4.52×10^{49}	6.21×10^{48}

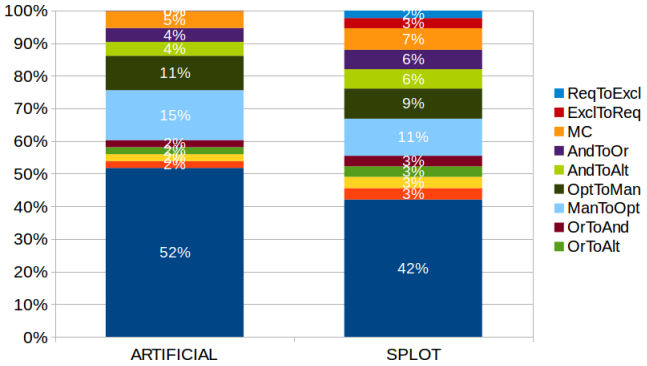


Figure 7: Distribution of mutants for each mutation class

characterized by the number of features: 10, 20, 50, 100, 200, 500, 1000, and 2000 features[2]. All feature models contain cross-tree constraints, growing with the number of features, and they all admit at least one product. The criteria used for their creation are explained in [11].

In the experiments, we apply only first order mutants since we assume here that the *coupling effect* [24] is valid also for feature models; indeed, the coupling effect states that if a test case finds simple faults, it will also find more complex faults.

A detailed discussion about the results of the experiments, guided by a series of research questions (**RQ**), follows.

RQ1 How many mutants are generated?

Fig. 6a and Fig. 6b show the number of mutations depending on the number of features and extra-constraints, respectively for the SPLOT and the artificial models. The number of mutants of a model is around the double of the sum of the number of its features plus the number of its extra-constraints. Indeed, a model can be mutated, for each feature, by the **MF** operator and by another feature-based mutation operator, and, for each constraint, by the **MC** operator and possibly by **ReqToExcl** or **ExclToReq**.

RQ2 How many mutants are generated by each mutation operator?

The distribution of mutations over the mutation operators strictly depends on the topology of the feature model. Fig. 7 shows the distribution of the types of mutations. Almost half of the generated mutations are missing features. This makes our methodology oriented towards the detection of omission errors, which are one of the most frequent errors: There is some evidence, from empirical investigations of software faults, that missing-condition faults are extremely common [22].

[2]Note that the eCos kernel has 1244 features. Therefore, we believe that the size of the selected models represents the size of real-life models.

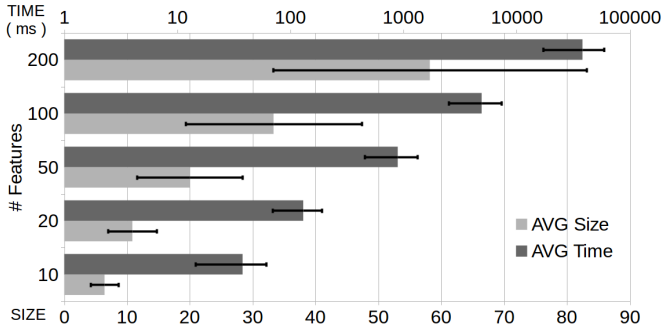


Figure 9: Test-suite generation time and size varying the number of features – Artificial model set

RQ3 How are mutants distributed in the classes presented in Sect. IV-C?

We have grouped all the mutants according to the classification presented in Sect. IV-C and originally proposed in [11]. Fig. 8 shows the results for the SPLOT models and for the artificial models. It is apparent that, while only few mutations of SPLOT models are equivalent (*refactoring* class), the artificial models produce many equivalent mutations, probably because they contain many anomalies (like redundant constraints, dead features, and false optionals [9]). This fact makes the generation of distinguishing configurations less efficient for artificial models than for SPLOT models, since equivalent mutants consume much time without producing useful tests.

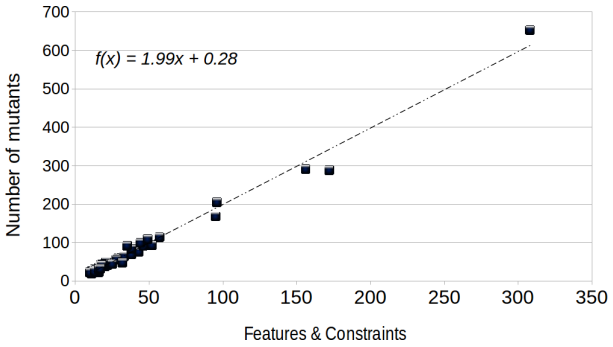
RQ4 Is the simplification of the xor expressions presented in Sect. IV-B effective?

The aim of this experiment is to determinate if the xor simplification introduces an improvement in terms of computation performance. The test suite generation process, performed without and with this simplification, shows an average reduction in terms of time of 46%, and a reduction for the most time consuming test of 49%. This result was expected since this technique reduces the computational load of the SMT solver necessary for the xor resolution.

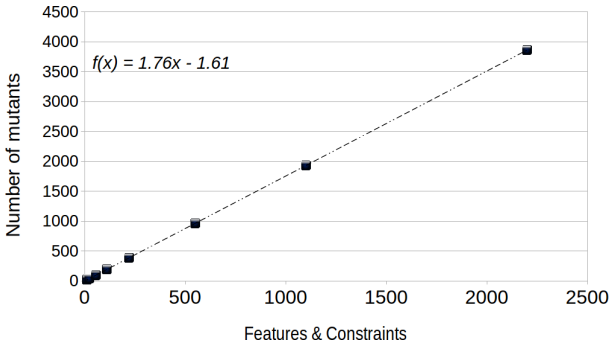
RQ5 How big are the fault-detecting test suites? How much time is required to generate them?

The chart of Fig. 9 shows that the size of artificial models test suites and their generation time increase with the number of features of the model under test. The time presents an exponential increase, while the size increases linearly, and both of them present a great STDEV error. We are able to generate test suites for models up to 200 features in around 38 seconds. For bigger models, our method takes a considerably amount of time and we were unable to generate test within the timeout of 15 minutes.

Table III reports the data related to the test generation for the SPLOT models. The generation of the entire test suite never took more than 138 seconds. During the experiments we have noticed that the size of the test suites for SPLOT models and their generation time do not always increase with

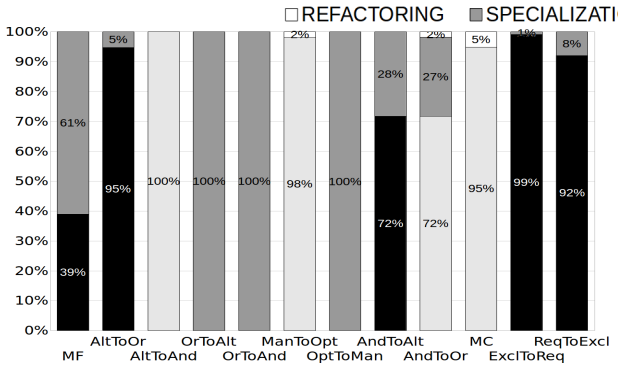


(a) Number of mutants for the SPLOT model set

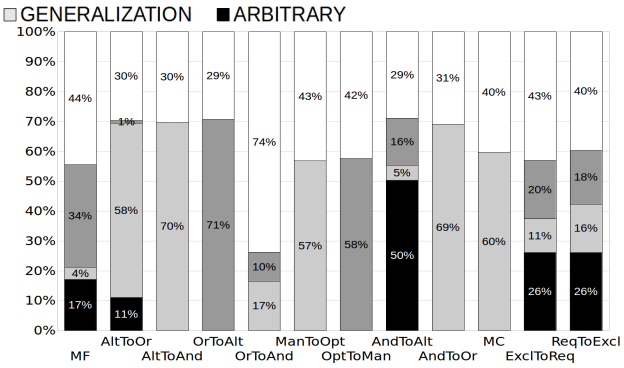


(b) Average number of mutants for each artificial model set

Figure 6: Number of mutants



(a) SPLOT



(b) Artificial models

Figure 8: Distribution of mutants for each model change class

Table III: Test generation data – SPLOT model set

	Size	Time (ms)	#Infeasible
Min	4	27	0
Max	134	137 029	8
Mean	19.9	5264.6	0.49
Sum	1053	279 026	26

Table IV: Correlation between features, products, generation time, and size of the test suites – SPLOT model set

Size vs Features	-0.17	Time vs Features	-0.12	Features vs Products	0.84
Size vs Products	-0.07	Time vs Products	-0.04	Time vs Size	0.83

the number of features. This observation has been confirmed by the Pearson correlation test, whose results are reported in Table IV. According to our opinion, the non-correlation can be related to the high STDEV that affects the observations of both sets but that influences more the SPLOT one due to its small dimension.

Time and size, instead, are mutually correlated in both model sets (as shown in Fig. 9 for the artificial set and in Table IV for the SPLOT set).

RQ6 Are distinguishing configurations useful?

We have performed an experiment in which our method is used to find faults in feature models developed by users. We have taken 6 small programs from the literature regarding the synthesis of feature models from preprocessor directives of C/C++ source code. Each program has from 19 to 38 lines of code containing from 2 to 5 `#ifdef` or equivalent directives. We have instructed 6 students and asked them to build the 6 feature models. The resulting models contain from 2 to 6 features and maximum 2 constraints. They can be validated by generating the distinguishing test suite, compiling the code with the options selected in each configuration, checking if the compiler successfully ends, and comparing the outcome (compiler success or failure) with the expected result. 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). We have compared our technique with three classical techniques: pairwise testing, considering all the products, and considering all the configurations. Table V reports, for each technique, the number of tests, the number of tests that fail (i.e., they identify a fault), the number of detected faulty models, and the ratio between the number of tests and number of detected faulty models. Our approach produces the minimum number of tests and it is able to discover 59% of the faulty models with only 21% of the tests w.r.t. the exhaustive testing (i.e., using all the configurations). By using all the products, only 29% of the faulty models are captured. As

method	#tests (*t*)	#failing tests	#faulty models (*fm*)	*t/fm*
Distinguishing configurations	109	13	10	10.9
Pairwise	165	12	5	33.0
All products	285	23	5	57.0
All configurations	516	119	17	30.4

Table V: Finding faults in models obtained from C/C++ code

Type	Specialization	Generalization	Arbitrary edit
#models	11	3	3

Table VI: Classification of faults in models from C/C++ code

already explained in Sect. III (Ex. 7), using only the products greatly reduces the fault detection capability. Note that using all the products or all the configurations is still practicable in these examples, but in general it is not. Also pairwise testing among the features (i.e., the preprocessor directives) discovers fewer faulty models than our method, and also has less failing tests (but it is at least better than using all the products in terms of test suite size). In terms of number of tests necessary to identify a faulty model (*t/fm*), our approach is much better than the others. Table VI shows the types of errors student models contain. The majority of faults are specializations: students were inclined to add more constraints than those imposed by the compiler according to the preprocessing directives. For instance, the code `#ifdef HAVE_LIBGCRYPT ... #elif defined (HAVE_LIBCRYPTO) ... #endif` was interpreted by the majority of students as an *alternative*, while the compiler correctly accepts both the labels true.

Threats to validity

As sanity check, we have compared our results with the classification produced by the FeatureIDE classifier [11] (see Sect. IV-C): for example, if we classify a mutant as equivalent, FeatureIDE must classify the mutation as refactoring.

As seen in the experiment **RQ5**, the test generation time grows exponentially with the size of the feature model. This fact limits the applicability of our approach. As future work, we plan to devise techniques able to handle bigger models (e.g., generating tests for *independent* sub-trees of the feature model). Note that some limiting policies of the collecting algorithm (e.g., collecting only a subset of the test predicates) can already considerably reduce the test generation time, but at the expense of the test suite size [21].

From our experiments, it is clear that the coupling effect does not hold in our case, since there are faults that we are not able to capture. This could be due to the fact that we have not implemented some mutation operators and/or that we have only considered first order mutants. Applying only first order mutants is probably not enough because the *component programmer hypothesis* (stating that a programmer writes models that are nearly correct) may not hold for our students.

As future work, we plan to consider the use of higher order mutants and of new mutation operators that add features and constraints. Especially the addition of constraints could be useful to find faults in over-constrained models (like most of those of experiment **RQ6**). Note that adding new mutation operators would increase the number of tests only linearly while the number of configurations grows exponentially with the number of features; therefore, we believe that our approach would still be competitive w.r.t. exhaustive testing.

VI. Related Work

Testing for feature models and SPLs is widely studied in literature. Recent surveys [25] count around 50 papers dealing with testing of feature models. Regarding how tests are generated, i.e., products are selected, another recent survey found 19 papers [26]. Surprisingly, the problem of test selection is considered still an open problem [3]. The main challenge is the combinatorial explosion of the number of possible configurations of feature models. All the researchers try to reduce the number of configurations and still preserve some kind of coverage over the feature models under test. A classical test generation approach consists in applying combinatorial interaction testing (CIT) – also called T-wise testing –, to SPLs and feature models [5], [6], [4]. CIT aims to achieve a high coverage of feature combinations with the smallest number of tests as possible. CIT does not focus explicitly on testing feature models, since it is mainly used for finding the faults due to the interaction of features in the products of the SPL.

There are several attempts to use techniques different from combinatorial testing. For instance, in [27] the authors try to determine if a feature is *irrelevant*, i.e., it augments, but does not change, the existing program behavior, making many feature combinations unnecessary as far as testing is concerned. In our case, some combinations are considered unnecessary if they do not contribute to any distinguishing configuration, but we do not consider the implementation of the feature models.

An initial proposal about using mutation and SPL testing can be found in [14]. The authors propose the concept of *small variation* between two models that can be detected by examining the outputs. This would be the basic for building our distinguishing configurations. However, in their approach the test-data generation technique relies on the execution or simulation of systems, it does not start from feature models and, therefore, it generally requires complete instantiated products or components in order to generate successful test data.

Segura et al. [20] proposed using *metamorphic* testing to generate test data for feature model analysis tools. They presented a set of metamorphic relations between input FMs and the set of products they represent. Based on these relations and given one FM and its known set of products, a set of neighboring FMs together with their corresponding set of products are automatically generated. The metamorphic relations that they propose are only additive and they do not want to represent design mistakes but they are introduced with the goal of stressing automated analysis tools. In their approach, tests are mainly transformed feature models, while derived configurations play a secondary role.

As already said, our mutations are a particular case of the 16 edit operations presented in [11] (except **ReqToExcl** and **ExclToReq** which are not considered). However, their goal is to help product line designers judge the impact of their edits, while we automatically apply a set of changes in order to find the distinguishing configurations that are used to validate the original models.

An approach using mutation analysis and feature models is presented in [12]. There are several differences with our approach. First, they exclusively focus on using mutation to evaluate the tests (and not for test generation). The tests are randomly generated from the space of products, mutation analysis is performed to measure the quality of the tests, and they found that the mutation score depends on test suite characteristics (similarity). Moreover, the authors define the mutation operators for the representation of the feature model as propositional formula (the result of our BOF function). However, it is well known that feature models (as presented in Sect. II-A) are not logically complete [28]. So, mutating the formula $\text{BOF}(\mathcal{M})$ is partially complementary to the mutation of the feature model $\mathcal{M}$, since not all the mutations of $\text{BOF}(\mathcal{M})$ may represent mutations of $\mathcal{M}$ and, vice versa, some mutations of $\mathcal{M}$ may be not represented by a simple mutation of $\text{BOF}(\mathcal{M})$. For instance, there is no feature model whose representation as proposition formula includes the negation of the *root* feature.

VII. Conclusions

We have presented a fault-based approach for testing SPLs using feature models. Our technique explicitly targets a set of fault classes in feature models. Exploiting a well known SMT solver, we are able to produce, in an efficient way, test suites with a guaranteed fault detection capability. In our approach, tests are given by *distinguishing configurations*, i.e., configurations able to distinguish a model from its faulty version; distinguishing configurations can be either valid products or not. Experiments show that our approach has a better fault detection capability and requires fewer tests than the approach that uses all the products and the pairwise approach that uses a subset of the products.

References

[1] 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.

[2] M. Mendonca, M. Branco, and D. Cowan, "S.P.L.O.T.: software product lines online tools," in *Proc. of the conference companion on Object oriented programming systems languages and applications (OOPSLA '09)*. New York, NY, USA: ACM, 2009, pp. 761–762.

[3] J. Lee, S. Kang, and D. Lee, "A survey on software product line testing," in *Proceedings of the 16th International Software Product Line Conference-Volume 1*. New York, NY, USA: ACM, 2012, pp. 31–40.

[4] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon, "Automated and scalable T-wise test case generation strategies for software product lines," in *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*. IEEE Computer Society, 2010, pp. 459–468.

[5] M. B. Cohen, M. B. Dwyer, and J. Shi, "Coverage and adequacy in software product line testing," in *Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis*. ACM, 2006, pp. 53–63.

[6] A. Calvagna, A. Gargantini, and P. Vavassori, "Combinatorial testing for feature models using CitLab," in *Int. Workshop on Combinatorial Testing (IWCT)*. IEEE Computer Society, 2013, pp. 338–347.

[7] L. J. Morell, "A theory of fault-based testing," *Software Engineering, IEEE Transactions on*, vol. 16, no. 8, pp. 844–857, 1990.

[8] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *Software Engineering, IEEE Transactions on*, vol. 37, no. 5, pp. 649–678, 2011.

[9] 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.

[10] D. Batory, "Feature models, grammars, and propositional formulas," *Software Product Lines*, pp. 7–20, 2005.

[11] T. Thum, D. Batory, and C. Kastner, "Reasoning about edits to feature models," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, May 2009, pp. 254–264.

[12] 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.

[13] P. Arcaini, A. Gargantini, and E. Riccobene, "Using mutation to assess fault detection capability of model review," *Software Testing, Verification and Reliability*, 2014.

[14] Z. Stephenson, Y. Zhan, J. Clark, and J. McDermid, "Test data generation for product lines – A mutation testing approach," in *Proceedings of the International Workshop on Software Product Line Testing (SPLiT 2004)*, Boston, MA, Aug. 2004, pp. 13–18.

[15] 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.

[16] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena, "Refactoring product lines," in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, ser. GPCE '06. New York, NY, USA: ACM, 2006, pp. 201–210.

[17] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Clelang-Huang, and P. Heymans, "Feature model extraction from large collections of informal product descriptions," Aug. 22 2013.

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

[19] N. Andersen, K. Czarnecki, S. She, and A. Wasowski, "Efficient synthesis of feature models," in *International Software Product Line Conference*. ACM, 2012, pp. 106–115.

[20] S. Segura, R. M. Hierons, D. Benavides, and A. Ruiz-Cortés, "Automated metamorphic testing on the analyses of feature models," *Information and Software Technology*, vol. 53, no. 3, pp. 245 – 258, 2011.

[21] P. Arcaini, A. Gargantini, and E. Riccobene, "Optimizing the automatic test generation by SAT and SMT solving for boolean expressions," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. IEEE, nov. 2011, pp. 388 –391.

[22] D. R. Kuhn, "Fault classes and error detection capability of specification-based testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 8, no. 4, pp. 411–424, Oct. 1999.

[23] B. Dutertre and L. de Moura, "The Yices SMT solver," SRI Available at http://yices.csl.sri.com/tool-paper.pdf, Tech. Rep., 2006.

[24] A. J. Offutt, "The coupling effect: fact or fiction," *SIGSOFT Software Engineering Notes*, vol. 14, pp. 131–140, November 1989.

[25] P. A. da Mota Silveira Neto, I. d. Carmo Machado, J. D. McGregor, E. S. de Almeida, and S. R. de Lemos Meira, "A systematic mapping study of software product lines testing," *Information and Software Technology*, vol. 53, no. 5, pp. 407–423, May 2011.

[26] I. do Carmo Machado, J. D. McGregor, and E. Santana de Almeida, "Strategies for testing products in software product lines," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, p. 1, Nov. 2012.

[27] C. H. P. Kim, D. S. Batory, and S. Khurshid, "Reducing combinatorics in testing product lines," in *Proc. of the 10th Int. Conference on Aspect-oriented Software Development (ASOD 2011)*. ACM, 2011, pp. 57–68.

[28] 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.