Testing the Evolution of Feature Models with *Specific* Combinatorial Tests

Andrea Bombarda
University of Bergamo
Bergamo, Italy
andrea.bombarda@unibg.it

Silvia Bonfanti
University of Bergamo
Bergamo, Italy
silvia.bonfanti@unibg.it

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

***Abstract*—Software Product Lines (SPLs) are commonly used for representing highly configurable systems, by using Feature Models (FMs). Like any other systems, SPLs (and consequently FMs) undergo changes over time, and developers may inadvertently introduce errors during this process. Consequently, it is of utmost importance to rigorously test SPLs during their evolution, in particular, by analyzing the changes introduced from one version of FM to its evolved counterpart.**

This paper presents the concept of "specificity", enabling testers to discern tests specifically tailored for evaluating a model evolution. In addition, we present SPECGEN, a BDD-based algorithm that aims at producing more specific combinatorial test suites. We conduct a comparative analysis between SPECGEN and established approaches. Our experiments demonstrate that employing SPECGEN results in test suites with higher specificity, maintaining the same size, and accomplishing this in a shorter time compared to conventional methods.

***Index Terms*—Software Product Lines, Feature Model, Combinatorial Testing, Specificity, Evolution**

I. Introduction

As with all software models and artifacts, a software product line (SPL) evolves over time for various reasons and the feature model (FM) representing that SPL can change as well. Feature model evolution has been extensively studied, however, little attention has been given to how other artifacts connected to the FM could or should evolve. In this paper, we focus on (valid) products derived from a FM to be used as test cases for the SPL under test. We assume that the designer of a SPL, once a FM has been designed for it, is interested in deriving a set of products, i.e. valid configurations, according to some criteria (for instance combinatorial interaction among features). Those products are then used to test the SPL (for example by actually building the products and checking their correctness) and, for this reason, we consider that set of products as a test suite. If the SPL, together with its FM, is modified, the original test suite likely must be modified, since some products are no longer valid while others must be considered due to the modifications of the FM. In [6] the author proposed an approach for reusing the old test suite and outlined the advantages of their approach in terms of diversity, size, and time. However, they do not consider how much a test suite fits the modifications, i.e., the efficacy of a test suite upon a feature model change to directly test them.

The first objective of this paper is to introduce the concept of "specificity" of a test suite, with the objective to measure the capability of a test suite to specifically test edits applied to a feature model. The underlying idea is simple: when a feature model is modified, it would be better that the new tests in the test suite were specifically introduced to test those modifications. The second goal of the paper is to devise and experiment a technique that tries to maximize the specificity of a test suite without decreasing the desired coverage (combinatorial in our case).

In detail, the contributions of our paper are the following ones. Given an arbitrary evolution e of a FM of the SPL under test, we introduce:

1) a formal definition of *specificity* that leads to a simple way to identify which tests are *specific* for e,
2) a measure for a test suite of its specificity for e as a quality measure, besides its size and coverage,
3) a Binary Decision Diagrams (BDD)-based algorithm that generates combinatorial interaction test suites for a FM,
4) a BDD-based algorithm that generates combinatorial interaction test suites that besides reaching the desired coverage, aim at being more specific for e.

We have compared our implementation, called SPECGEN (SPECific tests GENerator), with other more established approaches such as ACTS [31], a more classical BDD-based generation method (BDDGEN), the one implemented by FeatureIDE developers (INCLING), and GFE [6]. Our experiments show that our methodology allows practitioners to identify the specific tests in normal test suites and to increase the specificity of a test suite, with a reduction of the time required for test generation but without impacting on the size of the test suite.

The remainder of the paper is structured as follows. Sect. II reports the background on FMs, their evolution, and the way in which FMs can be represented using BDDs, while, in Sect. III, we give the definition of specificity for tests and test suites. Sect. IV presents our BDD-based approach for generating combinatorial test suites for FMs and its extension aiming at generating more specific test suites for FM evolutions. In Sect. V, we report the experiments we have performed to evaluate the proposed approach, while in Sect. VI potential threats to the validity of our findings are discussed. Finally, Sect. VII and Sect. VIII, respectively, report related work on testing feature models with tests designed for FM evolution and conclude the paper.

II. Background

This section provides an overview of basic concepts related to feature models, their evolution and testing, as well as how BDDs can be employed to represent FMs and their valid configurations.

A. Feature Models

In the field of software product line engineering, feature models [16], [26] serve as models that delineate all potential products within a software product line (SPL) based on their features and their interrelationships. To be precise, a feature model (denoted as FM) comprises a structured collection of features F, organized in a hierarchical manner. Each parent-child relationship within this hierarchy imposes a constraint falling into one of the subsequent categories: * *Or*: at least one of the sub-features must be selected if the parent is selected. * *Alternative*: exactly one of the children must be selected whenever the parent feature is selected. * *And*: if the relation between a feature and its children is neither an *Or* nor an *Alternative*. Each child of an *and* must be either: **–** *Mandatory*: the child feature is selected whenever its respective parent feature is selected. **–** *Optional*: the child feature may or may not be selected if its parent feature is selected.

Only the *root* feature in F has no parent and it is selected in every product. In addition to hierarchical relations, feature models may have *cross-tree constraints*, i.e., relations that cross-cut hierarchy dependencies. The most common cross-tree constraints are:

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

As common in the literature on FMs, in this work, we allow FMs to contain cross-tree constraints given by *general* propositional formulas. Furthermore, a feature can be *dead*, i.e., it can never be selected because of the constraints, or *core*, i.e., it is mandatory in every valid product, like the *root*.

B. Evolution and types of edits of a feature model

Both SPLs and their feature models undergo evolution [27] during their lifetime. This evolution includes actions such as adding, removing, relocating features, and modifying, adding, or removing constraints. Even minor alterations lead to altering the range of eligible feature combinations: previously valid configurations might no longer hold, while others could now become valid.

If the added/removed products could be easily identified, one could manually design validation activities focused to those products. However, changes of valid configuration sets are known to be impractical to determine manually, especially because the type of edits occurring during FM evolution is known to be arbitrary.

Several examples of feature model evolutions coming from the industry or other research studies are available in the literature (see Sect. V, where we use them as case studies for our experiments).

C. Tests for feature models

When working with a feature model $\tilde{FM}$, each (abstract) test, also called *configuration*, specifies which features in $\tilde{F}$ (the feature set of $\tilde{FM}$) are selected and which are not. More formally, in this work, we consider a test t as a function that returns the status of a feature $f \in \tilde{F}$ in t

$$t(f) = \begin{cases} \top(true) & \text{if } f \text{ of } \tilde{FM} \text{ is selected in t} \\ \bot(false) & \text{if } f \text{ of } \tilde{FM} \text{ is not selected in t} \end{cases}$$

In the following, in order to have a more compact notation, instead of using $t(f) = \top$ we will use $t = \{f \rightarrow \top, \dots\}$ and instead of using $t(f) = \bot$ we will use $t = \{f \rightarrow \bot, \dots\}$.

A configuration or test is valid in a FM if and only if it denotes a valid product, i.e., a configuration that does not violate any constraint in the FM.

Test cases for feature models can be generated following several criteria. Among those, in this paper, we focus on generating test suites achieving the t-wise combinatorial coverage, one of the most used [28]. More specifically, in this paper, we aim to achieve the pairwise coverage, i.e., we consider $t = 2$. Considering all the features in FM, we generate a test suite TS covering all the t-way interactions among them.

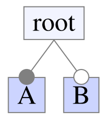
D. BDD representation of a FM

An efficient way to represent a FM is by means of Binary Decision Diagrams (BDDs). They are widely used within the domain of system design verification since they can be easily extended to represent tests. A BDD corresponds to a compressed form of a decision diagram:

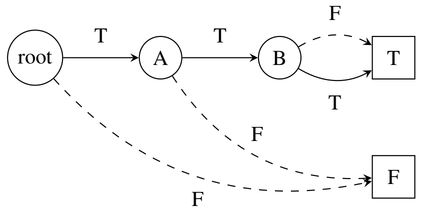
Definition 1 (Binary Decision diagram)**.** Let $B = \{F, T\}$ be the Boolean domain. A *binary decision diagram* is a graph that represents a function f over n Boolean variables: $f : B^n \rightarrow B$, such that every path depicts an assignment to each of the variables under which the function f returns the boolean value of the terminal.

A binary decision diagram can be used to evaluate the truth value of f when it is applied to the variables $x_1, \cdots, x_n$. Since FMs can be seen as models describing which configurations (i.e., truth assignments to all the features) are acceptable and which are not, a BDD can represent the set of possible products of a FM.

Typically, among BDDs, it is possible to perform unary operations such as *complement*, computation of the *cardinality*, or the most classical binary operations like *union*, *intersection*, and *difference*. In particular, since BDDs can represent logic functions, operations among BDDs are equivalent to logic operations. Given a BDD B representing the function f, its complement $\neg B$ represents the function $\neg f$. The union between two BDDs $B_1 \vee B_2$ represents the function $f_1 \vee f_2$. The intersection between two BDDs $B_1 \wedge B_2$ represents the function $f_1 \wedge f_2$. Finally, given a BDD B, its cardinality (or size) $|B|$ represents the number of all the possible paths

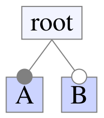


(a) FM

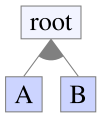


(b) BDD

Fig. 1: Correspondence between FM and BDD



(a) FM



(b) FM'

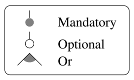


Fig. 2: Example of FM evolution with same feature set

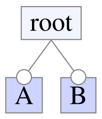
	root	A	B
t_1	$\top$	$\top$	$\bot$
t_2	$\top$	$\top$	$\top$

(a) TS for FM

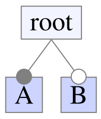
	root	A	B
t'_1	$\top$	$\top$	$\bot$
t'_2	$\top$	$\top$	$\top$
t'_3	$\top$	$\bot$	$\top$

(b) TS for FM'

TABLE I: Pairwise test suites for the feature models in Fig. 2



(a) FM



(b) FM'

Fig. 3: Example of FM evolution narrowing down the set of valid configurations

leading to the terminal node `T`. The value of cardinality of a BDD can be used to check the consistency between Boolean functions, i.e. if $f_1(x)$ and $f_2(x)$ are inconsistent, the intersection between the BDDs representing the two functions is empty.

Considering the feature model shown in Fig. 1a, which has a *root*, a *mandatory* feature (A) and an *optional* one (B), the corresponding BDD is shown in Fig. 1b. In this particular representation, dashed lines indicate that the feature from which the arrow starts is *unselected*, while continuous lines represent *selected* features.

As previously introduced, after having represented the FM as a BDD, it is possible to derive test cases by simply enumerating which paths lead to the `T` leaf. Similarly, BDDs can be easily used to model the tuples to be covered. This is done by forcing the assignments contained in the tuple tp as leading to the `T` leaf, and all the others to the `F` leaf.

Despite the majority of the approaches dealing with FMs adopt solver-based technique to generate test cases, in this work we use BDDs. Indeed, there are several works in the literature confirming that decision diagrams show better performance than logical solvers [15].

III. Definitions

Considering the FM evolution process as introduced in Sect. II-B, we define when a test case t is specific for a model evolution.

Definition 2 (Specific Test)**.** Given the evolution of the feature model FM to FM' with the same feature set F, we say that a test t is *specific* if and only if t is valid for FM' and it is not in FM.

Let's consider the example shown in Fig. 2 and the corresponding test suites reported in Tab. I, the test $t_1 = t'_1 = \{root \rightarrow \top, A \rightarrow \top, B \rightarrow \bot\}$ is *non specific* since it represents a valid product both in FM and FM'. Instead, the test $t'_3 = \{root \rightarrow \top, A \rightarrow \bot, B \rightarrow \top\}$ is *specific* because this configuration is valid in FM' and not in FM, thus it tests the changes occurred during the model evolution.

Our Def. 2 of specificity implies that it is asymmetric:

Property 1 (Asymmetry)**.** *Given two feature models FM_1 and FM_2, if a test t is specific for the evolution of FM_1 into FM_2, then it is not specific for the evolution of FM_2 into FM_1.*

One may argue that also tests valid in FM and not in FM' are specific to test the model evolution. This is indeed true because a test that became invalid could be considered specific to test the evolution. However, since it is now invalid in FM' we assume that it is useless and represents a configuration of a product that cannot be actually built and tested. Considering the asymmetry property, for particular model evolutions, it may be possible that no specific test exists, as formalized by the following property:

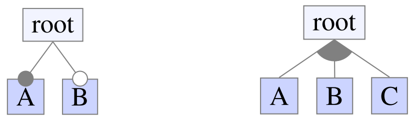
Property 2 (Absence of specific tests)**.** *Let FM and FM' be two feature models, and be FM' the evolved version of FM. Let T_{FM} be the set of valid configurations of FM and $T_{FM'}$ be the set of valid configurations of FM'. If $T_{FM'} \subseteq T_{FM}$, then no test specific for the evolution exists.*

This characteristic stems from the observation that when an evolved feature model merely narrows down the range of valid configurations (such as in the case of Fig. 3), all conceivable tests are valid in the previous version too.

A. Specific tests when the feature set changes

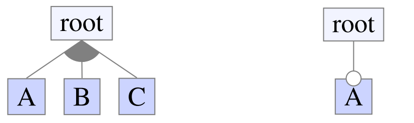
When a FM evolves, arbitrary edits may happen [27]. One of the most common model evolution is represented by adding or removing features. In that case, taking test cases (or possible configurations) derived from FM' and evaluating them over FM can be challenging since the feature set changes, and it is not trivial to define whether a test case is specific or not. In the following, we present how to deal with feature set changes in two different scenarios, i.e., when the feature set is extended and when it is reduced.

The first scenario, i.e., the **extension** of the feature set, is exemplified by Fig. 4. In particular, FM' adds a feature



(a) FM (b) FM'

Fig. 4: Example of FM evolution extending the feature set



(a) FM (b) FM'

Fig. 5: Example of FM evolution reducing the feature set

C to the set of features already available in FM. In this case, the evaluation of tests generated from FM' is straightforward: new features are ignored in the evaluation of the specificity, since they are added to test the changes done in FM' but they are not present in FM. Suppose having a test $t_1 = \{root \rightarrow \top, A \rightarrow \top, B \rightarrow \bot, C \rightarrow \bot\}$, which actually represents a valid product for FM'. We ignore the feature C which has been added specifically for FM' and t_1 is valid for FM as well. For this reason, the test is not specific. Then, if we consider the test $t_2 = \{root \rightarrow \top, A \rightarrow \bot, B \rightarrow \bot, C \rightarrow \top\}$, it is a valid product for FM'. However, if we ignore the feature C, which is not present in FM, then the *restriction* of t_2 on the features F, i.e., $\bar{t_2} = \{root \rightarrow \top, A \rightarrow \bot, B \rightarrow \bot\}$ does not represent a valid configuration for FM. For this reason, t_2 is valid for FM' but it is not for FM and t_2 can be considered as a specific test.

The second scenario occurs, instead, when the feature set is **reduced**, i.e., when FM' removes some of the features present in FM, such as in the example reported in Fig. 5. In this case, features that are removed by FM' must be considered unselected when evaluating the specificity, because those features in FM' must be considered absent. Thus, the test $t_3 = \{root \rightarrow \top, A \rightarrow \top\}$ is not specific, since it is *extended* to $\bar{t_3} = \{root \rightarrow \top, A \rightarrow \top, B \rightarrow \bot, C \rightarrow \bot\}$, which is valid also for FM. Instead, $t_4 = \{root \rightarrow \top, A \rightarrow \bot\}$ is a specific test, since its *extension* $\bar{t_4} = \{root \rightarrow \top, A \rightarrow \bot, B \rightarrow \bot, C \rightarrow \bot\}$ is not valid for FM.

This means that to evaluate a test for its specificty, a test generated for FM' must be first extended over all the features $F \cup F'$ and it must satisfy the following definition.

Definition 3 (Specific Test)**.** Given the evolution of the feature model FM to FM' with possibly different feature sets F and F', we say that a test t is *specific* if and only:

1) its restriction over F' is valid for FM'
2) its restriction over F is not valid for FM
3) $t(f)$ is false for every $f \in F \backslash F'$ (i.e., for every feature removed in FM')

B. *Specificity of a test suite*

Given two test suites, to understand which test suite allows users to better test the evolution, we can introduce the absolute specificity of a test suite starting from the definition of *specific test* (see Def. 2).

Definition 4 (Absolute specificity of a test suite)**.** Given two feature models FM and FM', being FM' the evolved version of FM, and being TS the test suite for FM', the *absolute specificity* of TS is the number of specific tests in TS:

$$Spec_{abs}(TS) = \|\{t | t \in TS \ \wedge \ isSpec(t, FM, FM')\}\|$$

where $isSpec(t, FM, FM')$ is a function evaluating if t is a specific test for the evolution from FM to FM'.

The absolute specificity of a test suite may sometimes be misleading, since it may favor having bigger test suites instead of smaller ones. Thus, we introduce the *relative* specificity.

Definition 5 (Relative specificity of a test suite)**.** Given two feature models FM and FM', being FM' the evolved version of FM, and being TS the test suite for FM', the *relative specificity* of TS can be computed as follows:

$$Spec_{rel}(TS) = \frac{Spec_{abs}(TS)}{\|TS\|}$$

IV. Generating Specific Tests

In this section, we first report the BDD-based generation technique for generating combinatorial test suites from a feature model and, then, we devise a method aiming to generate test suites with a higher specificity. As previously introduced in Sec. II-D, our approach is based on representing feature models in their corresponding BDD and on deriving from it a test suite that maximizes its specificity.

A. *BDD-based generation of combinatorial tests*

The BDD-based procedure generating combinatorial test suites from a feature model is depicted in Algorithm 1 and implemented in the BDDGEN tool. It takes as input the feature model FM for which users want to generate the test suite together with the set of all the possible tuples of features TP. Initially, thanks to the functionalities offered by the `ctwedge` environment, FM is automatically translated into a combinatorial model. In this way, the model can be used to generate a combinatorial test suite with all CT test generators. Then, all tuples tp are fetched (line 2) and the *collecting* procedure starts until all the tuples in TP are checked. The process, repeated for every tuple, is very straightforward.

For every tuple, the function `tryToCover` is called (line 3). This function (reported in Algorithm 2) tries to cover the tuple tp using one of the tests in TS, if possible (line 3-6 in Algorithm 2), or to create a new test starting from bdd_{notp} (line 8-11 in Algorithm 2). In the former case, given t_{bdd} the BDD corresponding to the test that can cover tp, t_{bdd} is updated to its intersection with the BDD representing the tuple tp, i.e., with tp_{bdd} (line 4 in Algorithm 2). In the latter, the test suite TS is enriched with a new BDD

Algorithm 1 Algorithm for the BDD-based generation of combinatorial test suites for a feature model

```
Input: $FM$ the feature model
Input: $TP$ the set of all the tuples derivable from $FM$
Output: the test suite
    ▷ Set of BDDs from which tests can be derived
 1: $TS \leftarrow \emptyset$
 2: for all $tp \in TP$ do                    ▷ Iterate over all the tuples
        ▷ Try to cover the tuple $tp$
 3:     TRYTOCOVER($bdd(tp)$,$TS$,$bdd(FM)$)
 4: end for
 5: return $TS.forEach().getTestCase()$
```

Algorithm 2 Function trying to cover a tuple

```
Input: $tp_{bdd}$ the BDD of tuple desired to cover
Input: $TS$ the set of existing BDDs
Input: $bdd_{notp}$ the bdd when no tuple is committed
Output: $true$ iff a test covering $tp$ is found or generated
 1: function TRYTOCOVER($tp$, $TS$, $bdd_{notp}$)
 2:     for all $t_{bdd} \in TS$ do
 3:         if $size(t_{bdd} \wedge tp_{bdd}) \neq 0$ then     ▷ Can $t$ cover $tp$?
 4:             $t_{bdd} \leftarrow t_{bdd} \wedge tp_{bdd}$
 5:             return true
 6:         end if
 7:     end for
        ▷ Can $tp$ be covered by a new test?
 8:     if $size(bdd_{notp} \wedge tp_{bdd}) \neq 0$ then
 9:         $TS \leftarrow TS \cup \{bdd_{notp} \wedge tp_{bdd}\}$
10:         return true
11:     end if
12:     return false
13: end function
```

corresponding to the test derived from the intersection among bdd_{notp} and the BDD representing the tuple tp, i.e., with tp_{bdd} (line 9 in Algorithm 2). Note that, in this way, `tryToCover` implements the *monitoring* strategy as introduced in [8] which consists in checking if a test generated for a set of tuples accidentally covers other tuples as well, thus reducing the size of the final test suite.

At the end of the collecting process, after having iterated over all the tuples, TS represents the set of BDDs from which the test suite we are looking for can be derived. Thus, from each BDD, we simply extract a test (which is one of the paths leading to the `T` leaf) and obtain a test suite covering all the feasible tuples and containing only test cases complying with the constraints of FM.

B. BDD and features removal and addition

In Sect. IV-A we have presented our approach for deriving a combinatorial test suite containing possible configurations or tests from the BDD representing the valid products of a FM. In order to adapt this approach to the generation of test suites for evolving FMs, we need to handle multiple BDDs, i.e., of both FM and FM', and to perform operations among them. However, as explained in Sect. III-A, the feature set may change during model evolution and this may make the operations more challenging.

In particular, when an evolved model removes some feature, it may be difficult to perform operations between its BDD and that of the original version, since the sets of features differ. For this reason, we introduce the concept of *Completed BDD*, which is a BDD containing all the features of both models.

Definition 6 (Completed BDD)**.** Let FM and FM' be two feature models, with FM' representing the evolved version of FM. Let F and F' be the set of features, respectively, of FM and FM'. The *Completed BDD* is the BDD for FM' over the complete set of features $F \cup F'$ and can be defined as:

$$bdd_c(FM') = bdd(FM') \bigwedge_{f \in F \setminus F'} bdd(f = false)$$

The completed BDD forces every f which has been removed by FM' (i.e., which is in F but not in F') to be unselected. In this way a test generated by using $bdd_c(FM')$ can be correctly checked for validity for FM, as explained in Sect. III-A in Def. 3. In the following, we will always use the *Completed BDD* for FM' in order to keep the tests comparable with the original version of the feature model FM. Note that a test derived from the *Completed BDD* can either be specific or not.

C. Generation of high-specificity combinatorial test suites

Algorithm 3 reports the process we follow for the generation of test suites with high specificity, which is implemented in the SPECGEN tool. It extends the BDD-based procedure previously shown in Algorithm 1.

In order to generate a new test suite and maximize the specificity, both feature models FM and FM' are given as inputs to our procedure. Moreover, the set of all tuples to be covered (i.e., those that can be derived from FM') is required. Then, the procedure computing the specific test suite can start.

First, $bdd_{initial}$ (line 1) is computed by intersecting the negation of $bdd(FM)$ and $bdd_c(FM')$. This BDD represents all the possible specific tests we are looking for. As introduced in Sect. III, an evolution of the model FM to FM' may lead only to non-specific test cases (e.g., when the set of valid products is merely restricted). For this reason, the size of $bdd_{initial}$ is evaluated at line 2. If the cardinality is 0, we can skip the search of specific test cases, since they do not exist, and focus only on non-specific ones. The set-up phase is concluded with lines 7 and 8 that initialize the set of specific and non-specific tests with the empty set.

Then, the process *collecting* all the tuples tp in TP (line 9-19) is performed. First, tp is checked against FM' in order to verify whether tp represents a valid assignment for products derived from FM'. This is performed by computing the size of the intersection between the BDD representing tp, i.e., $bdd(tp)$, and the one of FM', i.e., $bdd_c(FM')$, at line 10. As previously done for checking the existence of specific tests, if the size is 0, the tuple cannot be covered by any valid product and it is skipped. On the other hand, if the tuple can be covered and specific tests can be generated, the `tryToCover` function is called (line 14). In the case in which `tryToCover`

Algorithm 3 Algorithm generating specific combinatorial TSs

```
Input: $FM$ the original feature model
Input: $FM'$ the evolved feature model
Input: $TP$ the set of all the tuples derivable from $FM'$
Output: the specific test suite
     ▷ Initial BDD from which specific tests can be derived
 1: $bdd_{initial} \leftarrow \neg bdd(FM) \wedge bdd_c(FM')$
     ▷ $FM'$ only restricts the valid products
 2: if $size(bdd_{initial}) = 0$ then
 3:     $skipSpecific \leftarrow$ true
 4: else
 5:     $skipSpecific \leftarrow$ false
 6: end if
 7: $T_s \leftarrow \emptyset$                    ▷ Set of BDDs for specific tests
 8: $T_{ns} \leftarrow \emptyset$                 ▷ Set of BDDs for non-specific tests
 9: for all $tp \in TP$ do                ▷ Iterate over all the tuples
        ▷ Check the validity of the tuple
10:     if $size(bdd(tp) \wedge bdd_c(FM')) = 0$ then
11:         continue next $tp$
12:     end if
13:     if $\neg skipSpecific$ then
            ▷ Look for a specific test that can cover $tp$
14:         if TRYTOCOVER($bdd(tp)$,$T_s$,$bdd_{initial}$) then
15:             continue next $tp$
16:         end if
17:     end if
        ▷ No specific test can cover $tp$
18:     TRYTOCOVER($bdd(tp)$,$T_{ns}$,$bdd_c(FM')$)
19: end for
20: $TS \leftarrow T_s \cup T_{ns}$
21: return $TS.forEach().getTestCase()$
```

succeeds in covering *tp*, the next tuple is analyzed. Otherwise, the `tryToCover` function is executed over the set of non specific test cases (line 18).

At the end of the collecting process, during which the iteration over all the possible tuples is performed, the union between the BDDs representing specific tests T_s and non specific tests T_{ns} is computed (line 20). Thus, from each BDD in TS, we simply extract a test case (which is one of the paths leading to the `T` leaf) and obtain the test suite maximizing the specificity we are looking for. The t-wise coverage is assured-by-construction and the final test suite only contains tests representing valid products for FM'. Note that the maximization of the specificity is pursued by how the algorithm is composed. First, it tries to cover each tuple with a specific test, then, if no specific test can cover that tuple, a non-specific one is created. In this way, the number of non-specific tests is minimized.

V. Experiments

In this section, we evaluate the proposed approach against benchmark models available in the literature and with respect to alternative test generation approaches. In Tab. II we report the list of the FMs we use for conducting our experiments. They have been selected by analyzing the literature and keeping those used as case study by other relevant works and having at least one evolution step. Note that, for the models having more than a single evolution, we analyze evolutions between any step, in any order (i.e., we test the version $v1$ against $v2$, then $v1$ with $v3$, and so on, but also $v2$ against $v1$, then $v3$ with $v1$, and so on). In this way, we analyze the model evolution in both directions, i.e., when the model gets more complex between evolutions and when it is simplified. In total, we consider 102 FM evolutions. The replication package, containing the FMs, the evaluation script, the results of our experiments, and all source code for repeating the experiments, is available online at https://github.com/fmselab/ctwedge/tree/master/featuremodels.specificity.

Considering the benchmarks reported in Tab. II, we generate a test suite for each model evolution by using different tools, namely, ACTS [31] (one of the most powerful and used combinatorial test generators), INCLING (which is integrated into FeatureIDE for combinatorial product sampling) [1], BDDGEN (as introduced in Sect. IV-A), SPECGEN (as presented in Sect. IV-C), and the GFE technique presented in [6]. Note that ACTS does not support natively feature models, so we had to translate them in ACTS models including also the (tree and cross-tree) constraints. By instrumenting every generator, we compute relevant measures of each generation run. In particular, we consider the generation time, test suite size, and the relative specificity. All the experiments have been repeated 10 times in order to reduce the influence of non-deterministic timing, on a PC using an Intel(R) i7-8700 CPU @ 3.20GHz with 16 GB RAM.

The results obtained by each test generation strategy have been compared, across all FM evolutions, by using the Wilcoxon Signed-Rank test [30], a general test comparing the distributions in paired samples which does not require data to be normally distributed. Given x the measure to be compared between the two techniques, the Wilcoxon Signed-Rank test is performed using a significance level $\alpha = 0.05$ and with a null hypothesis H_0 stating that the distributions of x in the two techniques are equal. Moreover, in order to verify whether the conclusions drawn from the Wilcoxon test were significant, we have evaluated the effect size by computing the Cliff's delta δ_c. The effect of a technique is small if $|\delta_c| < 0.147$, medium if $0.147 \leq |\delta_c| < 0.33$ or large if $|\delta_c| \geq 0.33$.

We report the main results of the experiments in Tab. III, Tab. IV, and Tab. V. All tools in the tables are sorted starting from the best performing one to the worst one in terms of the considered measure. We report, for each tool T_i, the comparison with the "nearest" competitor T_j for which the test confirms the better performance of T_i w.r.t. T_j (✓ in the *confirmed* column), i.e., for which H_0 can be discarded. When T_i and T_j are not neighbors, we also report the comparison between T_i and all other T_k, with $i < k < j$ and we mark those comparisons with X in the *confirmed* column, meaning that H_0 cannot be discarded.

By applying the previously explained experimental methodology, in the following, we answer the following research questions. RQ1 and RQ2 focus on comparison among state-of-the-art test generators and BDDGEN, RQ3 compares SPECGEN with other tools in terms of specific test cases generation, and finally, RQ4 compares the GFE approach with others previously analyzed. RQs are organized such that, in this

TABLE II: List of the FM evolution examples from the literature – the number of versions (V), the minimum and maximum number of features (including the dead and core ones) across all model evolutions (#F), the minimum and the maximum number of products (#P), and the reference to the paper the model comes from.

Example	V	#F	#P	Ref.	Example	V	#F	#P	Ref.	Example	V	#F	#P	Ref.
AmbAssistLiving	2	24-32	$9.8{\cdot}10^4$-$5.0{\cdot}10^7$	[14]	AutomotiveMult.	3	6-13	5-192	[23]	BCS	3	13-17	128-768	[18]
Boeing	3	5-6	2-2	[29]	CarBody	4	6-13	4-40	[19]	ERP	2	42-57	$2.6{\cdot}10^4$-$2.6{\cdot}10^5$	[24]
HelpSystem	2	25-26	672-10^3	[32]	Linux (Simple)	3	5-10	7-33	[17]	MobileMedia	6	11-26	2-272	[13]
ParkingAssistant	5	6-16	1-32	[10]	Pick&PlaceUnit	9	5-11	3-81	[11]	SmartHome	2	38-61	$9.0{\cdot}10^5$-$3.9{\cdot}10^9$	[22]
SmartHotel	2	6-8	6-30	[4]	Smartwatch	2	12-15	96-192	[2]	WeatherStat.	2	22-23	528-660	[20]

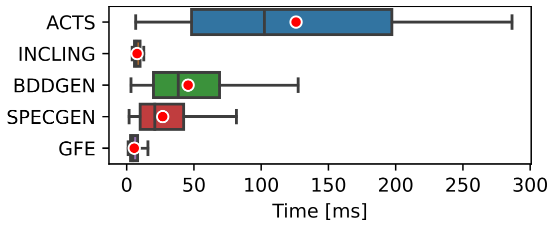


Fig. 6: Generation time analysis

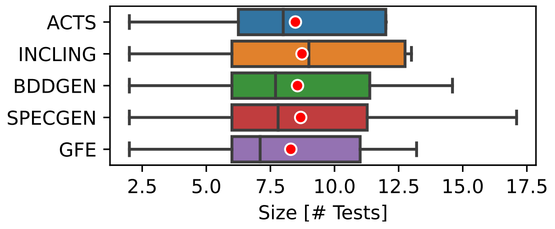


Fig. 7: Test suite size analysis

TABLE III: Comparison among techniques and tools in terms of generation time

Tool	$\bar{t}$ [ms]	Faster than	Statistical test results Confirmed?	p-value	$\lvert\delta_c\rvert$
GFE	5.63	INCLING	✓	$3 \cdot 10^{-15}$	0.61
INCLING	7.75	SPECGEN	✓	$2.6 \cdot 10^{-14}$	0.59
SPECGEN	26.78	BDDGEN	✓	$5 \cdot 10^{-15}$	0.61
BDDGEN	45.69	ACTS	✓	$3 \cdot 10^{-15}$	0.61
ACTS	125.94				

TABLE IV: Comparison among techniques and tools in terms of size

Tool	$\bar{s}$	Smaller than	Statistical test results Confirmed?	p-value	$\lvert\delta_c\rvert$
GFE	8.30	ACTS	✓	$0.2 \cdot 10^{-1}$	0.18
ACTS	8.48	INCLING	✓	$0.9 \cdot 10^{-2}$	0.20
		BDDGEN	X	0.7	0.03
		SPECGEN	X	0.4	0.07
BDDGEN	8.56	SPECGEN	✓	$0.3 \cdot 10^{-1}$	0.17
SPECGEN	8.68	INCLING	X	0.1	0.13
INCLING	8.73				

way, we first analyze the performance of the approach based on BDDs against the state-of-the-art ones and we investigate whether not focusing on generating specific test suites still implies having a good degree of specificity. Then, we analyze the evaluate the performance of the tool aiming to maximize the specificity. Finally, we investigate whether reusing tests cases from the original version of the FM impacts on the obtained specificity.

RQ1 Is the BDD-based test generation competitive against other state-of-the-art test generator tools?

In this research question, we are interested in comparing more traditional approaches, such as ACTS or INCLING, with the BDD-based one we present in this paper, implemented in BDDGEN. In particular, we compare different generation techniques as normally done in combinatorial testing, by analyzing the generation time (see Fig. 6) and the test suite size [7] (see Fig. 7).

In terms of generation time, in Fig. 6, we can see that exploiting BDDs for generating test cases for FMs is slower than the approach implemented in INCLING but faster than ACTS. Instead, in terms of test suite size, ACTS, and BDDGEN are the tools producing the smallest test suites among the analyzed tools, while INCLING generates the highest number of test cases. The Wilcoxon Signed-Rank test we report in Tab. III and Tab. IV, respectively, for the test suite generation time and test suite size confirms these observations. Concerning the generation time, the test confirms INCLING being the fastest test generation method among the tools analyzed in this RQ, with BDDGEN the average one, and ACTS the slowest one. All three tests are significant (i.e., with a p-value lower than α) and statistically relevant with a large effect size. On the other hand, for what concerns the test suite size, the only significant and statistically relevant test is the one between ACTS and INCLING, which confirms our preliminary observation of ACTS being the tool producing the smallest test suite. However, the difference in terms of test suite size is very limited among all the three considered tools.

In conclusion, we can state that the BDD-based test generation approach performs comparably to other test generation strategies, in terms of test suite size, while it is slower than INCLING but faster than ACTS.

RQ2 How much specific are the tests generated with general-purpose tools?

In Sect. III we have introduced the specificity which is an aspect that can be evaluated in every test suite when a FM evolves. In this research question, we are interested in evaluating the specificity of the test suites produced by general-purpose tools, i.e., those analyzed in RQ1.

TABLE V: Comparison among techniques and tools in terms of specificity

Tool	$\tilde{m}$ [%]	Higher than	Statistical test results Confirmed?	p-value	$\lvert\delta_c\rvert$
SPECGEN	53.15	INCLING	✓	$2.7 \cdot 10^{-5}$	0.33
INCLING	50.38	ACTS	✓	$0.4 \cdot 10^{-3}$	0.22
		BDDGEN	X	0.4	0.07
BDDGEN	49.96	ACTS	✓	$0.4 \cdot 10^{-1}$	0.16
ACTS	48.21	GFE	X	0.2	0.11
GFE	47.00				

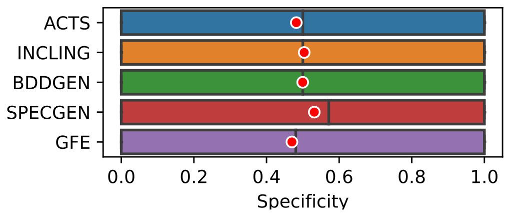


Fig. 8: Relative specificity analysis

Fig. 8 reports the relative specificity, while Tab. V reports the outcome of the Wilcoxon Signed-Rank test. The obtained result shows that even if a technique is not designed for producing specific test suites, such as for the case of ACTS, INCLING, and BDDGEN, some specific test case is always generated. Among the three techniques subject to this RQ, ACTS is the one producing test suites with the lowest specificity. This is demonstrated by the results shown in Tab. V, where ACTS is ranked lower than BDDGEN, which is ranked equivalent to INCLING. The test involving ACTS and BDDGEN is statistically relevant (i.e., with a p-value lower than α) and with an effect size reporting a medium impact. Instead, INCLING and BDDGEN perform comparably.

In conclusion, we can state that the three analyzed general-purpose tools produce on average 48% of specific test cases, and ACTS performs slightly worse than INCLING and BDDGEN in terms of specificity.

RQ3 What are the performances of the SPECGEN test generation strategy aimed to generate specific tests?

After having analyzed the general-purpose methods, we are now interested in evaluating the performance of SPECGEN, which aims at generating test suites maximizing the specificity, in terms of test suite size, generation time, and specificity.

The results of our experiments are reported in Fig. 6, for what concerns the generation time, Fig. 7, in terms of test suite size, and Fig. 8, for what regards the relative specificity. Moreover, we report the results of the Wilcoxon Signed-Rank tests for all three measures in Tab. III, Tab. IV, and Tab. V.

We can observe that SPECGEN, among the tools considered up to now (i.e., excluding the GFE generator, which will be analyzed in RQ4), is the tool producing the highest number of specific tests, thus leading to the highest relative specificity (see Fig. 8). This is confirmed by the Wilcoxon Signed-Rank test results in Tab. V, which show to be statistically relevant for what regards SPECGEN, with a medium-large effect size. In terms of generation time, SPECGEN shows to be faster than ACTS, but slower than INCLING. Finally, for what concerns the test suite size, the comparison between general-purpose tools and SPECGEN shows that all tools perform in a comparable way, except for BDDGEN which produces slightly bigger test suites than SPECGEN.

In conclusion, we can state that SPECGEN produces test suites with a similar test suite size, but with a higher specificity than general-purpose tools.

RQ4 What are the advantages or disadvantages of trying to reuse tests generated for FM to build the new test suite TS' for FM' (especially in terms of specificity)?

In this last research question, we are interested in evaluating the GFE approach proposed by [6], which tries to reuse part of the test suite for FM to generate a test suite for FM'. Our experiments show that GFE is the technique requiring the lowest amount of time for generating test suites (because some of the test cases are given as seeds and this is known to make the test generation faster [9]), producing a comparable number of test cases. However, as shown in Tab. V and Fig. 8, GFE is the approach leading to the lowest specificity.

In conclusion, these experiments confirm that reusing old tests and trying to repair and complement them, can reduce their ability to test the changes introduced during the evolution. Thus, from the tester point of view, there is a trade off between speeding up the test generation process and specifically testing the modifications introduced by the FM evolution.

VI. Threats to Validity

In this section, we discuss the threats to validity [12] and all the strategies we have undertaken to mitigate them.

Internal validity refers to the fact that the different outcomes obtained with the analyzed techniques and tools are actually caused by the different approaches themselves and by the way the experiments were carried out, and not by methodological errors. To mitigate this risk, we have carefully checked the code of the available generators and of the experiments to see if there could be other factors that have caused the outcome, such as errors in the tools. A possible threat to the *construct validity* comes from the assumption that our definition of test suites' specificity is suitable to measure the "quality" of a test suite for testing evolving FMs. To mitigate this risk, we have carefully checked the literature in order to find similar approaches, such as the one proposed by [3] where distinguishing configurations have been proposed, although some of them may no longer be valid in an evolved FM.

External validity is concerned with whether we can generalize the results outside the scope of the presented study. Under this lens, one threat to external validity refers to the case studies we have used in the experiments. We have tried to collect as many examples as possible, and we believe that they are representative enough of the possible evolutions of FMs (different numbers of products and features), even if more complex models may exist in a real scenario and there may be scalability issues for which further experiments are needed. In Sect. V, we have shown the effectiveness of employing the BDD-based method for producing test suites for evolving FMs. This approach proves to be a feasible technique, enabling the generation of a reduced number of test cases, and in certain instances, more specific ones, within a shorter generation time-frame compared to conventional combinatorial methods like ACTS. However, BDDs are known in the literature for failing to scale for large FMs [5]. Thus, further experiments are needed to generalize the results we present in this paper. Moreover, one may argue whether using specific tests actually leads to enhanced bug discovery. This is an additional threat to the external validity of our experiments. We plan to further investigate this aspect by automatically computing mutations on the evolved part of the FMs and checking whether specific tests are more fault-revealing than non-specific ones.

Another possible threat to external validity regards the testing criteria we used, since the approach we present in this paper may be not suitable for other criteria besides the pairwise combinatorial one. We believe that our algorithms can be extended to any testing criteria that can be represented by formal testing requirements that can be translated to BDDs (such as the mutation-based one presented in [3]).

VII. Related Work

The evolution of FMs is a widely studied topic, which poses several challenges for software engineering, especially for what concerns testing and requirements traceability. Particular focus has been put by researchers on finding patterns in SPL evolution. For example, in [25], [27], the authors compared the FMs produced during software evolution, found out that some types of evolution actions are more common than others, and highlighted that arbitrary edits are commonly performed during the SPL life cycle.

Researchers normally address the problem of testing evolving FMs in two different directions, namely, by trying to preserve a test suite as much as possible during the evolution process, or by focusing on testing new products that were not previously valid. For example, in [6], the GFE approach trying to preserve test cases from the original FM is presented. It is based on seeding test cases to a combinatorial test generator and repairing them when they are not valid anymore for the evolved version FM'. With our experiments, as discussed in Sect. V, we have shown how this method is complementary to the one we propose with this paper, as the specificity of the test suites obtained with GFE is lower than those obtained with SPECGEN. On the other hand, in [3], the authors proposed a fault-based approach for testing FMs, based on the identification of distinguishing sequences, i.e., configurations that are able to detect a given fault. This approach is similar to that we propose in this paper, as mutations simulating faults can also be used for simulating model evolution. However, the approach we propose with SPECGEN is a generalization of the one proposed by [3], and our definition and algorithms are applied to the context of FM evolution instead of fault detection.

Similar approaches are proposed for testing regular software undergoing evolution. For example, in [21], the authors propose a method for assessing the adequacy of a test suite after a program is modified and identifying new or modified behaviors that are not adequately exercised by the existing tests. This is what we do with the approach we implemented in SPECGEN, which generates test cases trying to target new product configurations that were not tested before.

VIII. Conclusion

Software Product Line evolution poses a great challenge on testing feature models since every time they evolve a new test suite is needed and no guidance is normally given on what are the characteristics the test suite needs to have. For this reason, in recent years, researchers have spent a great effort in defining strategies for testing in the most effective way possible evolving FMs.

In this paper, we have presented the novel definition of specific tests, i.e., those tests representing products that are valid in the evolved version of a FM but not in the original counterpart. They allow practitioners to better focus the effort of the testing process on the new products introduced with the evolution process. Secondly, we have introduced a measure for the specificity of a test suite, offering a quantitative assessment of how well the test suite tests the FM evolution. This metric provides valuable insights for practitioners to select tools producing test suites that are most apt for testing evolving SPLs. Moreover, in this paper, we have presented a BDD-based test generation strategy, BDDGEN, extended by the SPECGEN generator, designed to produce pairwise combinatorial test suites optimized for maximizing specificity. Through extensive empirical evaluations against state-of-the-art generators, including ACTS, BDDGEN, INCLING, and GFE, SPECGEN has demonstrated its capability to yield test suites of comparable size, but notably, in significantly less time than most other tools and with the highest specificity.

Despite the performed evaluation, as future work, more experiments are needed on additional and more complex models, in order to generalize our conclusion. We are planning to involve professionals and test experts in the loop for better evaluating the applicability of the approach, its potentials, and limitations. Moreover, we are interested in evaluating the impact of removing some of the assumptions we made in the definition of specificity which translates into the property of asymmetry and absence of specific tests. In this way, we would also consider specific tests that were valid in the FM but not in FM'.

References

[1] M. Al-Hajjaji, S. Krieter, T. Thüm, M. Lochau, and G. Saake. IncLing: efficient product-line testing using incremental pairwise sampling. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. ACM, Oct. 2016.

[2] N. Ali and J.-E. Hoing. Your opinions let us know: Mining social network sites to evolve software product lines. *KSII Transactions on Internet and Information Systems*, 13:21, 08 2019.

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

[4] L. Arcega, J. Font, Ø. Haugen, and C. Cetina. Achieving knowledge evolution in dynamic software product lines. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 505–516, 2016.

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

[6] A. Bombarda, S. Bonfanti, and A. Gargantini. On the reuse of existing configurations for testing evolving feature models. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume B*. ACM, Aug. 2023.

[7] A. Bombarda, E. Crippa, and A. Gargantini. An environment for benchmarking combinatorial test suite generators. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 48–56, 2021.

[8] A. Bombarda and A. Gargantini. An automata-based generation method for combinatorial sequence testing of finite state machines. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Oct. 2020.

[9] A. Bombarda and A. Gargantini. Incremental generation of combinatorial test suites starting from existing seed tests. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Apr. 2023.

[10] G. Botterweck, A. Pleuß, D. Dhungana, A. Polzer, and S. Kowalewski. Evofm: feature-driven planning of product-line evolution. In *PLEASE '10*. ACM Press, 2010.

[11] J. Bürdek, T. Kehrer, M. Lochau, D. Reuling, U. Kelter, and A. Schürr. Reasoning about product-line evolution using complex feature model differences. *Automated Software Engineering*, 23(4):67, oct 2016.

[12] R. Feldt and A. Magazinius. Validity threats in empirical software engineering research - an initial survey. In *SEKE*, 2010.

[13] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, and F. Dantas. Evolving software product lines with aspects: An empirical study on design stability. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 261–270, New York, NY, USA, 2008. Association for Computing Machinery.

[14] N. Gámez and L. Fuentes. Software product line evolution with cardinality-based feature models. In K. Schmid, editor, *Top Productivity through Software Reuse*, pages 102–118, Berlin, Heidelberg, 06 2011. Springer Berlin Heidelberg.

[15] T. Heß, C. Sundermann, and T. Thüm. On the scalability of building binary decision diagrams for current feature models. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A*, SPLC '21. ACM, Sept. 2021.

[16] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[17] M. Nieke. *Consistent Feature-Model Driven Software Product Line Evolution*. PhD thesis, Technische Universität Braunschweig, 2021.

[18] T. Pett, S. Krieter, T. Runge, T. Thüm, M. Lochau, and I. Schaefer. Stability of product-line sampling in continuous integration. In *Proceedings of the 15th International Working Conference on Variability Modelling of Software-Intensive Systems*, VaMoS '21, New York, NY, USA, feb 2021. Association for Computing Machinery.

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

[20] pure-systems GmbH. *pure::variants User's Guide*, 2022.

[21] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 218–227, 2008.

[22] A. R. Santos, R. P. de Oliveira, and E. S. de Almeida. Strategies for consistency checking on software product lines: A mapping study. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, EASE '15, New York, NY, USA, 2015. Association for Computing Machinery.

[23] C. Seidl, F. Heidenreich, and U. Aßmann. Co-evolution of models and feature mapping in software product lines. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 76–85, New York, NY, USA, 2012. Association for Computing Machinery.

[24] S.P.L.O.T. Repository of real feature models. [Online; accessed 08-September-2022].

[25] M. Svahnberg and J. Bosch. Evolution in software product lines: two cases. *Journal of Software Maintenance: Research and Practice*, 11(6):391–422, Nov. 1999.

[26] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 47(1):1–45, June 2014.

[27] T. Thüm, D. Batory, and C. Kästner. Reasoning about edits to feature models. In *2009 IEEE 31st International Conference on Software Engineering*, page 11. IEEE, 05 2009.

[28] M. Varshosaz, M. Al-Hajjaji, T. Thüm, T. Runge, M. R. Mousavi, and I. Schaefer. A classification of product sampling for software product lines. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*. ACM, sep 2018.

[29] J. White, J. A. Galindo, T. Saxena, B. Dougherty, D. Benavides, and D. C. Schmidt. Evolving feature model configurations in software product lines. *J. Syst. Softw.*, 87:119–136, jan 2014.

[30] R. F. Woolson. Wilcoxon signed-rank test, Sept. 2008.

[31] L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn. Acts: A combinatorial test generation tool. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 370–375, 2013.

[32] V. Štuikys, R. Burbaitė, K. Bespalova, and G. Ziberkas. Model-driven processes and tools to design robot-based generative learning objects for computer science education. *Science of Computer Programming*, 129:48–71, 2016. Special issue on eLearning Software Architectures.