The Journal of Systems and Software 231 (2026) 112645

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

My feature model has changed... What should I do with my tests?☆,☆☆

Andrea Bombarda, Silvia Bonfanti, Angelo Gargantini *

University of Bergamo, Department of Management, Information and Production Engineering, Bergamo, Italy

ARTICLE INFO

Dataset link: https://github.com/fmselab/ctwedge/tree/master/ctwedge.parent/ctwedge.fmtester

Keywords:
Feature models evolution
Specificity
Dissimilarity
Mutation score

ABSTRACT

Software Product Lines (SPLs) evolve over time, driven by changing requirements and advancements in technology. While much research has been dedicated to the evolution of feature models (FMs), less focus has been put on how associated artifacts, such as test cases, should adapt to these changes. Test cases, derived as valid products from an FM, play a critical role in ensuring the correctness of an SPL. However, when an FM evolves, the original test suite may become outdated, requiring either regeneration from scratch or repair of existing test cases to align with the updated FM. In this paper, we address the challenge of evolving test suites upon FM evolution. We introduce novel definitions of test suite dissimilarity and specificity We use these metrics to evaluate three test generation strategies: GFS (generating a new suite from scratch), GFE (repairing and reusing an existing suite), and SPECGEN (maximizing specific tests for the FM evolution). Additionally, we introduce a set of mutations to simulate FM evolution and obtain additional FMs. By using mutants, we conduct our analyses and evaluate the mutation score of test generation strategies. Our experiments, conducted on a set of FMs taken from the literature and on more than 3,200 FMs artificially generated with mutations, reveal that GFE often produces the smallest test suites with high mutation scores, while SPECGEN excels in specificity, particularly for mutations expanding the set of valid products.

1. Introduction

Software product lines (SPLs), like all software models and artifacts, naturally evolve over time due to various factors (Kröher et al., 2018; Bürdek et al., 2016; Pleuss et al., 2012; Passos et al., 2013; Marques et al., 2019). Consequently, the associated feature model (FM) for the SPL may also undergo changes. While significant research has explored the evolution of FMs, less focus has been placed on how other related artifacts should adapt to these changes (Botterweck and Pleuss, 2014). Indeed, when evolving SPLs, it is crucial to consider the evolution of other core assets such as architecture, documentation, and test cases (Seidl et al., 2012). Evolving these artifacts not only enhances product quality but also fosters scalability and maintainability within the product line (Montalvillo and Díaz, 2023; Ignaim et al., 2024; Hernández-López et al., 2018).

In this paper, we focus on how test cases which are (valid) products derived from an FM, should or could evolve. We assume that the designer of a SPL, once an FM has been designed for it, is interested in deriving a set of products, i.e., valid configurations, according to some criteria (e.g., combinatorial interaction among features (Oster et al., 2010) or product sampling (Varshosaz et al., 2018)), and these products constitute the test cases. We assume that each test case is then used to test the SPL (for example, by actually building the single product and checking its correctness and feasibility). If the SPL, together with its FM, is modified, the original test set must likely be modified, since some products are no longer valid while others must be considered due to the modifications of the FM.

When FM evolution happens, testers have two options: they can either start fresh and generate new test cases from scratch, or they can attempt to repair test cases generated for the previous version of the FM and retain those that are still valid. However, in the literature, there is no clear guideline on how to handle test cases when an FM undergoes evolution. This is the rationale behind this paper which extends two previous works (Bombarda et al., 2023, 2024). More specifically, the following are the contributions of our paper. Given an arbitrary evolution e of an FM of the SPL under test, we introduce:

1. Novel definitions of *distance* and *dissimilarity* between tests and test suites, which do not require to complete the test suite derived from each FM. These definitions, initially proposed in Bombarda et al. (2023) to evaluate the difference between two tests or test suites generated for two FMs, originally required

☆ This article is part of a Special issue entitled: 'Syst.+Sw.ProductLineEng.' published in The Journal of Systems & Software.
☆☆ Editor: Prof Raffaela Mirandola.

* Corresponding author.
E-mail address: angelo.gargantini@unibg.it (A. Gargantini).

https://doi.org/10.1016/j.jss.2025.112645
Received 19 February 2025; Received in revised form 6 July 2025; Accepted 19 September 2025
Available online 8 October 2025

modifying one of the two counterparts to include removed or added features for their computation;
2. A revised definition of *specific* tests for the evolution e. This definition, introduced in Bombarda et al. (2024), aims to assess how much a test suite focuses on covering new products introduced during FM evolution. In the original definition, it considered both non-selected and non-available features in the same manner. However, in this work, we distinguish these two scenarios;
3. A set of *mutations* useful to simulate the effect of the evolution e;
4. A set of *experiments* for determining the most appropriate test suite generation or repair approach based on the evolution e, depending on the updates implemented during e.

In this paper, we analyze three different test generation policies upon FM evolution: GFS, which is based on generating a new test suite from scratch; GFE, which starts the test generation from the test suite previously generated for the old version of the FM; and SPECGEN, which tries to maximize the percentage of *specific* tests.

We conducted a set of experiments, run over 13 different industrial FM families taken from the literature, with a total of 35 evolutions. Additionally, for this paper, we generated 3256 additional FMs starting from the industrial ones and by introducing artificial mutations. We assessed the average generation time, test suite size, dissimilarity, specificity, and mutation score for the test suites generated by the three policies. We found that, in general, GFE is the policy leading to the smallest test suites, with the lowest dissimilarity and the highest mutation score. However, it suffers, in terms of time, for simple FMs, where the pre-processing time overpasses that of the pure test generation. SPECGEN proved to be the test generation policy with the highest specificity, as it maximizes the number of test cases valid in the evolved FM but invalid in the original FM. In general, we observed that the metrics we assess for each test generation approach are significantly influenced by the type of evolution to which the FM is subjected. For instance, in terms of time, SPECGEN outperformed the other policies when mutations broaden the set of valid products. Thanks to our findings, we provide a set of guidelines to choose the best test generation approach depending on the changes implemented during the evolution.

The remainder of the paper is structured as follows. Section 2 introduces the background for our work, starting from FMs to the definition of what are tests in the SPL domain. In Section 3, we present the problem of SPL evolution and the way in which it can be simulated by using artificial mutations. Section 4 details, the metrics we use to compare different test generation strategies, while Section 5 presents the three policies we analyze in this paper. The experimental methodology and the results of our experiments are presented in Section 6. Section 7 discusses potential threats to the validity of our experiments and conclusions, while in Section 8 we report related works. Finally, Section 9 concludes the paper.

2. Background

This section offers an overview of fundamental concepts related to feature models, the testing methods applied to them, including combinatorial testing, and how decision diagrams (DDs) can be used to represent feature models and their valid configurations.

2.1. Feature models

In software product line engineering, feature models (Kang et al., 1990; Thüm et al., 2014) are used to represent all possible products within a Software Product Line (SPL) by defining features and the relationships between them. Specifically, a feature model (in the following referred to as FM) consists of a structured set of features F, arranged in a hierarchical format. Each parent–child relationship in this hierarchy establishes a constraint that assigns a feature or group of features to one of the following categories:

Table 1
Example of a test suite for the FM reported in Fig. 1.

	Screen	Basic	Color	Media	Camera	MP3	Calls	GPS
t_1	⊤	⊥	⊤	⊥	⊥	⊥	⊤	⊥
t_2	⊤	⊤	⊥	⊤	⊥	⊤	⊤	⊤
t_3	⊤	⊥	⊤	⊤	⊤	⊤	⊤	⊥
t_4	⊤	⊤	⊥	⊥	⊥	⊥	⊤	⊤
t_5	⊤	⊥	⊤	⊤	⊤	⊥	⊤	⊤
t_6	⊤	⊤	⊥	⊥	⊥	⊥	⊤	⊥

- *Or*, when at least one of the sub-features must be selected if the parent is selected;
- *Alternative*, when exactly one of the children must be selected whenever the parent feature is selected;
- *And*, when the relation between a feature and its children is neither an *Or* nor an *Alternative*. In this case, each child of an *and* must be either: – *Mandatory*, when the child feature is selected whenever its respective parent feature is selected; – *Optional*, if the child feature may or may not be selected if its parent feature is selected.

Among all features, only the *root* in F has no parent and it must be selected in every product. An example of an FM is reported in Fig. 1. It has 9 features, among which *MobilePhone* is the root, *Screen* and *Calls* are mandatory, *Media* and *GPS* are optional, the group having *MobilePhone* as a parent is an And group, the one having *Media* as a parent is an Or group and, finally, the one having *Screen* as a parent is an Alternative group.

Additionally, the set of valid products can be restricted through *cross-tree constraints*, i.e., relations that cross-cut hierarchy dependencies. These constraints can be *general* propositional formulas, with the most common being A *requires* B ($A \rightarrow B$), i.e., when the selection of a feature A in a product also implies the selection of the feature B, or A *excludes* B ($A \rightarrow \neg B$), i.e., when features A and B cannot be part of the same product. The FM reported in Fig. 1 has a cross-tree constraint requiring *Color* to be selected when *Camera* is selected.

Because of the cross-tree constraints, a feature can be *dead*, i.e., it can never be selected, or *core*, i.e., it is mandatory in every valid product, like the *root*.

2.2. Test suites for feature models

When designing a family of products through an SPL, it is paramount to test it to detect possible faults (Arcaini et al., 2015). Test suites for FMs are composed of (abstract) tests defining which features of the FM are selected and which ones are unselected by each product.

Definition 1 (*Test*)**.** A test ts for FM is a function mapping the selection status of each feature $f \in F$ in the product identified by ts:

$$ts(f) = \begin{cases} \top & if\ f\ is\ selected\ in\ ts \\ \bot & if\ f\ is\ not\ selected\ in\ ts \end{cases}$$

Given an FM, tests can be *valid* or *invalid*. The former case is that of tests defining the selection status for all the features in $f \in F$ and complying with the hierarchical and cross-tree constraints. The latter, instead, is the case of tests representing products that cannot be built because of noncompliance with constraints.

Table 1 reports a test suite for the FM shown in Fig. 1. In this representation, we can omit the *root* feature because it must be selected by all valid products in any FM. Test cases for FMs can be generated by following several criteria. However, normally, the number of tests for an FM is too high to perform exhaustive testing, thus product sampling (Varshosaz et al., 2018), search-based (Ensan et al., 2012; Devroey et al., 2016), and combinatorial testing (Johansen et al., 2011; Bombarda et al., 2024, 2023; Calvagna et al., 2013; Bombarda and

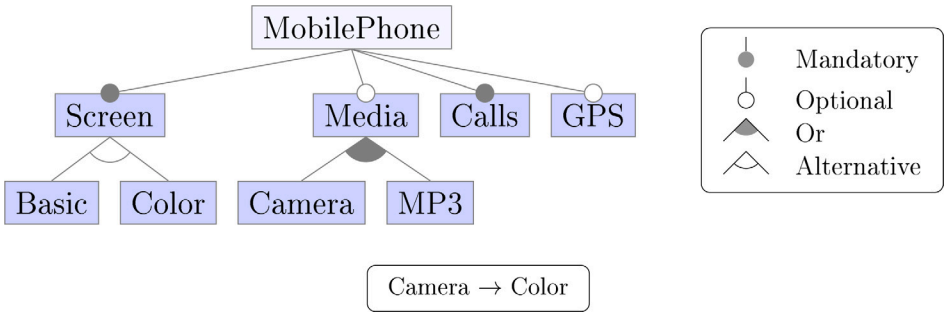


Fig. 1. Example of an FM for a Mobile Phone.

Model MobilePhone

Parameters :
MobilePhone : Boolean
Screen : { Basic Color NONE }
Media : Boolean
Camera : Boolean
MP3 : Boolean
Calls : Boolean
GPS : Boolean

Constraints :
MobilePhone == TRUE
Screen != NONE <=> MobilePhone == TRUE
Media == TRUE => MobilePhone == TRUE
Media == TRUE => Camera == TRUE || MP3 == TRUE
Camera == TRUE => Media == TRUE
MP3 == TRUE => Media == TRUE
Calls == TRUE <=> MobilePhone == TRUE
GPS == TRUE => MobilePhone == TRUE
Camera == TRUE => Screen == Color

Listing 1. Input Parameter Model for the FM in Fig. 1.

Gargantini, 2024a) approaches have been proposed to allow testers to select only valid products and keep the size of the test suite under control.

2.3. Feature models and combinatorial input parameter models

In this paper, we generate combinatorial test suites (Kuhn et al., 2013) for FMs. The aim is, given a strength t, to generate a test suite TS covering all t-wise interactions between features $f \in F$ by means of a combinatorial test generator and starting from an Input Parameter Model (IPM).

Mapping FMs to IPMs is straightforward. Each feature $f \in F$ is translated into a Boolean parameter for the IPM, and each cross-tree constraint in the FM is converted into a logical constraint between parameters in the IPM, as well as each constraint defining hierarchical relations and mandatory features. Additionally, if a more compact representation of the features is needed, *alternative* groups can be represented with a single parameter. Listing 1 reports the IPM representation of the FM in Fig. 1: Each feature is converted into a Boolean parameter (except the alternative ones, which are converted into an enumerative parameter) and a set of constraints limiting the set of valid products (i.e., those setting features as mandatory, or those defining the hierarchical dependency among features, or the cross-tree ones) is added.

Once an IPM for the FM under analysis is written, it can be used by a combinatorial test generator (e.g., Czerwonka, 2022a; Anon, 2023; Yu et al., 2013; Bombarda and Gargantini, 2022, 2023; Garvin et al., 2009; Wagner et al., 2020; Gargantini and Vavassori, 2014) and, given a strength t, a test suite like that shown in Table 1 is generated. In particular, Table 1 shows a pairwise test suite, meaning that every valid interaction between pairs of features is covered.

2.4. Mapping feature models to decision diagrams

An efficient way to represent FMs is by means of decision diagrams (DDs), which are defined as follows.

Definition 2 (*Decision Diagram*)**.** A decision diagram is a graph that represents a function $g : D \to B$ where $D = D_1 \cdots D_n$ and B is the Boolean domain, i.e., $B = \{F, T\}$.

In general, a DD is used to evaluate the truth value of a function g when it is applied to the variables $x_1, \ldots, x_n$. Depending on the domain of the variables $x_1, \ldots, x_n$, different DDs can be used: Binary Decision Diagrams (BDDs) (Akers, 1978) are used to represent Boolean functions, while Multivalued Decision Diagrams (MDDs) (Miller and Drechsler, 2002) support variables with different domains.

Summarizing, a DD can select which values of the input domains $D_1 \ldots D_n$ are selected by the function g. In fact, if the values $x_1, \ldots, x_n$ for the variables in $D_1 \ldots D_n$ are selected by g, then $g(x_1, \ldots, x_n) = T$, otherwise $g(x_1, \ldots, x_n) = F$.

FMs can be considered as models describing which configurations (i.e., truth assignments to all the features) are acceptable and which are not. Thus, a DD can represent the set of possible products for an FM. Fig. 2 reports an example of the mapping between an FM and a DD. Note that, in this case, all features in FM are Boolean (i.e., no alternative feature exists) so using an MDD or a BDD will produce the same DD. In this example, FM in Fig. 2(a) has a *root*, a *mandatory* feature (A), and an *optional* feature (B). In its DD representation, shown in Fig. 2(b), dashed lines indicate that the feature from which the arrow starts is *unselected*, while continuous lines represent *selected* features.

In this paper, we will use both BDDs and MDDs to represent FMs and derive test cases, as there are several works available in the literature confirming their optimal performance w.r.t. most classically used logical solvers (Heß et al., 2021; Bombarda and Gargantini, 2024b). Their use for deriving test cases is made possible by the operations among DDs such as *complement*, computation of the *cardinality*, or the most classical binary operations like *union*, *intersection*, and *difference*. These operations are equivalent to logic operations between the functions represented by the DDs.

Given the representation of an FM with a DD, valid test cases for the FM are all those paths leading to the T leaf. Similarly, DDs can be used to model t-tuples of features to be covered by a test case: We simply force all assignments contained in the t-tuple tp as leading to the T leaf, and all the others to the F leaf.

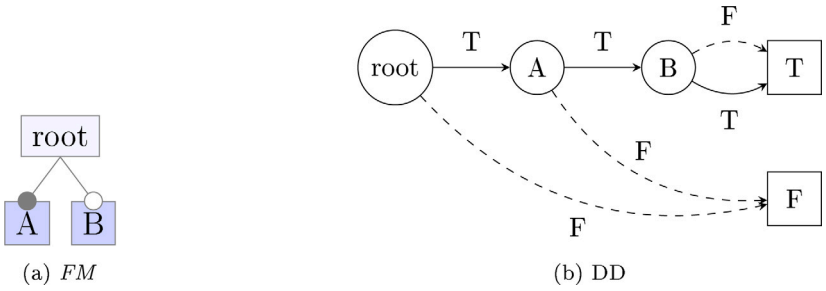


Fig. 2. Correspondence between an FM and a DD.

Table 2
Mutation operators for FM.

Operator name	Description
`MissingFeature`	A feature is removed
`ConstraintRemover`	A cross-tree constraint is removed
`OptToMan`	Optional feature → mandatory
`ManToOpt`	Mandatory feature → optional
`AlToAnd`	Alternative → AND (+ mand. children)
`AlToAndOpt`	Alternative → AND (+ opt. children)
`AlToOr`	Alternative → OR
`AndToAl`	An AND group → alternative
`AndToOr`	An AND group → OR
`OrToAl`	An OR group → alternative
`OrToAnd`	An OR group → AND (+ mand. children)
`OrToAndOpt`	An OR group → AND (+ opt. children)
`Negation`	A cross-tree constraint is negated
`LogicOrToAnd`	The or operators in a cross-tree constraint are changed in and
`ConstraintSubst`	A general constraint is modified by inserting a new feature, changing a logical operator, or removing part of it
`LiteralChg`	A randomly chosen literal is changed in the FM
`ImpliesToIff`	Implies is swapped in iff in a constraint
`RequiresToExcludes`	A requires constraint → excludes
`MoveFeature`	A feature is moved (with its descendants) as a child of another feature

3. Problem definition

In this paper, we address the evolution of FMs. Specifically, we consider a scenario where an initial feature model FM evolves into a new version FM'. In such cases, testers must decide how to handle a possible test suite previously generated for FM. In this section, we present the fundamental concepts related to FM evolution and explain how mutations can be used to simulate this process, and discuss key properties of test suites that are relevant in the context of model evolution.

3.1. Evolution and types of edits of a feature model

SPLs and their FMs evolve throughout their lifespan (Thüm et al., 2009; Lotufo et al., 2010). The evolution of an FM involves editing actions like adding, removing, or relocating features, as well as modifying, adding, or removing constraints. Even small changes can affect the set of permissible feature combinations: Configurations that were once valid may no longer be, while others may now become possible. In the literature, several examples of changes have been presented, mostly taken from industrial FM edits (Thüm et al., 2009; Alves et al., 2006; Passos et al., 2013; Lotufo et al., 2010). In this work, we assume that, when necessary, we can simulate those edits by using the mutation operators introduced in Arcaini et al. (2019) and here reported in Table 2.

The evolution process of an FM changes the set of valid 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 (Thüm et al., 2009). Consequently, tests derived for FM may become invalid for FM', new ones may need to be added to satisfy the testing requirements and testers should decide what to do with the old test cases. This is the motivation behind the analysis we propose in this paper.

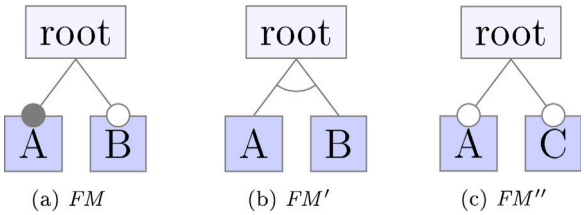


Fig. 3. Example of FM evolutions.

Example 1. Let us consider the FM in Fig. 3. From FM to FM', the set of features remains unchanged, but the hierarchical relation changes: the *And* becomes an *Alternative*. This means that a test/product $t_1 = \{root, A, B\}$, valid for FM, will not be valid for FM', since in FM' both features, A and B, cannot be selected.

Evolutions changing the feature set. Sometimes a change in an FM also changes the feature set by removing or adding a new feature. For instance, in Fig. 3, the feature B has been removed from FM' while a new feature C is added to obtain FM''. This means that a test for the original FM must be adapted to be used in the new FM, or on the contrary, a test in the new FM must be adapted if it is used in the old FM. We will return to this subject in Section 4.4.

4. Metrics for comparing test generation strategies

To support users in deciding which test generation policy is the best fit for their scenario, we introduced the following metrics to be considered according to the final goal: Generation time, test suite size, dissimilarity, specificity, and mutation score. In this section, we will provide more details on each of the metrics we consider in this paper. Throughout this section, we will use the example in Fig. 3 and the three test suites in Fig. 4 to illustrate relevant metrics.

4.1. Test suite generation time

One of the metrics used in choosing a generation strategy is the generation time. This is commonly done in the literature when different test generation tools or strategies are compared (Leithner et al., 2024; Bombarda et al., 2021). Commonly, the goal for testers is to minimize the generation time because it allows them to save resources that can otherwise be used for other activities.

	A	B
t_1	⊤	⊤
t_2	⊤	⊥

(a) A test suite TS for FM

	A	B
t_3	⊥	⊤
t_4	⊤	⊥

(b) A test suite TS' for FM'

	A	C
t_5	⊤	⊤
t_6	⊤	⊥
t_7	⊥	⊤
t_8	⊥	⊥

(c) A test suite TS'' for FM''

Fig. 4. Test Suites for the FM evolutions in Fig. 3.

4.2. Test suite size

The dimension of the test suite, i.e., the number of test cases, directly impacts the test execution time. For this reason, test suites with few test cases are preferable for systems that require considerable effort in terms of resources deployed.

Example 2. The size of this test suite TS in Fig. 4(a) is the number of test cases it contains, i.e., $\|TS\| = 2$.

4.3. Distance between two test suites: Dissimilarity

Introduced in Bombarda et al. (2023), the dissimilarity evaluates the difference between two test suites generated starting from a feature model FM and its evolution FM'. Since the distance between TS and TS' depends on the distance between each test t and t' in them, first we introduce the definition of the distance between two test cases.

Definition 3 (*Distance Between Two Test Cases*)**.** Let t and t' be two test cases generated from FM and FM', $(F \cap F')$ be the common features between FM and FM', and $F \Delta F'$ be the symmetric difference between features in FM and FM'.[1] The distance between the two test cases is:

$$testDist(t,t') = \|\{f \mid f \in (F \cap F') \wedge t(f) \neq t'(f)\}\| + \|F \Delta F'\| \quad (1)$$

Intuitively, the distance between two test cases counts how many features of two products, one for FM and the other for FM', are different from each other. In particular, it counts the number of common features where $t(f)$ and $t'(f)$ differ, then the number of elements in the symmetric difference between the feature sets of FM and FM' is added. The maximum distance between two tests is obtained when all the common features $t(f)$ and $t'(f)$ differ. Thus, the maximum possible $testDist(t,t')$ is equal to $\|F \cap F'\| + \|F \Delta F'\|$, which is equal to $\|F \cup F'\|$. The minimum distance is equal to $\|F \Delta F'\|$, i.e., when all the common features are equally either selected or not in both test cases.

Example 3. The distance $testDist(t,t')$ between the test cases in TS and TS' of Fig. 4 is computed as follows.

$t \in TS$	$t' \in TS'$	$F \cap F'$	$t(f) \neq t'(f)$	$F \Delta F'$	$testDist(t,t')$
t_1	t_3	$\{A,B\}$	$\{B\}$	$\emptyset$	1
t_2	t_4	$\{A,B\}$	$\emptyset$	$\emptyset$	0

After defining the distance between two tests in TS and TS', we define the distance between two test suites.

[1] The symbol Δ denotes the operation of symmetric difference of two sets: $A \Delta B = (A \setminus B) \cup (B \setminus A)$. The symmetric difference is the set of elements that are in either of the sets A and B, but not in their intersection.

Definition 4 (*Distance Between Test Suites*)**.** Given two test suites $TS = \{t_1 ... t_n\}$ derived from FM and $TS' = \{t'_1 ... t'_{n'}\}$ derived from FM', the distance between TS and TS' is defined as

$$testSuiteDist(TS,TS') = \sum_{i=1}^{\min(n,n')} testDist(t_i,t'_i) + |n-n'| \cdot \|F \cup F'\| \quad (2)$$

The distance between two test suites is the sum of the distances between two test cases t_i and t'_i, where i goes from 1 to the minimum between n and n'. If there are tests for which it is not possible to compute the test distance, i.e., n is greater than n' or vice versa, we increase the test distance by the number of excess tests multiplied by the number of features in FM and FM' (representing the maximum distance).

Example 4. Let us consider the test distances computed in Example 3. As the two test suites have the same number of test cases, the distances between TS and TS' is simply obtained by summing all $testDist(t,t')$. Thus, $testSuiteDist(TS,TS') = 1$.

Note that the test distance depends on the order in which test cases are considered. Thus, if we shuffle the test cases in TS', we obtain different values, as reported in the following table:

$t \in TS$	$t' \in TS'$	$F \cap F'$	$t(f) \neq t'(f)$	$F \Delta F'$	$testDist(t,t')$
t_1	t_4	$\{A,B\}$	$\{B\}$	$\emptyset$	1
t_2	t_3	$\{A,B\}$	$\{A,B\}$	$\emptyset$	2

In this case, $testSuiteDist(TS,TS') = 3$.

As defined above, the distance between two test suites of different sizes depends on the *order* in which the tests are considered when comparing the two tests suites. A shuffling of the larger test suite can increase or decrease its distance from the smaller test suite. To avoid this undesired effect, we formally define the dissimilarity as follows:

Definition 5 (*Dissimilarity*)**.** Given the test suites $TS = \{t_1 ... t_n\}$ derived from FM and $TS' = \{t'_1 ... t'_{n'}\}$ derived from FM', the dissimilarity is the minimum distance between TS and TS':

$$dissimilarity(TS,TS') = \min_{shufflingTS,TS'} testSuiteDist(TS,TS')$$

Intuitively, the dissimilarity refers to the minimum distance between test suites, determined after calculating the distances for all possible permutations of test cases within the two test suites. The dissimilarity aims at measuring the effort required to modify the artifacts derived from the test suite TS in order to adapt them to the new test suite TS'. The definition is inspired by Devroey et al. (2016), Pett et al. (2021) 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 Hemmati and Briand, 2010) is generally more effective than other distance measures. In our approach, features that are either removed or added, i.e., they belong to $\|F \cup F'\|$, contribute to increase the distance among tests, regardless if they are selected or not. This is because the addition or removal of a feature requires the modification of the test.

Example 5. Let us consider the two distances between TS and TS' computed in Example 4. Since both TSs contain two test cases, the two distances $testSuiteDist(TS, TS')$ we computed cover all possible orderings. Thus, the dissimilarity is the minimum between them. i.e., $dissimilarity(TS, TS') = 1$

4.4. Specificity

We introduce the specificity (Bombarda et al., 2024) to measure if a test suite is suitable to target the edits applied to a feature model FM in its evolution to FM'.

Definition 6 (*Specific Test - Same Feature Set*)**.** Given the evolution of the feature model FM to FM' with the same feature set $F = F'$, we say that a test t is *specific* if and only if t is valid for FM' and it is not in FM.

In this way, when a test is newly introduced for FM', it is considered specific. It clearly tests a product that before was not expected.

Example 6. Let us consider the evolution from FM to FM' in Fig. 3. Let us consider the test $t = \{A : \bot, B : \top\}$ (valid configuration) for FM', since it is not a valid test also for FM, it is specific for FM'.

In case the modified feature model FM' changes also the feature set w.r.t. FM, checking if a test t is specific for testing such changes, is more complex since the tests for FM cannot be applied as they are to FM'. In general, when we want to apply a test t originally generated for FM_1 (with feature set F_1) to another future model FM_2 (with a different feature set F_2), we can assume that all the features in FM_2 but not in the original FM_1 must be considered as not selected in t. Formally, given t a test for FM_1, we define a test t_{proj} (projection of t) for FM_2 as:

$$t_{\text{proj}}(f) = \begin{cases} t(f) & f \in (F_1 \cap F_2) \\ \bot & f \in (F_2 \setminus F_1) \end{cases} \tag{3}$$

Intuitively, t_{proj} is equal to t for all the features that were preserved, while it considers not selected all the features removed from FM_1. Note that the projection of a valid test could be invalid in another FM. By exploiting the projection of a test, we introduce the following definition of specific tests:

Definition 7 (*Specific Test - Different Features*)**.** Given the evolution of the feature model FM to FM' with different feature sets F and F', we say that a test t is *specific* if t selects a new feature $f \in F'$ where $f \notin F$, i.e., $\exists f \in F' \setminus F : t(f) = \top$ or if its projection t_{proj} would be invalid in FM.

Example 7. Let us consider the evolution from FM' to FM'' in Fig. 3. During this evolution step, a feature (i.e., B) is removed and a new feature (i.e., C) is added. Thus, to understand whether a test is specific, we need to consider the projection of test cases, as reported in the following table:

t	New features selected?	t_{proj} for FM'	Valid in FM'?	Is specific?
t_5	✓	–	–	✓
t_6	✗	$\{A : \top, B : \bot\}$	✓	✗
t_7	✓	–	–	✓
t_8	✗	$\{A : \bot, B : \bot\}$	✗	✓

Definition 8 (*Specificity of a Test Suite*)**.** Given a test suite TS, we define the specificity as the number of tests that are specific to test FM' divided by the total number of tests.

Example 8. Let us consider the discussion given in Example 7. The specificity of TS'' is $spec(TS'') = 3/4 = 0.75$.

4.5. Mutation score

Another measure to evaluate the quality of a test suite is its mutation score (Woodward, 1993). In our case, given a feature model FM, the mutation score of a test suite can be computed by generating N mutants of FM thanks to the operators presented in Section 3.1, and by counting the number of mutants that are killed. As defined in the literature, a mutant is killed by a test suite TS if at least a test t in TS becomes invalid because of that mutant. The mutation score will be the number of killed mutants divided by N. A test suite that achieves a high mutation score demonstrates its ability to detect subtle changes (mutants) in the FM under analysis. This implies the test suite is more likely to detect real faults introduced during development or evolution.

5. Policies

In this section, we present the test generation strategies we have identified for testing evolving FMs, and we highlight their pros and cons. More specifically, when a model FM evolves in a new FM', testers may choose to generate a new test suite from scratch (GFS Calvagna et al., 2013), to start from an existing test suite (GFE Bombarda et al., 2023), or to focus the new test suite on having specific characteristics, such as the highest specificity (SPECGEN Bombarda et al., 2024).

5.1. Generation from scratch (GFS)

When the FM evolves, the typical approach for obtaining a new test suite is to generate tests *from scratch*. In this process, the evolved FM is provided as input to a test generator, which produces an entirely new test suite. Despite being trivial, this technique presents many drawbacks:

- Some of the test cases previously generated for FM may be still valid for FM', but GFS does not take them into account;
- For complex FMs, regenerating the whole test suite may require considerable time;
- Tests previously manually written for FM to test specific critical configurations (which are still valid for FM') may be lost, while it would be better to keep them;
- Old and discarded test cases cannot be used anymore for regression testing (Legunsen et al., 2016).

These issues can be addressed if the test generation for FM' starts from already existing test cases, as proposed by the GFE approach explained in the following.

5.2. Generation from existing tests (GFE)

As previously discussed, reusing test suites upon FM evolution can provide several advantages and solve most of the limitations of the GFS approach. In this section, we present our GFE policy which exploits the test generation from seeds (Bombarda and Gargantini, 2022, 2023). In principle, generating tests starting from seeds is claimed to be important (Cohen et al., 1997), and supported by most combinatorial test generators, such as ACTS (Yu et al., 2013), jenny (Anon, 2023), PICT (Czerwonka, 2022a), and pMEDICI (Bombarda and Gargantini, 2023). However, not all tools support using invalid tests as seeds. In the case of FM evolution, instead, some tests may become invalid because of the restriction of the set of valid products, some feature is removed, or new features are added. For this reason, we here report how we have implemented the GFE policy (Bombarda et al., 2023) and supported invalid tests as seeds in pMEDICI, which generates combinatorial test suites by mapping the FM into the corresponding DD representation IPM (as introduced in Section 2.3).

Test suite pre-processing and repairing. At the beginning of test generation for the feature model FM', all tests in TS, originally generated for FM, are added to the seed set. For each test t in TS, we construct a new test case by including only the assignments from t that remain valid in FM'. This reconstruction is carried out, for each test t, feature by feature. If a feature f no longer exists in FM', it is skipped. If the current assignment to f is still valid (it can be easily checked by using the DD representation of both tests and FM', as reported in Section 2.4) the assignment is retained in the test under construction. Otherwise, the assignment is discarded, and we proceed to the next one. At the end of this process, we will have a set of tests partially filled with the old tests of TS and, then, the actual test generation of pMEDICI can start.

Test completion. During the test completion phase for the GFE approach, the test suite previously produced is enhanced to cover all t-tuples of interest. This process is carried out thanks to the internal structure of the pMEDICI tool: Each test case is stored in a structure called *text context*, which contains all assignments committed to the single test so far and all constraints represented through a DD (see Section 2.4). Thus, after the pre-processing phase, a set of *test context*s is created and each of them contains the assignments from TS still valid for FM'. By starting from this set of assignments, pMEDICI considers all t-tuples for FM' needing to be covered. For each t-tuple, if it is not already covered, we add it to an existing test context (if possible) or we build a new test context (and hence a test). In the end, pMEDICI produces a test suite TS' that is valid and achieves the combinatorial coverage for FM', but it reuses the old test suite TS as much as possible, thanks to the test suite pre-processing and repairing process previously described: Test seeds, which are derived from the original test suite generated for FM, are used when generating a test suite for FM'.

5.3. Generation of specific tests (SPECGEN)

Upon FM evolution, one may generate a test suite still achieving the desired t-wise coverage, but with most of the test cases focusing on new products and, thus, being specific as defined in Section 4.4. For this reason, in Bombarda et al. (2024) we introduced SPECGEN, a DD-based approach generating test suites maximizing the specificity.

The procedure implemented by the SPECGEN approach is reported in Algorithm 1.

It starts by considering the two FMs, FM and its evolution FM', and the set of all t-tuples of features to be covered (i.e., those that can be derived from FM'). First, $dd_{initial}$ (line 1) is computed by intersecting the negation of $dd(FM)$ and $dd(FM')$. This DD represents all the possible specific tests we are looking for. However, the evolution of FM in FM' may lead to having only non-specific test cases (see the definition in Section 3). Thus, the cardinality of the DD obtained with the intersection (line 7) may be null and, in that case, the test generation is performed without considering specific tests.

Then, the process of *collecting* all the t-tuples tp in TP (line 9–19) is performed. During this process, the compatibility of each t-tuple tp with FM' is checked. As done for the previous DD, the compatibility is checked by using the intersection between the DD representing tp, i.e., $dd(tp)$, and the one of FM', i.e., $dd(FM')$, at line 10, and computing its cardinality. If tp can be covered and specific tests can be generated, the `tryToCover` function, at line 14, is called. The algorithm implemented by this function is reported in Algorithm 2 and it is an instantiation of the *monitoring* strategy described in Bombarda and Gargantini (2020): It checks whether a test generated for one set of t-tuples inadvertently covers other t-tuples, thereby minimizing the overall size of the final test suite. This function tries to cover the t-tuple tp using one of the already generated tests in TS, if possible (line 3–6), or to create a new test starting from dd_{notp} (line 8–11). In the first scenario, given t_{dd} as the DD representing the test that can cover tp, t_{dd} is updated by intersecting it with the DD of the t-tuple tp, denoted as tp_{dd} (line 4). In the second scenario, the test suite TS is expanded by adding a new DD corresponding to the test derived from the intersection of dd_{notp} and the DD of the t-tuple tp, i.e., tp_{dd} (line 9).

Resuming analyzing Algorithm 1, in case `tryToCover` succeeds in covering tp, the next t-tuple is analyzed. Otherwise, the `tryToCover` function is executed over the set of non-specific test cases (line 18). At the end, after having iterated over all possible t-tuples, the union between the DDs representing specific tests T_s and non-specific tests T_{ns} is computed (line 20). By extracting from each of the DDs a single test case (i.e., a path leading to the T leaf), the algorithm obtains a test suite maximizing specificity and achieving the t-wise coverage.

Algorithm 1 Algorithm generating specific combinatorial test suites.

Input: FM the original feature model
Input: FM' the evolved feature model
Input: TP the set of all the t-tuples derivable from FM'
Output: the specific test suite
▷ Initial DD from which specific tests can be derived
1: $dd_{initial} \leftarrow \neg dd(fm) \wedge dd(FM')$
2: **if** $size(dd_{initial}) = 0$ **then** ▷ FM' only restricts the set of valid products
3: $skipSpecific \leftarrow$ **true**
4: **else**
5: $skipSpecific \leftarrow$ **false**
6: **end if**
7: $T_s \leftarrow \emptyset$ ▷ Set of DDs for specific tests
8: $T_{ns} \leftarrow \emptyset$ ▷ Set of DDs for non-specific tests
9: **for all** $tp \in TP$ **do** ▷ Iterate over all the t-tuples
10: **if** $size(dd(tp) \wedge dd(FM')) = 0$ **then** ▷ Check the validity of the t-tuple
11: **continue next** tp
12: **end if**
13: **if** $\neg skipSpecific$ **then**
▷ Look for a specific test that can cover tp
14: **if** tryToCover($dd(tp)$,T_s,$dd_{initial}$) **then**
15: **continue next** tp
16: **end if**
17: **end if**
▷ No specific test can cover tp
18: tryToCover($dd(tp)$,T_{ns},$dd(FM')$)
19: **end for**
20: $TS \leftarrow T_s \cup T_{ns}$
21: **return** $TS.forEach().getTestCase()$

Algorithm 2 Function trying to cover a t-tuple.

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

6. Experiments

In this section, we compare the three test generation policies proposed in Section 5 and we evaluate them by using the metrics we discussed in Section 4. We introduce our experimental methodology in Section 6.1, our results in Section 6.2 and, finally, we discuss them in Section 6.3. During our experiments, we considered the following research questions:

RQ1 How does the test suite generation time relate with the test generation policy?
RQ2 Is there a correlation between the test suite size and the chosen test generation policy?

Table 3
List of the FM evolution examples from the literature.

Example	V	M	#F	#P	Ref.	Example	V	M	#F	#P	Ref.
AmbAssistLiving	2	214	24–32	$9.8{\cdot}10^4$–$5.0{\cdot}10^7$	Gámez and Fuentes (2011)	SmartHotel	2	90	6–8	6–30	Arcega et al. (2016)
AutomotiveMult.	3	178	6–13	5–192	Seidl et al. (2012)	Smartwatch	2	145	12–15	96–192	Ali and Hoing (2019)
Boeing	3	90	5–6	2–2	White et al. (2014)	WeatherStat.	2	181	22–23	528–660	Anon (2022b)
CarBody	4	213	6–13	4–40	Pleuss et al. (2012)	MobileMedia	6	524	11–26	2–272	Figueiredo et al. (2008)
Linux (Simple)	3	115	5–10	7–33	Nieke (2021)	HelpSystem	2	193	25–26	$672{\cdot}10^3$	Štuikys et al. (2016)
ParkingAssistant	5	336	6–16	1–32	Botterweck et al. (2010)	SmartHome	2	294	38–61	$9.0{\cdot}10^5$-$3.9{\cdot}10^9$	Santos et al. (2015)
Pick&PlaceUnit	9	699	5–11	3–81	Bürdek et al. (2016)	ERP	2	298	42–57	$2.6{\cdot}10^4$-$2.6{\cdot}10^5$	S.P.L.O.T. (2025)
BCS	3	247	13–17	128–768	Pett et al. (2021)						

RQ3 Which is the test generation policy producing, for FM', test suites with the lowest dissimilarity from those for FM?

RQ4 How does the specificity of a test suite relate to the chosen test generation policy?

RQ5 How does the choice of the generation strategy impact on the mutation score of the produced test suites?

6.1. Experimental methodology

In this section, we describe the experimental methodology we have used for this paper. All experiments have been performed using the models listed in Table 3 and taken from the relevant literature in the FM evolution domain. More specifically, in order to answer the research questions listed above, we have gathered a set of 15 FMs and their evolutions, for a total of 50 models coming from real case studies, as done in Bombarda et al. (2023, 2024). Despite using the same models as our previous works, the results reported in this paper differ due to the changes we have made to the definitions of specificity and dissimilarity. More precisely, the new definitions are stricter, resulting in generally higher dissimilarity compared to Bombarda et al. (2023) and lower specificity compared to Bombarda et al. (2024). Additionally, for this work, we have included models obtained by applying the mutations presented in Table 2 to the industrial FMs: Starting with any of the models in Table 3, we artificially generated a new set of FMs. Each FM in this set is produced from an initial model with one of the mutations applied. The aim of including artificially generated FM evolutions is twofold: First, we increase the experimental dataset size; Second, we are able to analyze the performance of each test generation policy when a limited and known update is applied to the FM. However, some mutations can produce a high number of FMs. For example, starting from a single FM, the `LogicOrToAnd` will return a set of N FMs, and in each of them, a single OR will be changed into an AND in the constraints. To keep under control the number of FMs used in our analysis we limit up to 50 the number of FMs generated by applying a specific mutation to a single FM. Table 3 reports the number of versions (V), the number of mutants we generated for each FM (M), the minimum and maximum number of features including core and dead ones) across all the evolutions (#F), the minimum and the maximum number of products (#P), and the reference to the paper where the models come from. For the models having more than a single evolution step, we considered 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 this way, in total, we considered 35 evolutions.

For each FM, we generate a test suite using all the policies discussed in Section 5 and we record the test suite generation time and size, the dissimilarity, the specificity, and the mutation score. For GFE, which requires a test suite for the original FM to be given as a starting point, we generated it by using ACTS (Yu et al., 2013).

All the experiments have been repeated 10 times to reduce the influence of non-deterministic timing, on an Apple MacBook Pro using an M3 Max CPU, with 14 cores and 36 GB RAM. In such a way, with all FM evolution analyzed, we performed 32,558 test generations for each policy. Code, experiments, and experiment results are available online (Bombarda et al., 2025).

We compare the results obtained by each test generation policy and for each measure, across all FM evolutions, by using the Wilcoxon[2] Sum Rank test (Woolson, 2008), a general test comparing the distributions in paired samples that does not require data to be normally distributed. Given x the measure to be compared between the two techniques, the Wilcoxon Sum 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, and two-sided alternative hypotheses $H_1{}^+$ and $H_1{}^-$.

6.2. Experimental results

In the following, we discuss the results measured after our experiments, guided by the previously defined five research questions. Note that in the tables we report the average values, which differ from the median values reported with a line in each box in the figures.

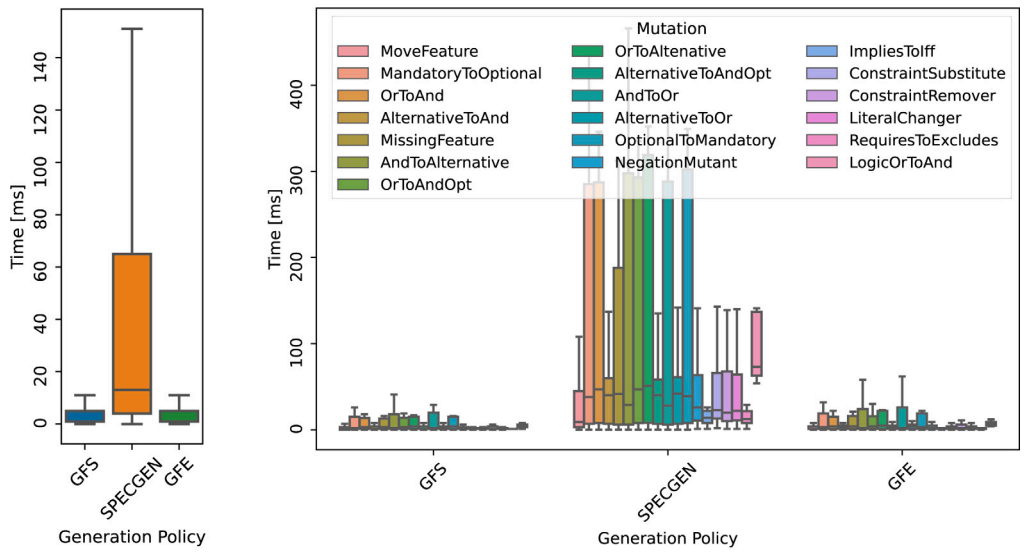
RQ1. Generation time. As presented in Section 4, one of the most common metrics for evaluating test generation strategies is the test suite generation time. Thus, in this research question, we evaluate the three analyzed policies (GFS, GFE, and SPECGEN) by comparing the time t required for generating a test suite.

A box plot with the test generation times is shown in Fig. 5(a). The plot highlights that SPECGEN is the approach having a higher median value ($med(t_{SPECGEN}) = 13$ ms), while GFE and GFS perform comparably in terms of median value ($med(t_{GFE}) = 1$ ms and $med(t_{GFS}) = 1$ ms). When it comes to the average value, SPECGEN stands out as the one with the highest value, with an average time of 262.3 ms and a standard deviation of 8.5 s. In contrast, GFE and GFS exhibit significantly different performance, with an average time of 231.2 ms for GFE and 144.5 ms for GFS, along with standard deviations of 5.6 s and 0.9 s, respectively. These observations are confirmed by the statistical test. The result is statistically significant between GFS and SPECGEN ($p = 0.0$), GFS and GFE ($p = 4.1 \cdot 10^{-47}$), and between SPECGEN and GFE ($p = 0.0$).

We investigated the reason for the difference between the conclusions we can draw from median and average values. We found that, in many cases, GFE is faster than GFS, and this contributes to the conflicting results. In particular, as experienced in Bombarda et al. (2023), for small models, GFE is faster than GFS (although the difference is negligible, considering the relatively short generation times) while for the biggest models (namely ERP and SmartHome) GFE requires much more time than GFS. This drives the average generation time to a higher value.

To identify different possible patterns with mutations, we report in Fig. 5(b) a box plot analyzing the correlation between times with different test generation policies and mutations. We can observe that the general trend holds: SPECGEN emerges as the approach that yields the highest median time. Certain mutations, like `LogicOrToAnd`, experience a more significant impact when using SPECGEN, while others, such as `ImpliesToIf`, exhibit less pronounced consequences.

[2] We used the Wilcoxon test implemented in the scipy.stats library



(a) Policy time (b) Time for each policy and mutation type

Fig. 5. Test suite generation time.

Table 4
Summary of statistical test results on test suite generation time.

Mutation	Average time [ms]			Non-sig. tests
	GFS	GFE	SPECGEN	
AlToAnd	125.8	166.7	82.9	
AlToAndOpt	144.3	181.9	90.2	
AlToOr	162.7	212.1	89.4	
AndToAl	233.2	343.4	297.4	
AndToOr	332.3	447.6	292.8	
ConstraintRemover	477.9	709.4	225.5	
ConstraintSubst	306.9	414.1	188.4	
ImpliesToIff	2.0	2.9	24.9	GFE vs GFS
LiteralChg	269.9	364.6	162.8	
LogicOrToAnd	1379.9	1837.0	17 671.6	
ManToOpt	168.1	233.5	243.9	
MissingFeature	172.9	240.0	247.4	
MoveFeature	82.5	119.9	133.7	
Negation	479.5	714.5	177.8	
OptToMan	273.2	853.4	1348.1	GFE vs GFS
OrToAl	266.2	438.4	347.3	
OrToAnd	253.6	313.4	340.6	GFE vs GFS
OrToAndOpt	253.0	304.6	346.0	GFE vs GFS
RequiresToExcludes	1.1	1.3	15.1	GFE vs GFS

The summary results of the statistical tests are reported in Table 4 where, for each mutation, we list the average time of each test generation policy and the tests that are non-statistically relevant (i.e., for which it is not possible to claim that a technique performs better than the other). Thanks to the results we obtained with the statistical test, we can claim that, in general, it is always possible to define which is the best-performing approach upon FM evolution, except for the cases in which GFE and GFS behave in a statistically similar way (ImpliesToIff, OptToMan, OrToAnd, OrToAndOpt, and RequiresToExcludes).

All in all, based on the summary of the results we report in Table 4, we can state that SPECGEN should be preferred, in terms of average test suite generation time, when adding new valid configurations (i.e., substituting an *Alternative* relation with an *And* or *Or*, when removing or substituting constraints and/or literals). In all the other cases, GFS and GFE should be preferred and, for complex FMs, the former has better performance than the latter.

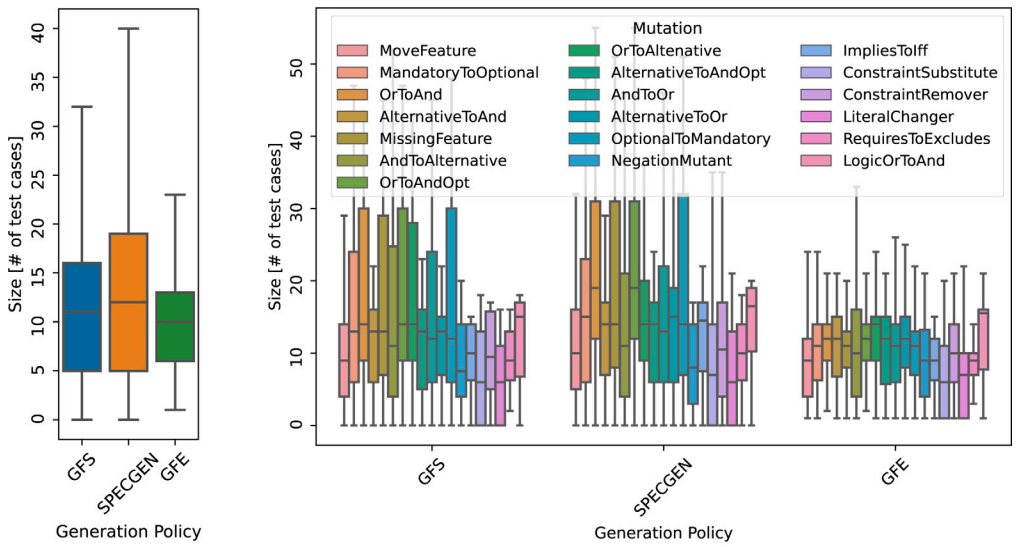
RQ2. Test suite size. In this research question, we evaluate the three analyzed policies (GFS, GFE, and SPECGEN) by comparing the size s of the generated test suites, i.e., the number of test cases. As all three techniques under analysis produce test suites that encompass all pairwise feature interactions, testers goal is to generate the smallest possible test suites. Reducing the size of these test suites helps save time during test execution while maintaining effectiveness, as each pair will still be covered.

A box plot with the test suite size is shown in Fig. 6(a). The plot highlights that SPECGEN is the approach having a higher median value ($med(s_{SPECGEN}) = 12$), followed by GFS ($med(s_{GFS}) = 11$) and GFE ($med(s_{GFE}) = 10$). When it comes to the average value, SPECGEN recorded the highest value, with an average test suite size of 14.3 and a standard deviation of 12.4, followed by GFS with an average test suite size of 12.7 and a standard deviation of 10.2, and by GFE with an average test suite size of 9.4 and a standard deviation of 5.5. These observations are confirmed by the statistical test. The result is statistically significant between GFS and SPECGEN ($p = 1.8 \cdot 10^{-46}$), GFS and GFE ($p = 2.6 \cdot 10^{-111}$), and between SPECGEN and GFE ($p = 0.0$).

In conclusion, we can state that SPECGEN is the tool generating the largest test suites, followed by GFS and GFE. This result highlights that focusing on generating specific test cases can diminish the combinatorial coverage of certain tests (those that are specific). Consequently, additional test cases are required to obtain the desired coverage. Furthermore, reusing an optimized test suite as a starting point (such as the one generated by ACTS) significantly influences the outcome of the test generation process.

As done in RQ1, to identify different possible patterns with mutations, we report in Fig. 6(b) a box plot analyzing the correlation between test suite size with different test generation policies and mutations. We can observe that the same trend holds considering all models as a whole: SPECGEN is the policy generating the highest number of test cases. Similar conclusions can be drawn by looking at the average values and statistical test results reported in Table 5. For all mutations, GFE is always the policy leading to the smallest test suite, followed by GFS and SPECGEN. In most cases (10 out of 19), GFE and GFS perform statistically equally. For eight mutations, SPECGEN generates a test suite with a comparable size to GFS. The only case in which SPECGEN is not the approach generating the largest test suite is the OrToAl mutation, but in this case, the test is not statistically significant.

All in all, we can state that GFE or GFS should be preferred, in terms of test suite size in all cases, while SPECGEN is always the worst test generation policy or behaves comparably to the others we analyzed.



(a) Policy size

(b) Size for each policy and mutation type

Fig. 6. Test suite size.

Table 5
Summary of statistical test results on test suite size.

Mutation	Average size			Non-sig. tests
	GFS	GFE	SPECGEN	
AlToAnd	11.7	11.1	13.8	GFS vs GFE
AlToAndOpt	11.5	10.8	13.7	GFS vs GFE
AlToOr	11.9	11.1	14.8	GFS vs GFE
AndToAl	14.0	11.6	14.8	GFS vs SPECGEN
AndToOr	15.1	11.1	16.3	
ConstraintRemover	10.8	9.6	11.4	All
ConstraintSubst	7.6	7.0	8.4	All
ImpliesToIff	9.7	9.2	12.6	GFS vs GFE
LiteralChg	7.3	6.7	8.6	All
LogicOrToAnd	15.5	12.3	17.6	All
ManToOpt	15.2	10.6	16.8	
MissingFeature	16.4	10.1	18.3	
MoveFeature	10.9	8.5	12.5	
Negation	9.6	8.7	9.6	All
OptToMan	16.6	9.8	19.6	
OrToAl	16.9	12.2	16.3	GFS vs SPECGEN
OrToAnd	17.9	11.1	21.1	
OrToAndOpt	17.7	11.0	21.1	
RequiresToExcludes	9.1	8.6	10.0	All

RQ3. Test suite dissimilarity. In this research question, we evaluate the three analyzed policies (GFS, GFE, and SPECGEN) by comparing the dissimilarity d between the test suites generated after FM evolution and those generated for the original FM from which the evolution began.

A box plot with the test suite dissimilarity is shown in Fig. 7(a). The plot highlights that SPECGEN is the approach having a higher median value ($med(d_{SPECGEN}) = 24.3\%$), followed by GFS ($med(d_{GFS}) = 22.0\%$) and GFE ($med(d_{GFE}) = 8.0\%$).

The same ranking can be observed when considering the average dissimilarity value for each test generation policy. SPECGEN recorded the highest value, with an average dissimilarity of 32.0% and a standard deviation of 25.7%, followed by GFS with an average dissimilarity of 30.0% and a standard deviation of 27.7%, and by GFE with an average dissimilarity of 17.7% and a standard deviation of 25.7%. These observations are confirmed by the statistical test. The result is statistically significant between GFS and SPECGEN ($p = 1.1 \cdot 10^{-47}$), GFS and GFE ($p = 0.0$), and between SPECGEN and GFE ($p = 0.0$).

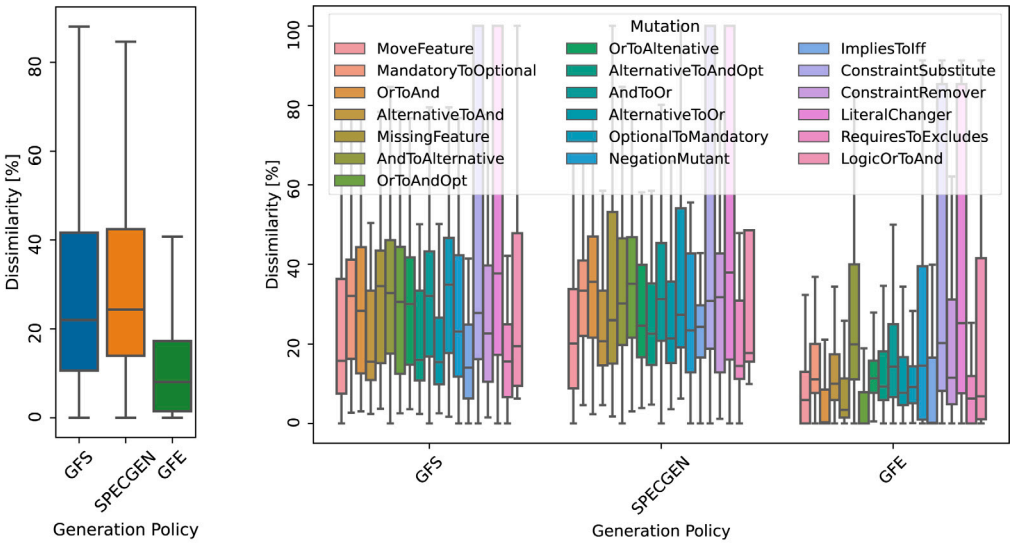
In conclusion, we can state that GFE is the approach leading to test suites with lower dissimilarity, followed by GFS and SPECGEN. This is an expected outcome, as GFE starts its generation process from an already-existing test suite (i.e., the one of the original model). Thus, only minor changes, deletions, or additions are performed and the dissimilarity is kept under control. On the other side, SPECGEN focuses on covering "new" products and this contributes to increasing the dissimilarity.

We have analyzed the impact of choosing one specific test generation strategy when models undergo specific evolutions to identify different possible patterns with mutations. We report in Fig. 7(b) a box plot analyzing the test suite dissimilarity with different test generation policies and mutations. The results we obtained confirm the trend observed when considering all test suites as a whole: GFE is, for all mutations, the approach leading to the lowest dissimilarity, while SPECGEN and GFS produce more dissimilar test suites. The impact of using GFE is, however, not the same for all mutations, and the variability of results strongly varies. While for most mutations the variability decreases, for the `LiteralChg`, `AndToAl`, `LogicOrToAnd`, and `ConstraintSubst` mutations the dissimilarity decreases but the variability remains at a high level, with `AndToAl` having a higher variability with GFE than with SPECGEN. However, despite this variability, the statistical test results reported in Table 6 confirm that GFE is the policy with the lower dissimilarity. With most of the mutations (10 out of 19), GFS obtained the same statistical results as SPECGEN, while for the others GFS led to a lower dissimilarity.

All in all, we can state that when the lowest dissimilarity is pursued, GFE should be always preferred, while SPECGEN and GFE are comparable.

RQ4. Test suite specificity. In this research question, we evaluate the three analyzed policies (GFS, GFE, and SPECGEN) by comparing the specificity *spec* of the test suites they generate (see Section 4.4).

A box plot with the value of the average specificity recorded when generating test suites with different policies is shown in Fig. 8(a). The plot highlights that SPECGEN is the approach having the highest median value ($med(spec_{SPECGEN}) = 33.3\%$), followed by GFE ($med(spec_{GFE}) = 16.7\%$) and GFS ($med(spec_{GFS}) = 10.0\%$). The same ranking can be observed when considering the average specificity value for each test generation policy. SPECGEN recorded the highest value, with an average specificity of 38.2% and a standard deviation of 37.3%, followed by GFE with an average specificity of 26.8% and a standard deviation of 30.0%, and by GFS with an average specificity of 26.1% and a standard deviation of 31.6%. These observations are confirmed



(a) Policy dissimilarity (b) Dissimilarity for each policy and mutation type

Fig. 7. Test suite Dissimilarity.

Table 6
Summary of statistical test results on test suite dissimilarity.

Mutation	Average dissimilarity [%]			Non-sig. tests
	GFS	GFE	SPECGEN	
AlToAnd	28.1	20.6	32.7	
AlToAndOpt	29.6	21.8	33.8	
AlToOr	26.1	18.0	31.8	
AndToAl	36.4	28.9	36.3	GFS vs SPECGEN
AndToOr	33.8	21.2	35.7	
ConstraintRemover	34.2	26.5	38.6	GFS vs SPECGEN
ConstraintSubst	47.4	38.4	48.1	GFS vs SPECGEN
ImpliesToIff	18.2	11.3	25.4	
LiteralChg	49.9	41.7	50.1	GFS vs SPECGEN
LogicOrToAnd	38.4	27.8	41.0	GFS vs SPECGEN
ManToOpt	33.3	18.6	36.5	
MissingFeature	32.8	12.2	33.1	GFS vs SPECGEN
MoveFeature	26.0	14.9	28.0	
Negation	36.3	28.0	37.7	GFS vs SPECGEN
OptToMan	37.4	18.6	38.3	GFS vs SPECGEN
OrToAl	32.4	17.9	31.2	GFS vs SPECGEN
OrToAnd	32.1	9.9	38.3	
OrToAndOpt	32.6	10.4	37.7	
RequiresToExcludes	17.1	10.6	22.1	GFS vs SPECGEN

Table 7
Summary of statistical test results on the specificity of test suites.

Mutation	Average specificity [%]			Non-sig. tests
	GFS	GFE	SPECGEN	
AlToAnd	54.2	31.5	66.6	
AlToAndOpt	53.3	31.4	65.8	
AlToOr	36.1	28.6	57.7	
AndToAl	50.9	53.3	56.6	GFE vs GFS
AndToOr	37.4	31.2	55.1	
ConstraintRemover	27.6	32.9	38.2	All
ConstraintSubst	21.3	36.7	23.3	GFS vs SPECGEN
ImpliesToIff	13.6	12.7	15.2	All
LiteralChg	21.8	37.6	23.2	GFS vs SPECGEN
LogicOrToAnd	17.6	28.9	18.1	All
ManToOpt	34.7	26.3	61.9	
MissingFeature	30.2	30.6	31.9	All
MoveFeature	19.9	23.5	31.7	
Negation	25.8	34.4	24.4	GFS vs SPECGEN
OptToMan	13.9	19.0	16.3	GFS vs SPECGEN
OrToAl	13.9	16.8	14.3	All
OrToAnd	26.9	19.3	59.7	
OrToAndOpt	25.3	18.5	60.2	
RequiresToExcludes	18.4	15.2	16.2	All

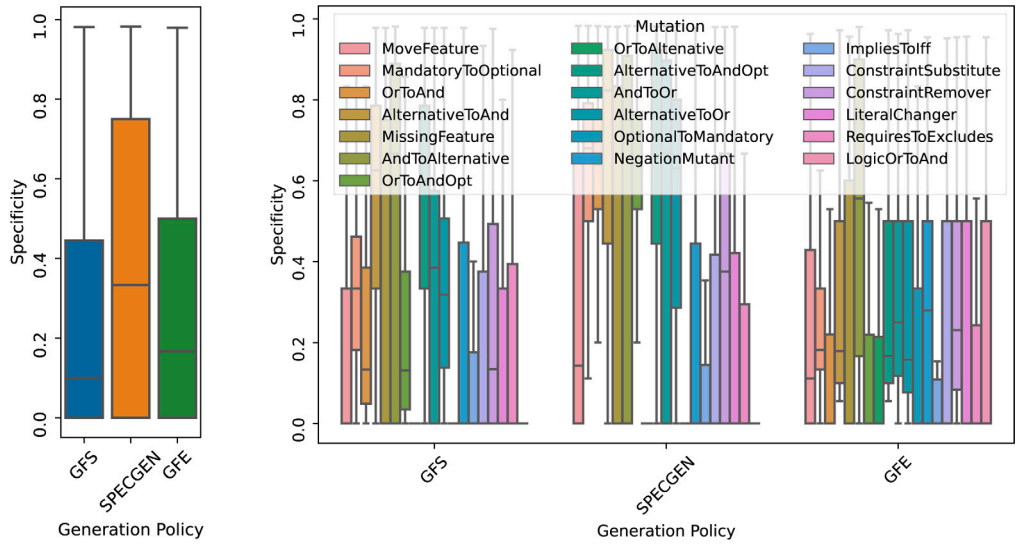
by the statistical test. The result is statistically significant between GFS and SPECGEN ($p = 2.9 \cdot 10^{-217}$), GFS and GFE ($p = 2.4 \cdot 10^{-24}$), and between SPECGEN and GFE ($p = 3.9 \cdot 10^{-150}$).

In conclusion, we can state that SPECGEN is the approach leading to test suites with higher specificity, followed by GFE and GFS. This is an expected outcome, as SPECGEN is an approach specifically designed to produce test suites maximizing the number of specific test cases.

We have analyzed the impact of choosing one specific test generation strategy when models undergo different evolutions to identify different possible patterns with mutations. We report in Fig. 8(b) a box plot showing the test suite specificity with different test generation policies and mutations. The results we obtained confirm the trend observed when considering all test suites as a whole: SPECGEN is, for most mutations, the approach leading to the highest specificity, while GFE and GFS produce less specific test suites. For some mutations (`OrToAl`, `OptToMan`, `LogicOrToAnd`), however, SPECGEN and GFS are not able to generate highly specific test suites, while GFE succeeds. Similarly, for the `ConstraintSubst`, `LiteralChg`, and `Negation`, SPECGEN can generate specific test suites, but with lower average specificity than those generated by GFE. We can conjecture that this is due to the repairing procedure implemented by GFE. It discards single assignments to repair test cases for the evolved models and, by doing this, it may generate test cases that are not optimized but are more specific. We have performed a set of statistical tests on each policy/mutation combination and Table 7 reports the outcome. These tests confirm our previous observations: For more than half of the mutations (10 out of 19), SPECGEN outperformed all other policies; For five mutations the three approaches performed in a statistically similar way.

All in all, we can state that when the highest specificity is pursued, SPECGEN should be preferred, except in the case in which mutations such as `OrToAl`, `OptToMan`, `LogicOrToAnd`, `ConstraintSubst`, `LiteralChg`, and `Negation` are applied. In these cases, using GFE can be advantageous.

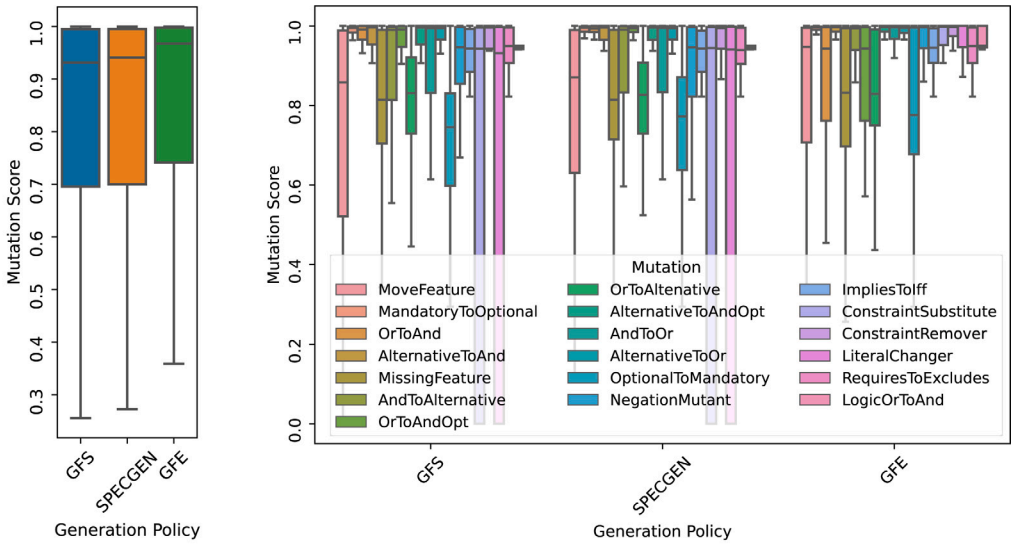
RQ5. Fault detection analysis. In this research question, we evaluate the three analyzed policies (GFS, GFE, and SPECGEN) by comparing the mutation score m of the generated tests (see Section 4.5) and, thus, their fault detection capability.



(a) Policy specificity

(b) Specificity for each policy and mutation type

Fig. 8. Test suite specificity.



(a) Policy mutation score

(b) Mutation score for each policy and mutation type

Fig. 9. Test suite mutation score.

A box plot with the mutation scores is shown in Fig. 9(a). The plot highlights that GFE is the approach having the highest median value ($med(m_{GFE}) = 96.7\%$), followed by SPECGEN ($med(m_{SPECGEN}) = 94.1\%$) and by GFS ($med(m_{GFS}) = 93.2\%$). The same ranking can be observed when considering the average mutation score for each test generation policy. GFE recorded the highest value, with an average mutation score of 84.8% and a standard deviation of 21.6%, followed by SPECGEN with an average mutation score of 77.7% and a standard deviation of 31.0%, and by GFS with an average mutation score of 76.4% and a standard deviation of 31.9%. These observations are confirmed by the statistical test. The result is statistically significant between GFS and SPECGEN ($p = 5.5 \cdot 10^{-8}$), GFS and GFE ($p = 6.9 \cdot 10^{-210}$), and between SPECGEN and GFE ($p = 3.7 \cdot 10^{-143}$). In conclusion, we can state that GFE is the approach leading to test suites with highest mutation score, followed by SPECGEN and GFS.

As done with the previous RQs, we have analyzed the impact of choosing one specific test generation strategy when models undergo different evolutions to identify different possible patterns with mutations. We report in Fig. 9(b) a box plot analyzing the correlation between mutation scores with different test generation policies and mutations. The same considerations we drew when considering all test suites as a whole still hold: GFE is the policy with the highest mutation score, followed by SPECGEN and GFS which perform comparably. The only mutations for which GFE is outperformed by the other policies are `OrToAnd` and `OrToAndOpt`, as shown by the average values and statistical tests results reported in Table 8. For most mutations (15 out of 19) GFS and SPECGEN perform in a comparable way, while only for two mutations (`ImpliesToIif` and `RequiresToExcludes`) there is no real advantage in using one specific policy.

All in all, we can conclude that for almost all mutations GFE is the best test generation policy when it comes to mutation score. The

Table 8
Summary of statistical test results on the mutation score of test suites.

Mutation	Average mutation score [%]			Non-sig. tests
	GFS	GFE	SPECGEN	
AlToAnd	86.9	98.3	87.0	GFS vs SPECGEN
AlToAndOpt	85.3	98.4	85.9	GFS vs SPECGEN
AlToOr	88.0	98.0	87.7	GFS vs SPECGEN
AndToAl	82.5	91.7	83.8	GFS vs SPECGEN
AndToOr	85.0	92.2	85.8	GFS vs SPECGEN
ConstraintRemover	80.1	97.5	79.2	GFS vs SPECGEN
ConstraintSubst	63.8	97.3	64.9	GFS vs SPECGEN
ImpliesToIff	90.9	92.9	91.6	All
LiteralChg	63.1	96.7	63.8	GFS vs SPECGEN
LogicOrToAnd	73.4	97.0	73.5	GFS vs SPECGEN
ManToOpt	91.8	98.9	92.6	GFS vs SPECGEN
MissingFeature	77.8	83.0	78.6	GFS vs SPECGEN
MoveFeature	72.0	81.5	73.5	
Negation	76.2	96.2	74.5	GFS vs SPECGEN
OptToMan	67.2	77.2	70.0	
OrToAl	77.5	84.1	77.7	GFS vs SPECGEN
OrToAnd	90.7	88.0	92.5	
OrToAndOpt	89.6	88.1	93.3	
RequiresToExcludes	93.9	93.9	91.2	All

Table 9
Best test generation policy for target.

Time ↓	Size ↓	Dissimilarity ↓	Specificity ↑	Fault detection ↑
GFS	GFE	GFE	SPECGEN	GFE

only mutations for which SPECGEN is preferable are `OrToAnd` and `OrToAndOpt`.

6.3. Discussion

In this section, we provide a summary discussion of the results we obtained with our RQs. This discussion aims to provide *guidelines* allowing testers to decide what to do with their test cases upon FM evolution, depending on the specific target and mutations.

General comparison. Table 9 reports a summary of the best test generation policy depending on the target, in a general view. As previously discussed, GFE emerges as the best test generation policy for what concerns size, dissimilarity, and fault detection capability. SPECGEN is a policy suggested only when the maximum specificity is pursued, i.e., when testers want to focus mainly on the new valid configurations. Finally, GFS emerges as the best test generation policy if test suites have to be generated in the shortest possible time. However, if the FMs under test are complex, using GFE can be a more performing alternative.

Mutation-based comparison. Table 10, presents the ranking of test generation policies for each mutation, depending on the target. These results can be considered as a set of guidelines, to aid testers in choosing what to do with their test cases after FM evolution. In the following, we present a non-exhaustive list of typical scenarios and discuss the appropriate testing approaches for each of them:

- **Standardizing a feature across all products:** In this scenario, during the evolution, a previously optional feature becomes mandatory. This can occur due to system improvements or because of new regulations, such as the mandatory installation of the Anti-lock Braking System in cars in Europe. In such cases, the evolution is likely to be implemented using the `OptToMan` mutation. For all targets (including time, size, dissimilarity, specificity, and fault detection), the GFE strategy should be adopted, as it consistently yields the best results.
- **Deprecating a feature:** In this scenario, during the evolution, a feature becomes deprecated and is removed from the products. This may happen because of technological advancements, such as in a desktop publishing application that removes its "'Flash Export" feature, as the technology is obsolete and no longer supported by modern web browsers. In such cases, the evolution is likely to be implemented using the `MissingFeature` mutation. For all targets, except for test generation time, generating test cases from an existing test suite (GFE) is the optimal choice. However, if the test generation must be completed as quickly as possible, using GFS is a better option.
- **Refactoring an FM:** In this scenario, the FM undergoes a refactoring process to improve its clarity. This can occur because during the FM evolution, in several stages, new independent features have been added but they could be grouped under a parent feature. In such cases, no new features are added, but the evolution is likely to be implemented using the `MoveFeature` mutation. Considering that most of the FM will remain unchanged, it is generally recommended to generate test cases from the existing test suite (GFE).
- **Removing or refactoring a constraint to increase options:** During the evolution of a system, the set of possible products can be expanded by removing or refactoring a cross-tree constraint. This might occur due to technological advancements or market repositioning. For instance, let us consider a company offering LLMs functionalities. Initially, only pro users could access the most advanced models. However, to gain a competitive edge, this constraint might be removed, allowing free-tier users to try the new models with potential usage limitations. In such cases, the evolution is likely to be implemented using the `ConstraintRemover`, `ConstraintSubst`, or `ManToOpt` mutations. Since testers may be interested in assessing the correct functioning of the newly added products, focusing on specific tests and using the SPECGEN approach is recommended. However, if testers still want to test existing products, GFE enables enhanced fault detection and reduced dissimilarity.
- **Introducing new major system capabilities:** During the evolution, the set of possible products can be expanded by adding new capabilities. Let us consider the scenario in which a car manufacturer's configuration system previously included a standard *Radio* with all trims. This feature could be evolved into a *Media System* group, from which customers can now choose a *Standard Radio*, a *Touchscreen Display*, or a *Premium Sound System*. In such cases, the evolution is likely to be implemented by using mutations expanding the set of possible products by only focusing on FM structure, such as `AlToAnd`, `AlToOr`, or `AlToAndOpt`. Commonly, testers are interested in assessing the correct functioning of the newly added configurations. Thus, they may opt to generate specific tests and use the SPECGEN approach. However, if testers still want to test existing products, GFE or GFS enable reduced test suite size and dissimilarity and enhanced fault detection capability.

As a rule of thumb, however, we can see that GFE prevails as the best technique for most mutations and targets.

Dissimilarity vs specificity. In principle, the objective of minimizing dissimilarity can conflict with the objective of increasing the specificity. Indeed, while minimizing the dissimilarity requires keeping a test suite that is the most similar as possible to the one previously generated for the initial version of a FM, increasing the specificity requires a test suite focusing on "new" configurations. The results presented in Tables 9 and 10 support this observation. GFE is the approach that enables the generation of test suites with reduced dissimilarity, while SPECGEN, which concentrates on specific tests, is the approach that generates test suites with enhanced specificity. The choice of test generation approach depends on the specific case. If testers prioritize testing new products, achieving the highest specificity is the ideal choice. Conversely, if regression testing is crucial, reducing dissimilarity is typically the preferred strategy.

Size/Dissimilarity vs fault detection. Testers always strive for maximum fault detection capability when testing FMs. Notably, the

Table 10
Test generation policy ranking for target and mutation (GE stands for GFE, GS is used for GFS, and SP stands for SPECGEN).

Mutation	Time↓			Size↓			Diss.↓			Spec.↑			Fault det.↑		
	1st	2nd	3rd	1st	2nd	3rd	1st	2nd	3rd	1st	2nd	3rd	1st	2nd	3rd
AlToAnd	SP	GS	GE	GE/GS		SP	GE	GS	SP	SP	GS	GE	GE	SP/GS	
AlToAndOpt	SP	GS	GE	GE/GS		SP	GE	GS	SP	SP	GS	GE	GE	SP/GS	
AlToOr	SP	GS	GE	GE/GS		SP	GE	GS	SP	SP	GS	GE	GE	SP/GS	
AndToAl	GS	SP	GE	GE	SP/GS		GE	SP/GS		SP	GE/GS		GE	SP/GS	
AndToOr	SP	GS	GE	GE	GS	SP	GE	GS	SP	SP	GS	GE	GE	SP/GS	
ConstraintRemover	SP	GS	GE	*Any*			GE	SP/GS		*Any*			GE	SP/GS	
ConstraintSubst	SP	GS	GE	*Any*			GE	SP/GS		GE	SP/GS		GE	SP/GS	
ImpliesToIff	GE/GS		SP	GE/GS		SP	GE	GS	SP	*Any*			*Any*		
LiteralChg	SP	GS	GE	*Any*			GE	SP/GS		GE	SP/GS		GE	SP/GS	
LogicOrToAnd	GS	GE	SP	*Any*			GE	SP/GS		*Any*			GE	SP/GS	
ManToOpt	GS	GE	SP	GE	GS	SP	GE	GS	SP	SP	GE/GS		GE	SP/GS	
MissingFeature	GS	GE	SP	GE	GS	SP	GE	SP/GS		*Any*			GE	SP/GS	
MoveFeature	GS	GE	SP	GE	GS	SP	GE	GS	SP	SP	GE	GS	GE	SP	GS
Negation	SP	GS	GE	*Any*			GE	SP/GS		GE	SP/GS		GE	SP/GS	
OptToMan	GE/GS		SP	GE	GS	SP	GE	SP/GS		GE	SP/GS		GE	SP	GS
OrToAl	GS	SP	GE	GE	SP/GS		GE	SP/GS		*Any*			GE	SP/GS	
OrToAnd	GE/GS		SP	GE	GS	SP	GE	GS	SP	SP	GS	GE	SP	GS	GE
OrToAndOpt	GE/GS		SP	GE	GS	SP	GE	GS	SP	SP	GS	GE	SP	GE	GS
RequiresToExcludes	GE/GS		SP	*Any*			GE	SP/GS		*Any*			*Any*		

data in Tables 9 and 10 reveal that GFE, which generates test cases based on an existing test suite, achieves the smallest test suite size and lowest dissimilarity while also delivering the highest fault detection. The only cases in which using SPECGEN achieves higher fault detection than GFE is with `OrToAnd` and `OrToAndOpt`. This is, in general, a noteworthy result, as it means that we can discover faults even when using smaller test suites.

7. Threats to validity

In this section, we discuss the threats to validity (Feldt and Magazinius, 2010) and all the strategies we have undertaken to mitigate them.

Internal validity refers to the fact that the different outcomes obtained with the analyzed approaches 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 implementing the experiments to see if there could be other factors that have caused the outcome, such as errors in the tools we exploited. A possible threat to the *construct validity* comes from the assumption that the metrics we selected for comparing test generation approaches are suitable to measure the "quality" of a test suite or of a test generation policy for testing evolving FMs. To mitigate this risk, we have carefully checked the literature to find similar approaches. For example, regarding the test suite specificity, a similar metric was proposed by Arcaini et al. (2015), and it was used to identify distinguishing configurations, i.e., configurations that were valid in the old FM but not in the evolved one, or vice versa. Regarding test suite size and generation time, as well as mutation score, these are established measures in the field of test generation. Finally, regarding test suite dissimilarity, we rely on the literature for its definition, where similar distances are often used, as in Devroey et al. (2016). We acknowledge that there may be instances where our distance metric does not accurately reflect the actual effort required to modify an existing test suite. For example, we consider modifying a test to be significantly less costly than writing a new test. However, there are cases where a modified test or a new one may require the same level of effort, so any change to one test has the same weight.

Lastly, *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 FMs we have used in the experiments discussed in Section 6. 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). By employing synthetic models generated through mutations applied to the models we found in the literature, we believe that the generalizability of our findings is enhanced. Nevertheless, more complex models may exist in a real scenario and there may be scalability issues for which further experiments are needed, especially when using approaches employing DDs for test generation (Benavides et al., 2010; Bombarda and Gargantini, 2024b).

8. Related works

The evolution of FMs is a topic of great interest in software engineering, particularly in the areas of testing and requirements traceability. This problem is particularly important when software requirements are subject to frequent changes, such as during agile processes (Mendonça et al., 2024). However, to the best of our knowledge, this is the first work comparing such a comprehensive set of policies (GFE, GFS, and SPECGEN) using a set of five metrics (time, size, dissimilarity, specificity, and fault detection capability).

Testing FMs is not only conceptually important to ensure the proper functioning of the products but can be used to identify anomalies that could arise during FM evolution (Nieke et al., 2018; Neves et al., 2011). In this paper, we simulated FM evolution through artificially applied mutations. This approach is commonly applied in the literature (Bürdek et al., 2015). In Svahnberg and Bosch (1999), Thüm et al. (2009), the authors compared the FMs designed during software evolution and found out that some types of evolution actions are more common than others, but they also highlighted that arbitrary edits are commonly performed during the SPL life cycles. Thus, using artificial mutations can be a way to simulate real evolutions.

Practitioners normally address the problem of testing evolving FMs in two different directions: by trying to preserve a test suite as much as possible during the evolution process (as done by the GFE technique Bombarda et al., 2023), or by focusing on testing new products

that were not previously valid (as done by SPECGEN Bombarda et al., 2024). The idea underlying approaches favoring the reuse of test cases is that, in this way, lower effort should be employed to write new test cases. Indeed, reusing test cases is recommended by many. In Makady and Walker (2018), 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. Despite (Makady and Walker, 2018) was not referring to FMs, in this work we have reached similar conclusions.

9. Conclusion

In this paper, we tackled the challenge of maintaining and evolving test suites for Software Product Lines (SPLs) when their Feature Models (FMs) change. In this process, we evaluated three test generation policies, namely GFS, GFE, and SPECGEN, against the most used metrics (test suite generation time, size, and mutation score). In addition, we considered two metrics, i.e., dissimilarity and specificity, which we introduced in previous work and we have enhanced and adapted them to be consistent.

Our experiments, conducted on 13 industrial FM families with 35 natural evolutions and over 3200 artificially mutated FMs, revealed several key insights. GFE generally outperformed the other policies in terms of producing smaller test suites with lower dissimilarity and higher mutation scores, making it the most efficient for maintaining test suites while ensuring high fault detection. SPECGEN excelled in generating specific tests, especially for evolutions that broadened the set of valid configurations, but required the longest generation times, in particular in cases in which the FM is complex, and produced larger test suites. Finally, GFS, while trivial, was found to be effective for scenarios where FMs are simple and pre-processing their test suites requires more time than the actual test generation, but it lagged behind GFE and SPECGEN in most metrics. With our experiments, we also observed that tailoring test generation policies to specific scenarios is important, as the same technique performs in different ways for different FM evolutions. For example, SPECGEN was particularly advantageous for evolutions introducing new valid configurations, while GFE was preferable for incremental updates that maintained much of the original FM structure and introduced only minor changes.

As future work, we plan to include in the experiments models obtained with higher-order mutations and to explore higher strengths beyond pairwise testing, to see whether our findings can be generalized to more complex contexts.

CRediT authorship contribution statement

Andrea Bombarda: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Formal analysis, Data curation, Conceptualization. **Silvia Bonfanti:** Writing – review & editing, Writing – original draft, Validation, Software, Project administration, Methodology, Formal analysis, Conceptualization. **Angelo Gargantini:** Writing – review & editing, Writing – original draft, Validation, Supervision, Software, Project administration, Methodology, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been partially by project SAFEST: Trust assurance of Digital Twins for medical cyber–physical systems, funded by the European Union-Next Generation EU, Mission 4, Component 2, Investment 1.1 Fondo per il Programma Nazionale di Ricerca e Progetti di Rilevante Interesse Nazionale (PRIN), code 20224AJBLJ-CUP F53D23004230006 under the National Recovery and Resilience Plan (NRRP), and in part by AdvaNced Technologies for Human-centrEd Medicine (ANTHEM)-Grant PNC0000003–CUP B53C22006700001-Spoke 1-Pilot 1.4. funded by the Italian Ministry of Universities and Research under the Complementary National Plan (PNC), in part by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

Data availability

All the code, models, test suites and the replication material for the experiments presented in this paper are available here: https://github.com/fmselab/ctwedge/tree/master/ctwedge.parent/ctwedge.fmtester.

References

Akers, 1978. Binary decision diagrams. IEEE Trans. Comput. C-27 (6), 509–516. http://dx.doi.org/10.1109/TC.1978.1675141.

Ali, N., Hoing, J.-E., 2019. Your opinions let us know: Mining social network sites to evolve software product lines. KSII Trans. Internet Inf. Syst. 13, 21. http://dx.doi.org/10.3837/tiis.2019.08.021.

Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C., 2006. Refactoring product lines. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering. GPCE '06, Association for Computing Machinery, New York, NY, USA, pp. 201–210. http://dx.doi.org/10.1145/1173706.1173737.

Anon, 2022b. Pure::variants user's guide. pp. 5–11, pure-systems GmbH. URL https://www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf.

Anon, 2023. Jenny website. http://burtleburtle.net/bob/math/jenny.html.

Arcaini, P., Gargantini, A., Radavelli, M., 2019. Achieving change requirements of feature models by an evolutionary approach. J. Syst. Softw. 150, 64–76. http://dx.doi.org/10.1016/j.jss.2019.01.045.

Arcaini, P., Gargantini, A., Vavassori, P., 2015. Generating tests for detecting faults in feature models. In: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation. ICST, pp. 1–10. http://dx.doi.org/10.1109/ICST.2015.7102591.

Arcega, L., Font, J., Haugen, Ø., Cetina, C., 2016. Achieving knowledge evolution in dynamic software product lines. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering. SANER, 1, pp. 505–516. http://dx.doi.org/10.1109/SANER.2016.24.

Benavides, D., Segura, S., Ruiz-Cortés, A., 2010. Automated analysis of feature models 20 years later: A literature review. Inf. Syst. 35 (6), 615–636. http://dx.doi.org/10.1016/j.is.2010.01.001.

Bombarda, A., Bonfanti, S., Gargantini, A., 2023. On the reuse of existing configurations for testing evolving feature models. SPLC '23, In: Proceedings of the 27th ACM International Systems and Software Product Line Conference, vol. B, Association for Computing Machinery, New York, NY, USA, pp. 67–76. http://dx.doi.org/10.1145/3579028.3609017.

Bombarda, A., Bonfanti, S., Gargantini, A., 2024. Testing the evolution of feature models with specific combinatorial tests. In: 2024 IEEE International Conference on Software Testing, Verification and Validation Workshops. ICSTW, IEEE, pp. 1–10. http://dx.doi.org/10.1109/icstw60967.2024.00025.

Bombarda, A., Bonfanti, S., Gargantini, A., 2025. Replication package. URL https://github.com/fmselab/ctwedge/tree/master/ctwedge.parent/ctwedge.fmtester. Online. (Accessed 24 January 2025).

Bombarda, A., Crippa, E., Gargantini, A., 2021. An environment for benchmarking combinatorial test suite generators. In: 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops. ICSTW, pp. 48–56. http://dx.doi.org/10.1109/ICSTW52544.2021.00021.

Bombarda, A., Gargantini, A., 2020. 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, pp. 157–166. http://dx.doi.org/10.1109/ICSTW50294.2020.00036.

Bombarda, A., Gargantini, A., 2022. Parallel test generation for combinatorial models based on multivalued decision diagrams. In: 2022 IEEE International Conference on Software Testing, Verification and Validation Workshops. ICSTW, IEEE, pp. 74–81. http://dx.doi.org/10.1109/icstw55395.2022.00027.

Bombarda, A., Gargantini, A., 2023. 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, pp. 197–205. http://dx.doi.org/10.1109/icstw58534.2023.00044.

Bombarda, A., Gargantini, A., 2024a. Integrating product sampling and behavioral testing for software product lines with combinatorial testing. In: 2024 IEEE International Conference on Software Testing, Verification and Validation Workshops. ICSTW, pp. 197–206. http://dx.doi.org/10.1109/ICSTW60967.2024.00046.

Bombarda, A., Gargantini, A., 2024b. On the use of multi-valued decision diagrams to count valid configurations of feature models. In: Proceedings of the 28th ACM International Systems and Software Product Line Conference. SPLC '24, Association for Computing Machinery, New York, NY, USA, pp. 96–106. http://dx.doi.org/10.1145/3646548.3672594.

Botterweck, G., Pleuss, A., 2014. Evolution of software product lines. In: Evolving Software Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 265–295. http://dx.doi.org/10.1007/978-3-642-45398-4_9.

Botterweck, G., Pleuß, A., Dhungana, D., Polzer, A., Kowalewski, S., 2010. EvoFM: feature-driven planning of product-line evolution. In: PLEASE '10. ACM Press, http://dx.doi.org/10.1145/1808937.1808941.

Bürdek, J., Kehrer, T., Lochau, M., Reuling, D., Kelter, U., Schürr, A., 2015. Reasoning about product-line evolution using complex feature model differences. Autom. Softw. Eng. 23 (4), 687–733. http://dx.doi.org/10.1007/s10515-015-0185-3.

Bürdek, J., Kehrer, T., Lochau, M., Reuling, D., Kelter, U., Schürr, A., 2016. Reasoning about product-line evolution using complex feature model differences. Autom. Softw. Eng. 23 (4), 67. http://dx.doi.org/10.1007/s10515-015-0185-3.

Calvagna, A., Gargantini, A., Vavassori, P., 2013. Combinatorial testing for feature models using CitLab. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops. pp. 338–347. http://dx.doi.org/10.1109/ICSTW.2013.45.

Cohen, D., Dalal, S., Fredman, M., Patton, G., 1997. The AETG system: an approach to testing based on combinatorial design. IEEE Trans. Softw. Eng. 23 (7), 437–444. http://dx.doi.org/10.1109/32.605761.

Czerwonka, J., 2022a. PICT GitHub page. https://github.com/microsoft/pict.

Devroey, X., Perrouin, G., Legay, A., Schobbens, P.-Y., Heymans, P., 2016. Search-based similarity-driven behavioural SPL testing. In: Proceedings of the Tenth International Workshop on Variability Modelling of Software-Intensive Systems. VaMoS '16, Association for Computing Machinery, New York, NY, USA, pp. 89–96. http://dx.doi.org/10.1145/2866614.2866627.

Ensan, F., Bagheri, E., Gašević, D., 2012. Evolutionary search-based test generation for software product line feature models. In: Notes on Numerical Fluid Mechanics and Multidisciplinary Design. Springer International Publishing, pp. 613–628. http://dx.doi.org/10.1007/978-3-642-31095-9_40.

Feldt, R., Magazinius, A., 2010. Validity threats in empirical software engineering research - an initial survey. In: SEKE.

Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Castor Filho, F., Dantas, F., 2008. Evolving software product lines with aspects: An empirical study on design stability. In: Proceedings of the 30th International Conference on Software Engineering. ICSE '08, Association for Computing Machinery, New York, NY, USA, pp. 261–270. http://dx.doi.org/10.1145/1368088.1368124.

Gámez, N., Fuentes, L., 2011. Software product line evolution with cardinality-based feature models. In: Schmid, K. (Ed.), Top Productivity Through Software Reuse. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 102–118. http://dx.doi.org/10.1007/978-3-642-21347-2_9.

Gargantini, A., Vavassori, P., 2014. Efficient combinatorial test generation based on multivalued decision diagrams. In: Yahav, E. (Ed.), Hardware and Software: Verification and Testing. Springer International Publishing, Cham, pp. 220–235.

Garvin, B.J., Cohen, M.B., Dwyer, M.B., 2009. An improved meta-heuristic search for constrained interaction testing. In: 2009 1st International Symposium on Search Based Software Engineering. pp. 13–22. http://dx.doi.org/10.1109/SSBSE.2009.25.

Hemmati, H., Briand, L., 2010. An industrial investigation of similarity measures for model-based test case selection. In: 2010 IEEE 21st International Symposium on Software Reliability Engineering. pp. 141–150. http://dx.doi.org/10.1109/ISSRE.2010.9.

Hernández-López, J.-M., Juaréz-Martínez, U., Sergio-David, I.-D., 2018. Automated software generation process with SPL. In: Mejia, J., Muñoz, M., Rocha, A., Quiñonez, Y., Calvo-Manzano, J. (Eds.), Trends and Applications in Software Engineering. Springer International Publishing, Cham, pp. 127–136.

Heß, T., Sundermann, C., Thüm, T., 2021. On the scalability of building binary decision diagrams for current feature models. SPLC '21, In: Proceedings of the 25th ACM International Systems and Software Product Line Conference, volume. A, Association for Computing Machinery, New York, NY, USA, pp. 131–135. http://dx.doi.org/10.1145/3461001.3474452.

Ignaim, K., Fernandes, J.M., Ferreira, A.L., 2024. An industrial experience of using reference architectures for mapping features to code. Sci. Comput. Program. 234, 103087. http://dx.doi.org/10.1016/j.scico.2024.103087, URL https://www.sciencedirect.com/science/article/pii/S0167642324000108.

Johansen, M.F., Haugen, Ø., Fleurey, F., 2011. Properties of realistic feature models make combinatorial testing of product lines feasible. In: Whittle, J., Clark, T., Kühne, T. (Eds.), Model Driven Engineering Languages and Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 638–652.

Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A., 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Kröher, C., Gerling, L., Schmid, K., 2018. Identifying the intensity of variability changes in software product line evolution. SPLC '18, In: Proceedings of the 22nd International Systems and Software Product Line Conference, vol. 1, Association for Computing Machinery, New York, NY, USA, pp. 54–64. http://dx.doi.org/10.1145/3233027.3233032.

Kuhn, D.R., Kacker, R.N., Lei, Y., 2013. Introduction to Combinatorial Testing, first ed. Chapman & Hall/CRC.

Legunsen, O., Hariri, F., Shi, A., Lu, Y., Zhang, L., Marinov, D., 2016. An extensive study of static regression test selection in modern software evolution. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: FSE 2016, Association for Computing Machinery, New York, NY, USA, pp. 583–594. http://dx.doi.org/10.1145/2950290.2950361.

Leithner, M., Bombarda, A., Wagner, M., Gargantini, A., Simos, D.E., 2024. State of the CArt: evaluating covering array generators at scale. Int. J. Softw. Tools Technol. Transf. 26 (3), 301–326. http://dx.doi.org/10.1007/s10009-024-00745-2.

Lotufo, R., She, S., Berger, T., Czarnecki, K., Wasowski, A., 2010. Evolution of the linux kernel variability model. In: Software Product Lines: Going beyond. Springer Berlin Heidelberg, pp. 136–150. http://dx.doi.org/10.1007/978-3-642-15579-6_10.

Makady, S., Walker, R.J., 2018. Debugging and maintaining pragmatically reused test suites. Inf. Softw. Technol. 102, 6–29. http://dx.doi.org/10.1016/j.infsof.2018.05.001.

Marques, M., Simmonds, J., Rossel, P.O., Bastarrica, M.C., 2019. Software product line evolution: A systematic literature review. Inf. Softw. Technol. 105, 190–208. http://dx.doi.org/10.1016/j.infsof.2018.08.014.

Mendonça, W.D., Assunção, W.K., Vergilio, S.R., 2024. Feature-oriented test case selection and prioritization during the evolution of highly-configurable systems. J. Syst. Softw. 217, 112157. http://dx.doi.org/10.1016/j.jss.2024.112157.

Miller, D., Drechsler, R., 2002. On the construction of multiple-valued decision diagrams. In: Proceedings 32nd IEEE International Symposium on Multiple-Valued Logic. pp. 245–253. http://dx.doi.org/10.1109/ISMVL.2002.1011095.

Montalvillo, L., Díaz, O., 2023. In: Lopez-Herrejon, R.E., Martinez, J., Guez Assun¸cao, W.K., Ziadi, T., Acher, M., Vergilio, S. (Eds.), Handbook of Re-Engineering Software Intensive Systems into Software Product Lines. Springer International Publishing, Cham, pp. 495–512. http://dx.doi.org/10.1007/978-3-031-11686-5_20, Ch. Evolution in Software Product Lines: An Overview.

Neves, L., Teixeira, L., Sena, D., Alves, V., Kulesza, U., Borba, P., 2011. Investigating the safe evolution of software product lines. SIGPLAN Not. 47 (3), 33–42. http://dx.doi.org/10.1145/2189751.2047869.

Nieke, M., 2021. Consistent Feature-Model Driven Software Product Line Evolution (Ph.D. thesis). Technische Universität Braunschweig.

Nieke, M., Mauro, J., Seidl, C., Thüm, T., Yu, I.C., Franzke, F., 2018. Anomaly analyses for feature-model evolution. In: Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. In: GPCE 2018, Association for Computing Machinery, New York, NY, USA, pp. 188–201. http://dx.doi.org/10.1145/3278122.3278123.

Oster, S., Markert, F., Ritter, P., 2010. Automated incremental pairwise testing of software product lines. In: Software Product Lines: Going beyond. Springer Berlin Heidelberg, pp. 196–210. http://dx.doi.org/10.1007/978-3-642-15579-6_14.

Passos, L., Czarnecki, K., Apel, S., Wasowski, A., Kästner, C., Guo, J., 2013. Feature-oriented software evolution. In: Proceedings of the 7th International Workshop on Variability Modelling of Software-Intensive Systems. VaMoS '13, Association for Computing Machinery, New York, NY, USA, pp. 1–8. http://dx.doi.org/10.1145/2430502.2430526.

Pett, T., Krieter, S., Runge, T., Thüm, T., Lochau, M., Schaefer, I., 2021. 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, Association for Computing Machinery, New York, NY, USA, pp. 1–9. http://dx.doi.org/10.1145/3442391.3442410.

Pleuss, A., Botterweck, G., Dhungana, D., Polzer, A., Kowalewski, S., 2012. Model-driven support for product line evolution on feature level. J. Syst. Softw. 85 (10), 2261–2274. http://dx.doi.org/10.1016/j.jss.2011.08.008.

Santos, A.R., de Oliveira, R.P., de Almeida, E.S., 2015. 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, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/2745802.2745806.

Seidl, C., Heidenreich, F., Aßmann, U., 2012. Co-evolution of models and feature mapping in software product lines. SPLC '12, In: Proceedings of the 16th International Software Product Line Conference, vol. 1, Association for Computing Machinery, New York, NY, USA, pp. 76–85. http://dx.doi.org/10.1145/2362536.2362550.

S.P.L.O.T., 2025. Repository of real feature models. URL http://52.32.1.180:8080/SPLOT/feature_model_repository_depot.html. Online. (Accessed 24 January 2025).

Štuikys, V., Burbaitė, R., Bespalova, K., Ziberkas, G., 2016. Model-driven processes and tools to design robot-based generative learning objects for computer science education. Sci. Comput. Program. 129, 48–71. http://dx.doi.org/10.1016/j.scico.2016.03.009, Special issue on eLearning Software Architectures.

Svahnberg, M., Bosch, J., 1999. Evolution in software product lines: two cases. J. Softw. Maint.: Res. Pr. 11 (6), 391–422.

Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G., 2014. A classification and survey of analysis strategies for software product lines. ACM Comput. Surv. 47 (1), 1–45. http://dx.doi.org/10.1145/2580950.

Thüm, T., Batory, D., Kästner, C., 2009. Reasoning about edits to feature models. In: 2009 IEEE 31st International Conference on Software Engineering. IEEE, p. 11. http://dx.doi.org/10.1109/ICSE.2009.5070526.

Varshosaz, M., Al-Hajjaji, M., Thüm, T., Runge, T., Mousavi, M.R., Schaefer, I., 2018. A classification of product sampling for software product lines. SPLC '18, In: Proceedings of the 22nd International Systems and Software Product Line Conference, vol. 1, Association for Computing Machinery, New York, NY, USA, pp. 1–13. http://dx.doi.org/10.1145/3233027.3233035.

Wagner, M., Kleine, K., Simos, D.E., Kuhn, R., Kacker, R., 2020. CAGEN: A fast combinatorial test generation tool with support for constraints and higher-index arrays. In: 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops. ICSTW, pp. 191–200. http://dx.doi.org/10.1109/ICSTW50294.2020.00041.

White, J., Galindo, J.A., Saxena, T., Dougherty, B., Benavides, D., Schmidt, D.C., 2014. Evolving feature model configurations in software product lines. J. Syst. Softw. 87, 119–136. http://dx.doi.org/10.1016/j.jss.2013.10.010.

Woodward, M., 1993. Mutation testing—its origin and evolution. Inf. Softw. Technol. 35 (3), 163–169. http://dx.doi.org/10.1016/0950-5849(93)90053-6, URL https://www.sciencedirect.com/science/article/pii/0950584993900536.

Woolson, R.F., 2008. Wilcoxon signed-rank test. http://dx.doi.org/10.1002/9780471462422.eoct979.

Yu, L., Lei, Y., Kacker, R.N., Kuhn, D.R., 2013. ACTS: A combinatorial test generation tool. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation. pp. 370–375. http://dx.doi.org/10.1109/ICST.2013.52.