# On the Reuse of Existing Configurations for Testing Evolving Feature Models

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

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

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

## ABSTRACT

Software Product Lines (SPLs) are used for representing a variety of highly configurable systems or families of systems. They are commonly represented by feature models (FMs). Starting from FMs, configurations, used as test cases, can be generated to identify the products of interest for further activities. As the other types of software, SPLs and their FMs may evolve due to changing requirements or bug-fixing. However, no guidance is usually given on what to do with derived configurations when an FM evolves. The common approach is based on generating all configurations from scratch, which is not optimal since a greater effort is required for concretizing the new tests, and some of the old ones may be still applicable.

In this paper, we present the use of a technique for generating combinatorial tests for evolving feature models: this technique incrementally builds the new combinatorial configuration set starting from the one generated from the previous model. Furthermore, we present a novel definition of *dissimilarity* among configuration sets that can be used to evaluate how much an evolved test suite differs from the previous one and thus allows evaluating the effort required for adapting old test cases to the new ones.

Our experiments confirm that using the proposed technique, in general, leads to lower dissimilarity and test suite size w.r.t. the generation of tests from scratch.

## CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; **Feature interaction**.

**ACM Reference Format:**
Andrea Bombarda, Silvia Bonfanti, and Angelo Gargantini. 2023. On the Reuse of Existing Configurations for Testing Evolving Feature Models. In *27th ACM International Systems and Software Product Line Conference - Volume B (SPLC '23), August 28-September 1, 2023, Tokyo, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3579028.3609017

## 1 INTRODUCTION

Software Product Lines (SPLs) are increasingly used in practice for representing systems having some common aspects and variability parameters. In an SPL, the different members of a family of similar software products are explicitly distinguished by means of their features. A feature is defined as a user-visible aspect or characteristic of a software system [1]. Features in a product line have different relations that can be described compactly by means of Feature Models (FMs). Product samples, to be used as tests, can be derived from FMs by using several criteria [2, 3] and they are used for testing actual products of the SPL [4]. In the following, we will use simply the terms *test* to refer to a *configuration sample* and *test suite* to indicate a set of possible *product configurations* generated from a given FM. In this work, we focus on combinatorial testing criteria which is one of the most used criteria [2].

Establishing an SPL is generally considered a long-term investment since it can be complex to represent all the products of interest. During their lifetime, FMs can evolve, with the introduction or removal of features or the change of constraints among them, in order to meet ever-changing requirements [5]. Existing research mostly focuses on techniques and case studies regarding the evolution of variability models alone [5–8], while little attention has been given to the coevolution of the test suites, which are an important asset of the SPL. The evolution of tests is normally tackled for code [9], where the information available in existing test cases is used and, through a set of heuristics, they are repaired and adapted to the evolved software. However, the evolution can involve models of the systems, from which tests have been derived. It is recognized that an open problem in model-based testing is what to do with the derived test cases when a system model evolves because some faults are found or new requirements are set by the users.

Testing techniques for FMs do not consider their history and the evolution in SPLs [2]. When a model is modified, a new configuration set is generated from the new model, and the testing activity restarts. We call this basic technique Generation From Scratch (GFS). This approach has two main drawbacks:

(1) the generation itself may be time-consuming, since every time a new complete test suite is generated;
(2) existing tests are discarded together with all what is attached to them.

While the former problem may require only some additional computation time (although the test generation itself may require some human intervention as well), the latter can increase the cost of testing activities significantly. Indeed, there are generally other artifacts linked to existing configuration sets. For instance, test suites are commonly translated into concrete tests and then used to build actual products and/or perform real experiments and these activities can come with higher costs. All the activities performed so far on the existing feature model risk being lost, if the test suite is completely discarded and a new one is generated. For this reason, we first contribute by devising a technique, called Generation From Existing tests (GFE), supported by a tool, that is able to reuse an existing test suite, by repairing existing tests or their parts, and to extend it only by those tests that are really required by the new feature model.

GFS and GFE can be compared in terms of the number of tests and time required to generate the tests, which give a first approximate measure of the effort required to update the testing activities related to the evolution of the model. However, we are interested in introducing a measure of how much the new test suite differs from the old one. The idea behind this endeavor is that if the final test suite built for the evolved model obtained by evolution is very dissimilar to the original one, then a great effort will be required to perform test concretization. On the contrary, if the evolved test suite is very similar (even equal) to the original one, then the effort to reuse the existing tests will be minimal. Hence, the second contribution of this paper is the formal definition of the *dissimilarity* among test suites and the introduction of a simple greedy algorithm which computes it.

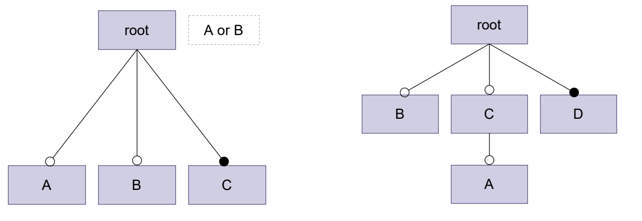
The experiments we have carried out, confirm that using GFE instead of GFS, in general, leads to lower test suites size and dissimilarity, but for big models can require some extra time.

The paper is structured as follows. Section 2 presents the background on FMs and on the GFS technique used to generate tests from them. Section 3 introduces the GFE technique and the tool which we have extended in order to support it. Section 4 reports the measures used to compare the test generation strategies and the novel concept of dissimilarity. Then, in Section 5 we report the experiments carried out in order to evaluate the proposed techniques, and in Section 6 we discuss possible threats to validity. Finally, Section 7 presents related work on FMs and tests evolution, while Section 8 concludes the paper.

## 2 BACKGROUND

*SPL and Feature models.* In software product line engineering, feature models [10, 11] are a special type of information model representing all possible products of an SPL in terms of features and relations among them. Specifically, a feature model *FM* is a hierarchically arranged set of features *F*, where each parent-child relation between them is a constraint of one of the following types: • *Or*: at least one of the sub-features must be selected if the parent is selected. • *Alternative* (xor): exactly one of the children must be selected whenever the parent feature is selected. • *And*: if the relation between a feature and its children is neither an *Or* nor an *Alternative*, it is called *and*. Each child of an *and* must be either: – *Mandatory*: the child feature is selected whenever its respective parent feature is selected. – *Optional*: the child feature may or may not be selected if its parent feature is selected.

Only one feature in *F*, which is the *root* of *FM*, has no parent and it is selected in every product. In addition to parental relations, it is possible to add *cross-tree constraints*, i.e., relations that cross-cut hierarchy dependencies. *Simple* cross-tree constraints are:



**(a) Initial feature model** *FM* **(b) Evolved feature model** *FM′*

**Figure 1: Example of a feature model evolution**

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

In this work, we allow feature models to contain complex cross-tree constraints (i.e., given by *general* propositional formulas).

A feature model can contain *dead* features, i.e., those that cannot be selected, and *core* features, i.e., those mandatory in every valid product, like the *root*.

An example of feature model is shown in Figure 1a. It has 4 features: one is the root (root), one is mandatory (C) while the other two (A and B) are optional. Moreover, it contains the constraint A or B which requires at least one of the specified features to be selected in every valid product. In that example, root and C are core features.

### 2.1 Feature model evolution

SPLs and their feature models evolve over time [5]. Features can be added, removed, moved, and constraints modified. Each change, however small, is likely to change the set of legal feature combinations: certain configurations can be no longer valid, and others could become valid instead.

Consider the feature model shown in Figure 1a, which is modified as shown in Figure 1b by moving the feature A as a sub-feature of C, removing the mandatory constraint on the C feature, by removing the cross-tree constraint, and by adding the mandatory D feature. While in the original *FM* all the configurations without C are not admissible, in the new model *FM′*, such configurations may represent valid products. Moreover, D is not present in any of the valid configurations for the original *FM*, while it must be added in all valid products of *FM′*.

### 2.2 Test suites for Feature Models

When working with a feature model *FM*, each (abstract) test specifies which features of *FM* are selected and which are not. Since the feature set can evolve, we want to distinguish when a test does not select a feature by choice (so the product identified by that test will have that feature unselected) or because simply that feature does not exist. We will use a test *t* as a function that returns the status of a feature *f* in *t*

$$t(f) = \begin{cases} \top & \text{if } f \text{ is selected in t} \\ \bot & \text{if } f \text{ is not selected in t} \\ \_ & \text{if } f \text{ does not exist in the feature model} \end{cases}$$

**Table 1: Example of a test suite for *FM* and one for *FM′*. The meaning of $t_{\nexists}$ and *testDist* will be presented in Section 4**

|  | **A** | **B** | **C** | **D** |  | **A** | **B** | **C** | **D** | $testDist(t_i, t'_i)$ |
|---|---|---|---|---|---|---|---|---|---|---|
| *TS* for *FM* |  |  |  |  | *TS′* for *FM′* |  |  |  |  |  |
| $t_1$ | ⊥ | ⊤ | ⊤ | _ | $t'_1$ | ⊥ | ⊥ | ⊤ | ⊤ | 2 |
| $t_2$ | ⊤ | ⊥ | ⊤ | _ | $t'_2$ | ⊤ | ⊥ | ⊤ | ⊤ | 1 |
| $t_3$ | ⊤ | ⊤ | ⊤ | _ | $t'_3$ | ⊤ | ⊤ | ⊤ | ⊤ | 1 |
| $t_{\nexists}$ | _ | _ | _ | _ | $t'_4$ | ⊥ | ⊤ | ⊥ | ⊤ | 4 |
| $t_{\nexists}$ | _ | _ | _ | _ | $t'_5$ | ⊥ | ⊥ | ⊥ | ⊤ | 4 |

For example, Table 1 reports two test suites for the models shown in Figure 1. In particular, all the $t_i$ represent tests of the test suite generated starting from the original model (see Figure 1a), while all the $t'_i$ represent the tests appertaining to the test suite generated starting from its evolution (Figure 1b). Note that, for easier interpretation, Table 1 reports the same features and the same number of tests for both feature models, even if the former has only 3 real tests and does not have the feature D. In this case, we consider the test cases and features added as being unavailable and we mark them as _. Note that the *root* feature is omitted in tests since it must be selected by each valid product.

Since the number of tests for a feature model is normally too high for performing exhaustive testing, the literature reports several techniques for creating test suites for feature models such as product sampling [2], search-based algorithms [12, 13] and combinatorial testing, which allows testers to select only valid products.

## 2.3 Generation From Scratch (GFS)

Once the considered feature model evolves, the common current solution, for obtaining a new test suite, is based on test generation from scratch (GFS) [14]: the evolved feature model is given as input to a test generator and a completely new test suite is generated.

This technique is trivial but has the following drawbacks:

- regenerating the whole test suite does not take into account some of the old tests that may be still valid for the evolved feature model;
- regenerating the whole test suite may require considerable time, especially when feature models are complex;
- the old test suite may contain some specific tests that were written in order to test critical configurations of the system; if the test suite is regenerated, these tests may be lost, while it would be better to keep them and, possibly modify them if they became invalid;
- old tests should be kept as long as possible in order to be used as a solid base for regression testing.

In order to address these limitations, in the following, we propose a method that allows reusing, completely or partially, the tests from the old test suite and evolving them in order to be adapted to the evolved feature model.

## 3 REUSE OF EXISTING TEST SUITES

As discussed in the previous sections, reusing existing test suites can provide several advantages. In this section, we introduce a simple way to adapt old test suites, by adopting an incremental technique that we call Generation From Existing tests (GFE). There are several approaches for the generation of combinatorial test suites starting from existing tests, called *seed tests*. For instance, already in [15], the authors highlight the importance that testers could guarantee the inclusion of their favorite test cases by specifying them as seed tests. Generating test sets from seeds is also supported by several tools such as ACTS [16], jenny [17], PICT [18], and pMEDICI [19]. Some of them allow the use of *partial* tests, i.e. tests that do not assign value to each parameter. However, all of them assume that seed tests are *valid* tests, since they are not intended to be applied to evolving models or for reusing test suites generated for different models. In our case, though, some tests may become *partial* due to the addition of new features as well as *invalid* due to the change in the structure of the feature model or of its constraints. For this reason, the use of existing tools as they are now implemented is not feasible. We have decided to extend pMEDICI, which is open source[1], produces one test at a time, and is able to deal with constraints among parameters of Boolean and enumerated types, in the following way.

**pMEDICI test generation strategy:** During test generation, pMEDICI builds one test at a time trying to cover many tuples in each test. It stores each test case in a structure called *test context* that contains all the assignments committed so far for that single test under construction and all the constraints by means of a Multi-Valued Decision Diagram (MDD) which is an extension of the classical Binary Decision Diagram (BDD). The pMEDICI regular process have been modified, for this work, as follows.

**Test suite pre-processing and repairing:** At the beginning of the generation for the feature model *FM′*, we add all the tests given in an existing test suite *TS* (likely for a different previous feature model *FM*): for each test $t$ in *TS* we build a test context containing only all the valid assignments of $t$. To do this, we consider one feature $x$ at a time. If $x$ is no longer present in *FM′*, we skip it. If the current assignment to $x$ in $t$ is still valid (easily checked by using the MDD), we add it to the current context, otherwise, we skip it (only that assignment). At the end, we will have a set of test contexts partially filled with the old tests of *TS* and then the actual generation of pMEDICI can start.

**Test suite completion:** pMEDICI considers all the tuples for the new model *FM′* that need to be covered and for each tuple, if it is not already covered, it is added to an existing test context (if possible) or a new test context (and hence a test) is built. At the end, pMEDICI produces a test suite *TS′* that is valid and reaches the combinatorial coverage of *FM′*, but it reuses the old test suite *TS* as much as possible.

## 4 HOW TO COMPARE TEST GENERATION STRATEGIES

When it comes to choosing a test generation strategy or tool, it may be difficult to find the most suitable one for the specific situation. In the following, we present three measures we use in this paper for comparing test generation strategies: generation time, test suite size, and dissimilarity.

### 4.1 Generation time and test suite size

We consider two of the most common measures, namely the *time* required for test suite generation and the *number of tests* in the test

[1]https://github.com/fmselab/ct-tools/tree/main/pMEDICI

suite [20]. Reducing the former is normally preferred when tests can be executed in a short amount of time (i.e., the generation time impacts more than the test execution time) while reducing the latter is preferred for systems where tests require a considerable effort for being executed (i.e., testers prefer to have fewer test cases). For instance, if tests need to be translated, as often done in model-based testing for concretizing abstract tests [21], a lower number of test cases is normally preferred.

## 4.2 Dissimilarity between test suites

When feature models evolve, tests representing valid products must evolve, too. Here, we want to evaluate the "effort" required to testers when concretizing the evolved test suite by modifying the previous one. For this reason, we propose a definition of *dissimilarity* that allows for evaluating the difference between two test suites, generated starting from a feature model *FM* and its evolution *FM'*. This definition is inspired by [13, 22] which examined various types of distance measurement in the context of SPL testing and reported that Hamming distance (i.e., the number of points at which two corresponding tests are different [23]) is generally more effective than other distance measures. Because we want to somehow count how many feature values change when tests evolve, first we want to identify the features of interest that *could* change:

Definition 1 (Changeable Features). *Let $D_{FM}$ and $C_{FM}$ be the set of dead and core features of a feature model FM, FM and FM' be two feature models, F and F' be their feature sets. The set of* changeable features *is defined as:*

$$chngF(FM, FM') = (F \cup F') \setminus ((D_{FM} \cap D_{FM'}) \cup (C_{FM} \cap C_{FM'}))$$

We consider changeable all the features, except those that are dead or core in both models. Now, we introduce the definition of distance between two generic test cases.

Definition 2 (Distance between test cases). *Let FM and FM' be two feature models and* ***chngF****(FM, FM') its changeable features set. The distance between two tests t and t' derived, respectively, from FM and FM', can be computed as follows:*

$$testDist(t, t') = |\{f \mid f \in chngF(FM, FM') \wedge t(f) \neq t'(f)\}|$$

This distance counts the number of features in which two products are different. Each feature $f$ that could change is counted in the distance when it is either selected or not by only one of the two test cases, or when it is present only in *FM* or *FM'*. Note that the *testDist* is symmetric, i.e. $testDist(t_1, t_2) = testDist(t_2, t_1)$.

The maximum distance between two tests is obtained when all the features that could change are actually different in the two test cases. Thus, the maximum possible *testDist* is equal to $|chngF(FM, FM')|$.

Our distance always counts newly added (or removed) features, since they belong to *chngF* and cannot keep the same value in the test. For this reason, any test $t'$ of a feature model that introduces $n$ new features will have a minimum distance of $n$ to any test of the original model. This is desired because $t'$ is different for those new features to any test for the old feature model, regardless if new features are selected or not.

**Example 1.** Let's consider the two feature models in Figure 1, $t_1$ a test case for the feature model *FM* in Figure 1a, and $t'_1$ a test case for the feature model *FM'* in Figure 1b, as reported in Table 1. The distance between these two test cases can be computed as follows. First, we compute the set $chngF = \{A, B, C, D\}$.
Then, we fetch each feature $f \in chngF$ and count for how many $t_1(f) \neq t'_1(f)$:

$$testDist(t_1, t'_1) = |\{f \mid f \in chngF \wedge t_1(f) \neq t'_1(f)\}| = |\{B, D\}| = 2$$

## Distance among test suites

Having defined the distance between two test cases, we can now introduce the concept of distance and dissimilarity between test suites. Intuitively, to measure the distance between two test suites *TS* and *TS'*, we sum the distances between the tests in *TS* and *TS'* taken one by one.

Definition 3 (Distance between test suites). *Given two arbitrarily ordered test suites $TS = \{t_1...t_n\}$ and $TS' = \{t'_1...t'_n\}$, respectively derived from the feature models FM and FM', and having equal size n, the distance between TS and TS' is defined as*

$$testSuiteDist(TS, TS') = \sum_{i=1}^{n} testDist(t_i, t'_i)$$

Definition 3 assumes that both test suites have equal size. In case one test suite is smaller than the other, we complement it with as many inexistent test cases $t_{\nexists}$ as needed. In a $t_{\nexists}$, any feature $f$ belonging to either *FM* or *FM'*, is not existing:

$$\forall f \in (F \cup F'),\ t_{\nexists}(f) = \_$$

Note that our definition of distance among test suites is able to distinguish the case in which a feature $f$ is not selected by a test case from that in which $f$ does not exist, and the case in which a test case is composed only by not selected features from that in which the test case is empty. This is the reason why we do not use more classical definitions of distances based on sets, such as Jaccard distance. We will motivate this choice in Section 6.

Since the distance between test suites is computed by summing the distances between test cases taken one by one, it depends on the order in which the test cases are considered. For this reason, we define the *dissimilarity* between test suites as follows:

Definition 4 (Dissimilarity between test suites). *Let TS and TS' be two test suites derived respectively from two feature models FM and FM'. The dissimilarity between the two test suites is defined as the minimum distance and it is computed as follows:*

$$dissimilarity(TS, TS') = \min testSuiteDist(TS, TS')$$

In order to scale all dissimilarities in the same range, one may want to compare test suites using the *percentage dissimilarity*, which computes the ratio between the actual dissimilarity and the ones of the worst scenario. Considering two feature models *FM* and *FM'*, *F* and *F'* the set of all the features respectively of *FM* and *FM'* (excluding not existing features), *TS* and *TS'* two test suites derived respectively from two feature models *FM* and *FM'*, and *chngF(FM, FM')* the set of changeable features, the worst case dissimilarity is computed as follows:

$$dissimilarity_{worst}(TS, TS') = \max(|TS|, |TS'|) \times |chngF(FM, FM')|$$

Indeed, considering that when a test suite has fewer tests than the other we complement it with the inexistent test case $t_{\nexists}$, both test suites will result in having $\max(|TS|, |TS'|)$ tests, each one with a maximum distance of $|chngF(FM, FM')|$, as previously discussed.

The *percentage dissimilarity* is obtained as follows:

$$dissimilarity_{\%}(TS, TS') = \frac{dissimilarity(TS, TS')}{dissimilarity_{worst}(TS, TS')} \cdot 100\%$$

In the following example, we show the calculation of the percentage dissimilarity for the feature models shown in Fig 1.

**Example 2.** Given the two test suites in Table 1, the distances between the tests are those reported in the $testDist(t_i, t_i')$ column. As reported in Definition 3, in order to compute the distance between $TS$ and $TS'$, the two test suites must have the same number of test cases. For this reason, we append to $TS$ two inexistent test cases $t_{\nexists}$. In this way, we can compute the distance between the two test sets as follows:

$$testSuiteDist(TS, TS') = \sum_{i=1}^{5} testDist(t_i, t_i') = 2 + 1 + 1 + 4 + 4 = 12$$

Considering that, based on our experiments, the proposed order in which to consider the tests is the one leading to the lowest distance, the computed distance is the *dissimilarity* we are looking for. Furthermore, the percentage dissimilarity between the two test suites is

$$dissimilarity_{\%}(TS, TS') = \frac{12}{5 \times 5} \cdot 100\% = 48\%$$

## 4.3 A greedy approach to compute the dissimilarity function

Since the dissimilarity between two test suites depends on the order in which tests are considered, its computation can be represented as an optimization problem: choose the best tests ordering that minimizes the distance. We were able to represent this problem using a matrix (containing all the distances among tests in the two test suites) as Mixed Integer Programming with all the constraints to force the function to be (partially) one-to-one. We were able to solve this problem using GAMS[2] (General Algebraic Modeling System), which is a high-level modeling system for mathematical optimization, and it is designed for modeling and solving linear, nonlinear, and mixed-integer optimization problems. However, we found that even to solve a simple problem, our GAMS program required minutes of elaboration. Moreover, each feature model evolution may require a different GAMS program, and the automation of the generation of the correct GAMS program is a challenging problem. For this reason, we decided to implement a greedy algorithm, inspired by the Hungarian algorithm proposed in [22], to compute the minimal distance between two test suites.

Alg. 1 computes an alleged minimal distance between $TS$ and $TS'$. It compares each test in $TS$ to one of $TS'$, and finds the matching between test cases having the lowest distance. Initially, the distance is set to the worst case, i.e., to the total number of changeable features *nChgFeats* (line 2). Then, each pair $\langle t, t' \rangle$, where $t \in TS$ and $t' \in TS'$, is checked and the distance between the two tests is evaluated (line 4). When the minimum distance between $t$ and $t'$ is found, the value of *dissimilarity* is updated and the two tests are removed from the two test suites (lines 9-10). This process is repeated until one of the two test suites is completely empty. Then, if a test suite has some remaining test, all of them contributes to the dissimilarity with the highest distance between test cases possible (lines 12-15).

**Algorithm 1** Greedy algorithm for computing the dissimilarity

```
Require: TS the old test suite
Require: TS' the new test suite
Require: nChgFeats the total number of changeable features, i.e.,
         |chngF(FM, FM')|
Ensure: dissimilarity the dissimilarity between TS and TS'
 1: while $TS \neq \emptyset$ and $TS' \neq \emptyset$ do
 2:     $min \leftarrow nChgFeats$
 3:     for all $t \in TS$ and $t' \in TS'$ do                ▷ Find the closest pair
 4:         if $testDist(t, t') \leq min$ then
 5:             $min \leftarrow testDist(t, t')$
 6:             $t_{min} \leftarrow t$; $t'_{min} \leftarrow t'$
 7:         end if
 8:     end for
 9:     $dissimilarity \leftarrow dissimilarity + min$
10:     $TS \leftarrow TS - t_{min}$; $TS' \leftarrow TS' - t'_{min}$    ▷ Remove $t_{min}$ and $t'_{min}$
11: end while
    ▷ Compute the dissimilarity due to remaining tests
12: if $TS \neq \emptyset$ then
13:     $dissimilarity \leftarrow dissimilarity + (nChgFeats \cdot |TS|)$
14: else if $TS' \neq \emptyset$ then
15:     $dissimilarity \leftarrow dissimilarity + (nChgFeats \cdot |TS'|)$
16: end if
```

**Table 2: List of the FM evolution examples from the literature**

| Example | V | #F | #P | Ref. | Example | V | #F | #P | Ref. |
|---|---|---|---|---|---|---|---|---|---|
| AmbAssistLiving | 2 | 24-32 | $9.8\cdot10^4$-$5.0\cdot10^7$ | [25] | SmartHotel | 2 | 6-8 | 6-30 | [26] |
| AutomotiveMult. | 3 | 6-13 | 5-192 | [27] | Smartwatch | 2 | 12-15 | 96-192 | [28] |
| Boeing | 3 | 5-6 | 2-2 | [29] | WeatherStat. | 2 | 22-23 | 528-660 | [30] |
| CarBody | 4 | 6-13 | 4-40 | [31] | MobileMedia | 6 | 11-26 | 2-272 | [32] |
| Linux (Simple) | 3 | 5-10 | 7-33 | [24] | HelpSystem | 2 | 25-26 | $672$-$10^3$ | [33] |
| ParkingAssistant | 5 | 6-16 | 1-32 | [8] | SmartHome | 2 | 38-61 | $9.0\cdot10^5$-$3.9\cdot10^9$ | [34] |
| Pick&PlaceUnit | 9 | 5-11 | 3-81 | [35] | ERP | 2 | 42-57 | $2.6\cdot10^4$-$2.6\cdot10^5$ | [36] |
| BCS | 3 | 13-17 | 128-768 | [22] | | | | | |

**Example 3.** Let's consider the feature model of the Linux Kernel presented in [24], which has two evolutions. We have generated the test suite $TS$ from the LinuxKernelv2 model and $TS'$ from the LinuxKernelv3, and then we have repeatedly calculated the dissimilarity between $TS$ and $TS'$ by using the proposed algorithm. Every time, before applying the greedy computation of the dissimilarity by Alg. 1, we shuffled both test suites. We have found that depending on the order of test cases in the test suites, the computed percentage dissimilarity may vary from 16.19% to 18.10%. This proves that the greedy algorithm could return dissimilarity that is close to the minimum but it is not the minimum.

# 5 EXPERIMENTS

In this section, we present the experiments and the results obtained by comparing the GFS and GFE test generation techniques for evolving feature models. In particular, we are interested in answering the following research questions:

*Which is the test generation technique leading to ...*

RQ1 ... the lowest generation time?

RQ2 ... the lowest test suite size?

[2]https://www.gams.com

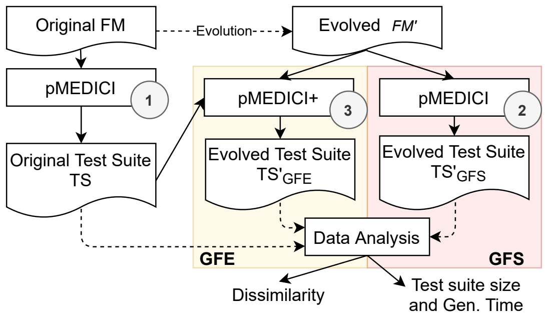


**Figure 2: Experimental methodology - pMEDICI+ is the extended version of pMEDICI used for GFE.**

RQ3 ... a lower dissimilarity of the new test suite w.r.t. the original test suite?

## Experimental methodology

In order to answer the research questions listed above, we have gathered a set of 15 feature models and their evolutions, for a total of 50 models coming from real case studies. Table 2 reports the list of the FMs we have used in our experiments, the number of versions (V), the minimum and maximum number of features (including the core and dead ones) across all the evolutions (#F), the minimum and the maximum number of products (#P), and the reference to the article where the models come from. Note that, for the models having more than a single evolution, we consider evolutions only between two consecutive steps (i.e., we compare the version $v1$ with the version $v2$, then $v2$ with $v3$, and so on). In total, we consider 35 evolutions.

The replication package, containing the feature models, the evaluation script, the results of our experiments, and an executable jar for repeating the experiments is available online [37].

For each model and its evolution, we apply the process depicted in Figure 2 and explained in the following, with strength $t = 2$, i.e., applying pairwise test generation. First, we use pMEDICI for computing the test suite $TS$ achieving the intended combinatorial coverage on the *original* feature model (step 1 in Figure 2). Then, we generate the test suite $TS'_{GFS}$ from the *evolved* model by applying GFS (step 2 in Figure 2). Afterward, we proceed with the evaluation of GFE based on the incremental generation of tests starting from partial test seeds (step 3 in Figure 2) using *pMEDICI+*: starting from the *evolved* feature model and the original test suite, we generate the evolved test suite $TS'_{GFE}$ as explained in Section 3. Then we compare both test suites $TS'_{GFE}$ and $TS'_{GFS}$ by analyzing their size and their generation time. Moreover, we can compare them both with the original test suite $TS$ in terms of dissimilarity.

We have repeated the experiments 10 times on a PC using a Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz (16 physical cores, 32 logical cores) with 256 GB RAM. Note that, even if pMEDICI supports multithreading [38], each experiment has been carried out using only 1 thread, but the approach would work with higher number of threads as well. In this way, we isolate the effect of the chosen generation technique from that of the number of threads used.

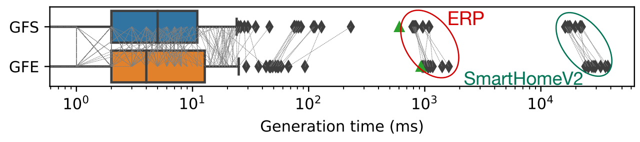


**Figure 3: Test suite $TS'$ generation times with GFS and GFE. The $x$ axis is on a log. scale.**

In order to compare the results obtained by each test generation strategy, we use the Wilcoxon Signed-Rank test [39], a general test that compares the distributions in paired samples and that does not require data to be normally distributed. Given $x$ the measure to be compared between the two techniques, the test is performed using a significance level $\alpha = 0.05$ and the null hypothesis $H_0$ stating that the distributions of $x$ in the two techniques are equal. Besides the Wilcoxon Signed-Rank test, we have evaluated the effect size by computing the Cliff's delta $\delta_c$. The effect of a technique is small if $|\delta_c| < 0.147$, medium if $0.147 \leq |\delta_c| < 0.33$ or large if $|\delta_c| \geq 0.33$.

## 5.1 RQ1: Test suite generation time analysis

As presented in Section 3, one of the most common measures used to evaluate a test generation strategy is the test suite generation time.

For this reason, in this research question, we evaluate GFS and GFE by comparing the time $t$ required for generating a test suite. We have obtained $\bar{t}_{GFS} = 570.75$ ms, and $\bar{t}_{GFE} = 874.77$ ms, but $pvalue = 0.45$ by using a Wilcoxon Signed-Rank test. Therefore, we cannot reject the null hypothesis even if $\bar{t}_{GFS} < \bar{t}_{GFE}$, as also suggested by a $\delta_c = 0.03$.

A box plot with the test generation times is shown in Figure 3[3]. We can observe that in many cases GFE performs better than GFS (see the median value and the box of the plot); for small models, GFE is faster than GFS (but by a negligible amount of time, considering that these generation times are small) while for the biggest models (namely ERP and SmartHome) GFE requires much more time than GFS. It is apparent that for big models, pre-processing is not worthwhile and GFS is the best solution. For these reasons, we would recommend the use of GFS regarding the generation time, but we plan to find a useful and validated measure of the complexity of feature models that could suggest when the use of GFE is preferable instead.

## 5.2 RQ2: Test suite size analysis

One of the most pursued objectives in software testing is the reduction in test suite size. Thus, for comparing the proposed techniques we have gathered the test suite size $s$ obtained in all the performed experiments. The results of our experiments on the feature models taken from the literature are shown in Figure 4.

We have compared the results of the two techniques by using a Wilcoxon Signed-Rank test and we have obtained $pvalue = 1.74 \cdot 10^{-7}$, $\bar{s}_{GFS} = 9.72$ test cases, and $\bar{s}_{GFE} = 9.27$ test cases (with an effect size $\delta_c = 0.20$).

[3]The box-plots include lines linking the corresponding points with the two techniques; the green triangles show the mean values.

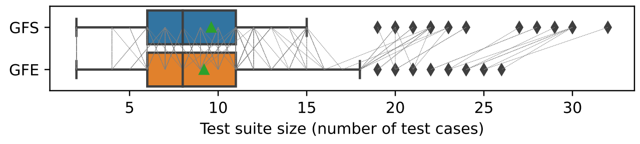


**Figure 4: Test suite $TS'$ sizes with GFS and GFE.**

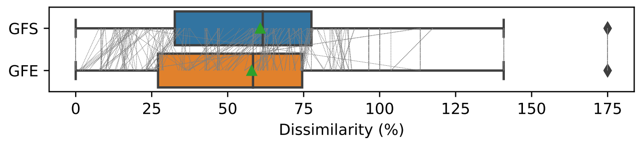


**Figure 5: Dissimilarity between $TS$ and $TS'$ with GFS and GFE.**

Therefore, since $pvalue < \alpha$, we can reject the null hypothesis and claim that the average size of GFE is smaller than that of GFS, as highlighted, in the most of cases in Figure 4. This is somehow unexpected, and mitigates less positive results on the test generation time for more complex models. In those cases in which minimizing the test suite size is more important than saving some time during test generation, GFE is preferable w.r.t. GFS.

### 5.3 RQ3: Dissimilarity analysis

Given the definition of dissimilarity as explained in Section 4.2, we here analyze the *percentage dissimilarity* of the test suites produced by using GFS and GFE.

As for the previous measures, we have compared the dissimilarities $d$ obtained on the feature models in Table 2, with the two techniques, using the Wilcoxon Signed-Rank test and we have obtained $pvalue = 1.81 \cdot 10^{-15}$, $\bar{d}_{GFS} = 59.37$, and $\bar{d}_{GFE} = 55.91$. Therefore, we can reject the null hypothesis and, since $\bar{d}_{GFS} > \bar{d}_{GFE}$, we can conclude that GFE generally outperforms GFS in terms of dissimilarity. This is confirmed by the plot in Figure 5 where we can observe that the dissimilarity of the test suite produced with GFE is lower, in terms of mean value, than that of the test suite produced with GFS. The difference on average is around 3.46 over values of around 59, so it is not negligible (~5%). Moreover, the Cliff's delta $\delta_c = 0.30$ confirms the correlation between the chosen techniques and the difference in the dissimilarity.

The experiments and the analysis we have performed show that GFE may take **more time** for big models, but it produces **smaller** test suites that are above all **more similar** to the original test suites produced for the original model. Smaller size and higher similarity can reduce the cost of adapting existing configurations for evolving feature models.

## 6 THREATS TO VALIDITY

In this section, we discuss the threats to validity [40] and all the strategies we have undertaken to mitigate them.

*Internal validity* refers to the fact that the different outcomes obtained with the two techniques (GFS and GFE) are actually caused by the techniques 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 to see if there could be other factors that have caused the outcome, such as errors in the tools or particular combinations or ordering of tests or test sequences. For instance, the result may depend on the order in which tests are first generated and then extended (see Example 3). To mitigate this risk, we have applied a shuffling procedure in several parts and performed the experiments many times.

A possible threat to the *construct validity* comes from the assumption that our definition of test suites' *dissimilarity* is suitable to measure the distance among test suites (provided that the greedy algorithm is accurate enough). We rely on the literature for this, where similar distances are often used like in [13]. Given the way in which we compute the dissimilarity, adding or removing a test case to the old test suite contributes to increasing the dissimilarity. This is motivated by the fact that adding a new test case requires the effort for its concretization linear with the number of features. Also removing a test case increases the dissimilarity, because it requires the old test suite to be modified. However, note that by applying GFE, a test is never completely discarded unless it contains no valid assignment. Furthermore, we have simulated how a test suite could be translated in JUnit code, and we confirm that our definition is valuable and captures the effort required by a test modification. For instance, in FeatureIDE, every feature of the model must be set in the test case, even if it is unselected, as in the following Java code: `configuration.setManual(feature, Selection.UNSELECTED)`. This justifies that, if a feature is added or removed (so it causes a dissimilarity), the test must be modified regardless of whether it is selected or not. We recognize that there may be cases in which our distance does not capture the real effort required to modify an existing test suite. For instance, we consider the modification of a test to be much less costly than writing a new test, but there are some cases in which a modified test or a new one require the same effort, so any change of one test has the same weight. Our generation techniques could still be used by a designer who wants to define his/her own distance and then decide which test suite is preferable (GFS and GFE) according to his own metric.

*External validity* is concerned with whether we can generalize the results outside the scope of our study. 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 number of products and features). However, the use of MDDs in pMEDICI may make our technique not scalable when considering models with a great number of features.

Another generalization regards the testing criteria we used, since the approach may be not suitable for other criteria besides the pairwise combinatorial one. We believe that our algorithm and tool could be extended to be used with any testing criteria that can be represented by formal testing requirements that can be translated to MDDs. For other testing criteria, however, the incremental building may become more expensive than the generation from scratch. In our experiments, we use pMEDICI, also for GFS, whereas any combinatorial tool could be used to generate the initial test suite (to be amended later using GFE). We have actually tried another tool (ACTS) and obtained exactly the same statistical conclusions (not reported here), with an even longer running time for GFS, but further experiments are needed. Moreover, we recognize that our work is focused only on changes in FMs and not on other artifacts [41].

However, these changes are out of the scope of this paper and we plan to investigate how to apply the methodology proposed when other changes beyond the FM occur. Finally, the last generalization threat regards the number of threads used by pMEDICI for generating test suites. Indeed, in [38], we have found that the test suite size and generation time may vary when a different number of threads is used. Instead, in this paper, we have reported only experiments using only a single thread for generating test cases, both with GFS and GFE, in order to evaluate the impact of the test generation technique without the influence of the multithreading.

## 7 RELATED WORK

Test case evolution is a well-known problem in software engineering, normally studied when the *code* of the system under test is modified for any reason [42]: test cases designed for the early versions of the system may become obsolete and possibly fail during the software lifecycle. This happens even more frequently when software requirements are subject to frequent changes, such as during agile processes. A way to avoid this is to *repair* the tests. In [9], the authors identify a series of common actions the developers perform in order to update the tests and through a series of heuristics devise an automatic method capable of repairing test cases invalidated by changes in the software. This technique can successfully repair up to 90% of the broken test cases [43], although there is no guarantee that it will repair completely the test suite. In [44], the authors present a test-suite augmentation technique, based on dependence analysis and partial symbolic execution for regression testing, which can signal if existing test suites are adequate for the changes introduced in a program and, if not, provide guidance for creating new test cases that specifically target the changes in the program. This can be extremely useful but leaves the tester the burden of writing new tests for the evolved software. Regression testing is challenging also in the context of SPLs since it must be made efficient through a test case selection method that selects and, somehow, reuses only the test cases relevant to the changes [45]. In [46], the authors propose an approach to reducing the repetition of test cases during regression testing. To a certain extent, the goal is the same as the approach we propose in this paper, since by reusing test cases we aim at reducing repetitions as well.

*Test scripts repairing* is a widespread practice in the context of Capture-Replay testing. The scripts obtained by Capture–Replay tools are very sensitive to changes in web applications, so they need to be constantly updated and possibly automatically repaired. In that area, there are attempts to devise techniques that try to extract, whenever possible, *models* of the web application. For instance, in [47], the authors try to build a DOM-model which represents the modifications of the web interface. Then they try to use that model to distinguish tests that are retestable but need a repair, those that are obsolete to be discarded, and those that are reusable without modification. Repair can then be performed either by hand or automatically thanks to a series of strategies.

Reusing test cases is recommended by many. In [48], the authors have conducted empirical studies with industrial developers, and observed that repairing test suites is more cost-effective than rewriting or regenerating them from scratch.

There are some approaches that study the evolution of tests when a *model-based testing* approach is applied. In [49], UML class/object diagrams and statecharts, augmented with OCL constraints, are used to model a critical system. Upon an evolution of these models, an automated process classifies tests as outdated, unimpacted, or re-testable ones, which need to be updated and adapted. These concepts can be mapped to our measure of test distance $testDist$. For instance, $testDist = 0$ means that the test is unimpacted. Their methodology, however, does not propose a way to repair existing tests, leaving this task to the test generation tool.

In this paper, we propose the use of the *dissimilarity* for evaluating the impact of test suites evolution. The measurement of the difference between test suites, in the field of model-based testing, has been proposed by [13, 50]. In [13], the authors use a dissimilarity distance very similar to our definition, but only among tests in the same test suite. In their case, higher dissimilarity is pursued, since it indicates that tests are effectively covering products. In [50], the authors study the use of distance functions and machine learning to help to reduce the discard of MBT tests when use case specifications given in CLARET language are updated. They have tried many distance measures, and they claim that using the Hamming function (very close to our definition) between test suites allows testers to decide in a more effective way which tests should be kept after the requirements of the system have changed. A similar definition of *similarity* is proposed in [51] among tests from the same feature model. That definition is extended by [22] and used to evaluate algorithms for product sampling during Continuous Integration. In [22], the authors consider the possibility that the feature set changes. However, unlike the definition proposed in that paper, we focus on the *differences* among tests and we consider only features whose value could actually change. This means that our dissimilarity is more sensitive to changes. For example, two test suites can have a distance of 100% in our approach (i.e., similarity equals to 0), while the similarity defined in [22] is always greater than $|(D_{FM} \cap D_{FM'}) \cup (C_{FM} \cap C_{FM'})|/|F \cup F'|$.

Reusing artifacts upon Feature Models evolution is tackled also in [52], where the authors aim at updating Modal Implication Graphs (MIGs) when constraints are added or removed (while feature sets remain stable). They present an algorithm able to incrementally compute updated MIGs after FMs evolution, and this incremental algorithm is much faster (but the expected MIG is unique). In our case, we allow the evolution of the feature set too, and the computation speed is only one of the possible advantages we evaluate.

Incremental test generation for feature models is presented in [53–55], but in all cases the test suite is incrementally built from scratch, without considering test seeds or existing test suites. All approaches are similar to pMEDICI without any extension since it incrementally combines tuples in order to build test suites.

## 8 CONCLUSIONS

In this paper, we have proposed the GFE technique which is implemented in pMEDICI+ and allows for the incremental generation of combinatorial test suites for FMs. When an FM evolves, GFE exploits the previous test suite and keeps those tests (or their parts) that are still applicable to the evolved FM. In this way, testers may avoid the common problems of the GFS technique, which instead generates test cases from scratch. Furthermore, in this paper, we have introduced the concept of *dissimilarity* that can be used for evaluating the distance between two test suites, e.g., the one obtained from the old FM and the one from the evolved model.

To evaluate the performance of GFE w.r.t. those of GFS, we have used 50 FMs coming from real case studies and compared the dissimilarity, the test suite size, and generation time for each model evolution. Through a series of three research questions, we have shown that using GFE instead of GFS, in general, may impact on generation time but leads to lower dissimilarity and test suite size.

As future work, other measures among evolving test suites could be introduced, for better estimating the cost/benefits of retaining and repairing old tests. For instance, the cost of updating a test may be not linearly linked to the number of features changed in it.

*Acknowledgment.* We would like to thank Luca Parimbelli for the preliminary work done for this project during his master's thesis.

## REFERENCES

[1] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-21, Nov. 1990.

[2] 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.

[3] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 650–670, 2014.

[4] I. D. C. Machado, J. D. Mcgregor, Y. a. C. Cavalcanti, and E. S. De Almeida, "On strategies for testing software product lines: A systematic literature review," *Inf. Softw. Technol.*, vol. 56, no. 10, p. 1183–1199, oct 2014. [Online]. Available: https://doi.org/10.1016/j.infsof.2014.04.002

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

[6] J. Guo, Y. Wang, P. Trinidad, and D. Benavides, "Consistency maintenance for evolving feature models," *Expert Systems with Applications*, vol. 39, no. 5, pp. 4987–4998, apr 2012.

[7] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski, "Evolution of the linux kernel variability model," in *Software Product Lines: Going Beyond.* Springer Berlin Heidelberg, 2010, pp. 136–150.

[8] 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.

[9] M. Mirzaaghaei, F. Pastore, and M. Pezze, "Supporting test suite evolution through test case adaptation," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 231–240.

[10] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (foda) feasibility study," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-90-TR-021, 1990. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231

[11] 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*, vol. 47, no. 1, pp. 1–45, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2580950

[12] F. Ensan, E. Bagheri, and D. Gašević, "Evolutionary search-based test generation for software product line feature models," in *Notes on Numerical Fluid Mechanics and Multidisciplinary Design.* Springer International Publishing, 2012, pp. 613–628. [Online]. Available: https://doi.org/10.1007/978-3-642-31095-9_40

[13] X. Devroey, G. Perrouin, A. Legay, P.-Y. Schobbens, and P. Heymans, "Search-based similarity-driven behavioural SPL testing," in *Proceedings of the Tenth International Workshop on Variability Modelling of Software-Intensive Systems*, ser. VaMoS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 89–96. [Online]. Available: https://doi.org/10.1145/2866614.2866627

[14] A. Calvagna, A. Gargantini, and P. Vavassori, "Combinatorial testing for feature models using CitLab," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops.* IEEE, Mar. 2013. [Online]. Available: https://doi.org/10.1109/icstw.2013.45

[15] D. Cohen, S. Dalal, M. Fredman, and G. Patton, "The AETG system: an approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437–444, jul 1997.

[16] 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*, 2013, pp. 370–375.

[17] "Jenny website," http://burtleburtle.net/bob/math/jenny.html.

[18] "PICT GitHub page," https://github.com/microsoft/pict.

[19] 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).* Los Alamitos, CA, USA: IEEE Computer Society, 2023.

[20] 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)*, 2021, pp. 48–56.

[21] M. Utting and B. Legeard, Eds., *Practical Model-Based Testing.* San Francisco: Morgan Kaufmann, 2007.

[22] 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*, ser. VaMoS '21. New York, NY, USA: Association for Computing Machinery, feb 2021. [Online]. Available: https://doi.org/10.1145/3442391.3442410

[23] H. Hemmati and L. Briand, "An industrial investigation of similarity measures for model-based test case selection," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*, 2010, pp. 141–150.

[24] M. Nieke, "Consistent feature-model driven software product line evolution," Ph.D. dissertation, Technische Universität Braunschweig, 2021.

[25] N. Gámez and L. Fuentes, "Software product line evolution with cardinality-based feature models," in *Top Productivity through Software Reuse*, K. Schmid, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 06 2011, pp. 102–118.

[26] 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)*, vol. 1, 2016, pp. 505–516.

[27] 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*, ser. SPLC '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 76–85. [Online]. Available: https://doi.org/10.1145/2362536.2362550

[28] 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*, vol. 13, p. 21, 08 2019.

[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.*, vol. 87, pp. 119–136, jan 2014. [Online]. Available: https://doi.org/10.1016/j.jss.2013.10.010

[30] *pure::variants User's Guide*, pure-systems GmbH, 2022. [Online]. Available: https://www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf

[31] 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*, vol. 85, no. 10, pp. 2261–2274, 2012.

[32] 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*, ser. ICSE '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 261–270. [Online]. Available: https://doi.org/10.1145/1368088.1368124

[33] 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*, vol. 129, pp. 48–71, 2016, special issue on eLearning Software Architectures. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642316300247

[34] 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*, ser. EASE '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2745802.2745806

[35] 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*, vol. 23, no. 4, p. 67, oct 2016.

[36] S.P.L.O.T., "Repository of real feature models," [Online; accessed 08-September-2022]. [Online]. Available: http://http://52.32.1.180:8080/SPLOT/feature_model_repository_depot.html

[37] A. Bombarda, S. Bonfanti, and A. Gargantini, "Replication package for the paper "On the Reuse of Existing Configurations for Evolving Feature Models"," Jun. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8038553

[38] A. Bombarda and A. Gargantini, "Parallel test generation for combinatorial models based on multivalued decision diagrams," in *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW).*

Los Alamitos, CA, USA: IEEE Computer Society, 4 2022, pp. 74–81. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSTW55395.2022.00027

[39] R. F. Woolson, "Wilcoxon signed-rank test," Sep. 2008. [Online]. Available: https://doi.org/10.1002/9780471462422.eoct979

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

[41] C. Kröher, L. Gerling, and K. Schmid, "Identifying the intensity of variability changes in software product line evolution," in *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*. ACM, Sep. 2018. [Online]. Available: https://doi.org/10.1145/3233027.3233032

[42] R. Tzoref-Brill and S. Maoz, "Modify, enhance, select: Co-evolution of combinatorial models and test plans," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 235–245. [Online]. Available: https://doi.org/10.1145/3236024.3236067

[43] M. Mirzaaghaei, F. Pastore, and M. Pezzè, "Automatic test case evolution," *Softw. Test. Verif. Reliab.*, vol. 24, no. 5, p. 386–411, aug 2014. [Online]. Available: https://doi.org/10.1002/stvr.1527

[44] 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*, 2008, pp. 218–227.

[45] P. A. da Mota Silveira Neto, I. do Carmo Machado, Y. C. Cavalcanti, E. S. de Almeida, V. C. Garcia, and S. R. de Lemos Meira, "A regression testing approach for software product lines architectures," in *2010 Fourth Brazilian Symposium on Software Components, Architectures and Reuse*. IEEE, Sep. 2010. [Online]. Available: https://doi.org/10.1109/sbcars.2010.14

[46] P. Jung, S. Kang, and J. Lee, "Automated code-based test selection for software product line regression testing," *Journal of Systems and Software*, vol. 158, p. 110419, Dec. 2019. [Online]. Available: https://doi.org/10.1016/j.jss.2019.110419

[47] J. Imtiaz, M. Z. Iqbal, and M. U. khan, "An automated model-based approach to repair test suites of evolving web applications," *Journal of Systems and Software*, vol. 171, p. 110841, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220302314

[48] S. Makady and R. J. Walker, "Debugging and maintaining pragmatically reused test suites," *Information and Software Technology*, vol. 102, pp. 6–29, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584917302872

[49] E. Fourneret, F. Bouquet, F. Dadeau, and S. Debricon, "Selective test generation method for evolving critical systems," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, mar 2011.

[50] T. Diniz, E. L. Alves, A. G. Silva, and W. L. Andrade, "Reducing the discard of MBT test cases using distance functions," in *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. ACM, Sep. 2019. [Online]. Available: https://doi.org/10.1145/3350768.3350790

[51] M. Al-Hajjaji, T. Thüm, M. Lochau, J. Meinicke, and G. Saake, "Effective product-line testing using similarity-based product prioritization," *Software & Systems Modeling*, vol. 18, no. 1, pp. 499–521, dec 2016.

[52] S. Krieter, R. Arens, M. Nieke, C. Sundermann, T. Heß, T. Thüm, and C. Seidl, "Incremental construction of modal implication graphs for evolving feature models," in *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A*. ACM, sep 2021.

[53] E. Uzuncaova, S. Khurshid, and D. Batory, "Incremental test generation for software product lines," *IEEE Transactions on Software Engineering*, vol. 36, no. 3, pp. 309–322, 2010.

[54] S. Oster, F. Markert, and P. Ritter, "Automated incremental pairwise testing of software product lines," in *Software Product Lines: Going Beyond*. Springer Berlin Heidelberg, 2010, pp. 196–210.

[55] M. Al-Hajjaji, S. Krieter, T. Thüm, M. Lochau, and G. Saake, "Incling: Efficient product-line testing using incremental pairwise sampling," *SIGPLAN Not.*, vol. 52, no. 3, p. 144–155, oct 2016. [Online]. Available: https://doi.org/10.1145/3093335.2993253