

# Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-wise Test Suites for Large Software Product Lines

Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, Yves Le Traon

▶ **To cite this version:**

Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, et al.. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-wise Test Suites for Large Software Product Lines. 2012. hal-00756084v2

**HAL Id: hal-00756084**
**https://hal.science/hal-00756084v2**

Submitted on 23 Nov 2012

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-wise Test Suites for Large Software Product Lines

Christopher Henard*, Mike Papadakis*, Gilles Perrouin†,◇, Jacques Klein*, Patrick Heymans†,‡ and Yves Le Traon*
*Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg, Luxembourg
†Precise Research Center In Software Engineering (PReCISE), University of Namur, Namur, Belgium
‡INRIA Lille-Nord Europe, Université Lille 1 – LIFL – CNRS, France
Email: *{firstname.lastname}@uni.lu; †{firstname.lastname}@fundp.ac.be

***Abstract*—Software Product Lines (SPLs) are families of products whose commonalities and variability can be captured by Feature Models (FMs). T-wise testing aims at finding errors triggered by all interactions amongst $t$ features, thus reducing drastically the number of products to test. T-wise testing approaches for SPLs are limited to small values of $t$ – which miss faulty interactions – or limited by the size of the FM. Furthermore, they neither prioritize the products to test nor provide means to finely control the generation process. This paper offers (a) a search-based approach capable of generating products for large SPLs, forming a scalable and flexible alternative to current techniques and (b) prioritization algorithms for any set of products. Experiments conducted on 124 FMs (including large FMs such as the Linux kernel) demonstrate the feasibility and the practicality of our approach.**

***Index Terms*—SPL, Testing, T-wise Interactions, Search-based, Prioritization, Similarity**

## I. Introduction

A *Software Product Line* (SPL) "is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [1]. Features are thus the key to the discrimination of SPL members by showing their commonalities and differences. Features are often organized in a *Feature Model* (FM) [2], [3] which represents all the possible products of the SPL by expressing relationships and constraints between features. Henceforth, we consider a *product* to be a combination of features conforming to the constraints of the FM.

Testing an SPL is an inherently difficult activity [4]. Although testing all the products would be ideal, it is rarely feasible in practice. Indeed, the number of possible configurations (i.e. the products) induced by a given FM usually grows exponentially with the number of features, quickly leading to millions of possible products to test. As a result, test engineers are seeking for solutions to reduce the size of their test suites so that they can meet release deadlines and cost constraints.

Previous work [5], [6] has identified Combinatorial Interaction Testing (CIT) as a relevant approach to reduce the number of products for testing. CIT is a systematic approach for sampling large domains of test data. It is based on the observation that most of the faults are triggered by the interactions between a small number of variables. For example, Kuhn *et al.* [6] have shown that 2-wise or pairwise interactions are able to disclose 80% of the bugs. In some cases, higher interaction strengths ($t$-wise in general) may be needed [7]. Recently, such approaches have been adapted to SPL testing [8], [9], [10], generating products from the FM covering all the valid (with respect to the FM constraints) pairwise combinations of features. Some of them, like [10], also cover $t$-wise .

However, computing all $t$-wise interactions in the presence of constraints, as it is the case for FMs, is known to be NP-complete in the general case [10], [11]. As a result, although $t$-wise generation techniques from FMs have greatly improved, now relying on efficient satisfiability (SAT) solvers [12], higher interaction strengths ($t > 2$) may remain inaccessible for large FMs. This is particularly problematic since 3-wise interactions were shown to commonly appear in SPL testing practice [13]. Since such an exact computation may remain out of reach, one may ask if it is possible to cope with these difficult situations:

[RQ1] *Can we mimic $t$-wise test generation, partially but efficiently while achieving decent coverage?*

While $t$-wise testing drastically reduces the number of products to consider, this number may still be too high to fit the budget allocated for SPL testing. For example, 2-wise coverage for the Linux FM (over 6,000 features) already requires 480 products to be tested [12]. Therefore, being able to prioritize the test suite with the most relevant products is critical. In this paper, the most relevant products are those that exhibit the highest number of $t$ feature interactions. The process of identifying these products is referred to as test prioritization. This forms our second research question:

[RQ2] *What are the most relevant products and how to prioritize them?*

To answer RQ1, we introduce a similarity heuristic [14] given in the form of a fitness function. We assess the suitability of the function to characterize $t$-wise coverage of a given test suite. The intuitive idea underlying this approach is: the more different the products (in terms of selected or unselected

◇FNRS Postdoctoral Researcher.

features), the more likely their ability to cover different $t$-wise interactions. We provide a search-based strategy to generate valid sets of products (i.e. respecting the constraints of the FM) for $t$-wise testing.

To answer RQ2, we first consider that the most important products are those covering the most $t$-wise interactions. Indeed, they can potentially reveal more bugs as they test more feature interactions. We then introduce two prioritization algorithms, named *Greedy* and *Near Optimal*.

Our approach introduces significant flexibility in the testing process: the number of products that can be tested (i.e. fitting the budget) can be specified as well as the time allowed for generating them. The use of similarity has the following two advantages. First, it is very fast to compute. Second, it is independent of the $t$ value. This implies that there is no need to compute and enumerate the huge number of combinations involved in the $t$ feature interactions. The applicability of the proposed strategies is evaluated on both real and generated FMs. This holds even for the largest FM, which contains over 6,000 features. The experimental data and the implementation are publicly available at `http://research.henard.net/SPL/`.

The remainder of this paper is organized as follows: Sections II and III introduce the context and the concepts underlying the proposed approaches. Sections IV and V respectively detail the product prioritization and generation techniques. Section VI reports on the empirical study. Finally, Section VII discuss related work before Section VIII concludes the paper.

## II. Background

### A. SPL Products as Test Cases

In this work, we focus on a model-based testing of SPLs where the variability model is an FM. In this context, one product (an abstract test case) is represented as a set of $n$ features of an FM as $P = \{\pm f_1, ..., \pm f_n\}$, where $+f_i$ indicates a feature which is selected by this product, and $-f_i$ an unselected one. Table I illustrates an example of three products and four features. For instance, product $P_1 = \{+f_1, +f_2, +f_3, -f_4\}$ supports all the features except $f_4$.

### B. T-wise Testing and Coverage

T-wise testing focuses on the interactions between any $t \geq 2$ features of an SPL [11]. Such an interaction is called a $t$-set. It is noted that unselected features are also involved in such interactions. For instance, with reference to Table I, $(+f_1, -f_2, -f_4)$ is a 3-set covered by $P_3$. The ability of a given test suite to find bugs (i.e. its fault detection power) can be estimated by the number of $t$-sets covered by the products of the test suite and is called $t$-wise coverage. In this context, $V_t$ denotes the set of all the valid $t$-sets of a given FM, implying that all the $t$-sets containing incompatible features, the $t$-sets where a given feature is both selected and unselected or the $t$-sets violating the constraints are excluded. More formally, $V_t$ can be expressed as:

$$V_t = \{(\pm f_{x_1}, ..., \pm f_{x_t}) \,|\, f_{x_1}, ..., f_{x_t} \in F \wedge valid(\pm f_{x_1}, ..., \pm f_{x_t})\},$$

where $F$ represents the set of features of an FM and where $valid(\pm f_{x_1}, ..., \pm f_{x_t})$ is a function checking the consistency of a given $t$-set with respect to the FM. Since an FM can easily be translated into a Boolean formula [15], the $valid$ function can be computed using an off-the-shelf SAT solver [16].

Similarly, the valid $t$-sets covered by one particular product $P$ are defined by the subset $v_t^P$, where $v_t^P \subseteq V_t$. In the same lines, a test suite composed of $m$ products $\{P_1, ..., P_m\}$ covers a subset of $V_t$, i.e. $\bigcup_{i=1}^{m} v_t^{P_i} \subseteq V_t$. Thus, test suite coverage can be computed as the ratio of the number of $t$-sets covered by the test suite to the number of total valid $t$-sets:

$$Coverage = \frac{\#\bigcup_{i=1}^{m} v_t^{P_i}}{\#V_t},$$

where $\#A$ denotes the cardinality of the set $A$.

Classical approaches [8], [9], [10] to $t$-wise testing have coverage ratio of 1 as they cover all the $t$-sets. Finally, set coverage redundancy expresses the possibility that, by removing any product, the coverage value is not altered.

TABLE I
Example of Three Products For an FM of Four Features

| | | Features | | | |
|---|---|---|---|---|---|
| | | $f_1$ | $f_2$ | $f_3$ | $f_4$ |
| Products | $P_1$ | × | × | × | |
| | $P_2$ | × | × | | × |
| | $P_3$ | × | | × | |

## III. The Similarity Heuristic

Similarity is an heuristic used here to compare two products. In model-based testing, it has been found that dissimilar test suites have a higher fault detection power than similar ones [14]. The results presented in this paper (Section VI) suggest that two dissimilar products are more likely to cover a greater number of valid $t$-sets than two similar ones.

In this context, we define a distance measure $d$ between two products $P_i$ and $P_j$ to evaluate their degree of similarity. As explained in Section II-A, one product is considered as a set of selected or unselected features. Thus, a straightforward distance measure is a set-based one, like the Jaccard distance [17] or any other set-based distance metrics such as the Dice or Anti Dice measures [14]. In our context, if $P$ represents the possible products of an SPL, the Jaccard distance is mathematically given by:

$$d: \begin{array}{ccc} P \times P & \longrightarrow & [0, 1.0] \\ (P_i, P_j) & \longmapsto & 1 - \frac{\#P_i \cap P_j}{\#P_i \cup P_j}, \text{ where } P_i, P_j \in P. \end{array}$$

The resulting distance varies between 0 and 1. More particularly, a distance equal to 1 indicates that the two considered products are completely different. A distance equal to 0 denotes that the two products are the same (redundant). It is noted that an unselected feature is also an element of the set representing a product. For instance, with reference to Table I, $P_1 = \{+f_1, +f_2, +f_3, -f_4\}$, $P_2 = \{+f_1, +f_2, -f_3, +f_4\}$ and $P_3 = \{+f_1, -f_2, +f_3, -f_4\}$. Thus, $d(P_1, P_2) = 1 - \frac{\#\{+f_1, +f_2\}}{\#\{+f_1, +f_2, +f_3, -f_3, +f_4, -f_4\}} = 1 - \frac{2}{6} \approx 0.67$, $d(P_1, P_3) = 0.4$ and $d(P_2, P_3) \approx 0.86$. In this example, $P_1$ and $P_3$ are the most

similar products (they share the lowest distance), whereas $P_2$ and $P_3$ are the most dissimilar ones. Thus, if we had to choose only two products, $P_2$ and $P_3$ would be the most likely to cover the greatest number of $t$-sets according to the similarity heuristic.

## IV. Product Prioritization

In this section, the similarity distances are used for prioritizing a given set of products. The aim of this process is to order a set $S$ of $m$ products $S = \{P_1, ..., P_m\}$ according to their ability to cover $t$-sets. Therefore, by testing $k \leq m$ products, the greatest possible level of coverage, for any number of $k$ products and any $t$ value, is achieved. More formally [18],

*Given:* a set of products, $S$, the set of all the permutations of $S$, $P_S$ and a function $f$ from $P_S$ to the real numbers, $f : P_S \longrightarrow \mathbb{R}_+$.

*Problem:* finding $S' \in P_S$ such as $(\forall S'' \in P_S | S'' \neq S')[f(S') \geq f(S'')]$. In this context, $f$ is the $t$-wise coverage achieved by $S$. To this end, two algorithms named *Greedy* and *Near Optimal* are introduced. They produce a list $L$, which is the result of the prioritization.

### A. Greedy Prioritization

Informally, this approach iterates over the initial unordered set of products $S$, looking for the two products sharing the maximum distance. These two products are then removed from $S$ and added to the resulting list $L$. This process is repeated until all the products from $S$ are added to $L$. Algorithm 1 formalizes this procedure.

### B. Near Optimal Prioritization

Informally, this approach selects at each step the product which is the most distant to all the products already selected during the previous steps. To this end, the two products belonging to $S$ and sharing the highest distance are first added to $L$. These two products are then removed from $S$. The next step consists in adding to $L$ and removing from $S$ the product sharing the maximum distance to all the products already added to $L$: for each product of $S$, we sum the individual distances with the other products of $L$, thus giving a value for the set. Then the maximum is obtained by comparing these set values (Alg.2, line 11). This process is repeated until $S$ is empty and is more formally described in Algorithm 2.

This technique allows having more diversity than the greedy one for $k < m$ products, but it is computationally more expensive. This is due to the need of calculating all the distances from one product to the others (Alg.2, line 11).

**Algorithm 1** Greedy Prioritization$(S)$

1: **input:** $S = \{P_1, ..., P_m\}$ ▷ Unordered set of $m$ products
2: **output:** $L$ ▷ Prioritized list of $m$ products
3: $L \leftarrow []$
4: **while** $\#S > 0$ **do**
5: **if** $\#S > 1$ **then**
6: Select $P_i, P_j$ from $S$ **where** $max\,(d(P_i, P_j))$
7: ▷ Take the first one in case of equality
8: $L.add(P_i)$
9: $L.add(P_j)$
10: $S \leftarrow S \setminus \{P_i, P_j\}$
11: **else** ▷ $S$ contains only one element
12: $L.add(P_i)$ **where** $P_i \in S$
13: $S \leftarrow \emptyset$
14: **end if**
15: **end while**
16: **return** $L$

**Algorithm 2** Near Optimal Prioritization$(S)$

1: **input:** $S = \{P_1, ..., P_m\}$ ▷ Unordered set of $m$ products
2: **output:** $L$ ▷ Prioritized list of $m$ products
3: $L \leftarrow []$
4: Select $P_i, P_j$ from $S$ **where** $max\,(d(P_i, P_j))$
5: ▷ Take the first ones in case of equality
6: $L.add(P_i)$
7: $L.add(P_j)$
8: $S \leftarrow S \setminus \{P_i, P_j\}$
9: **while** $\#S > 0$ **do**
10: $s \leftarrow size(L)$
11: Select $P_i \in S$ **where** $max\left(\sum_{j=1}^{s} d(P_i, L.get(j)\right)$
12: ▷ Take the first one in case of equality
13: $L.add(P_i)$
14: $S \leftarrow S \setminus \{P_i\}$
15: **end while**
16: **return** $L$

## V. Search-based Product Generation

In this section, we go one step further and take benefit from the similarity heuristic to guide the generation of products (i.e. the test cases). The objective of the test generation process is to provide a set of products that fulfills the requirements of a test criterion. In the present context, this criterion is the $t$-wise coverage. If $P_p$ denotes the set of all the possible products and $P^m$ a set of $m$ products, this process is formally defined as:

*Given:* a FM, the desired number of products, $m$, a given amount of time, $T$, and a function $f$ from $P^m$ to the real numbers, $f : P^m \longrightarrow \mathbb{R}_+$.

*Problem:* finding $P^m \in P_p$ with respect to $T$ such as $[max(f(P_m))]$. In this context, $f$ is the $t$-wise coverage achieved by $S$. Toward this direction, we introduce a test generation approach, based on the (1+1) Evolutionary Algorithm [19]. Specifically, the test generation problem is formulated as a search-based one. Instead of using complex constraints, the space of all the valid products is defined as the search space. Thus, meta-heuristic techniques can be used in order to efficiently explore this space. In view of this, similarity is used as a fitness function towards searching for products in this space. It enables: (a) a computationally interesting approach, as it will be explicitly explained in the following section and (b) prioritizing the generated products without necessitating much additional computation.

### A. A Similarity-based Fitness Function

Our intuition, which will be confirmed in the next section, is that the similarity heuristic is a relevant choice to define a fitness function $f$ to evaluate a set of products. Thus, if we consider a set $S$ of $m$ products $S = \{P_1, ..., P_m\}$, $f$ is formally defined as follows:

$$f : \begin{array}{ccc} P^m & \longrightarrow & \mathbb{R}_+ \\ (P_1, ..., P_m) & \longmapsto & \sum_{j>i\geq 1}^{m} d(P_i, P_j). \end{array}$$

For instance, with reference to Table I and Section III, $f(P_1, P_2, P_3) = d(P_1, P_2) + d(P_1, P_3) + d(P_2, P_3) \approx 1.93$.

This function, which generalizes the similarity distances for $m$ products, allows evaluating the quality of a set of products in terms of $t$-wise coverage. Indeed, the information conveyed by this function is: the higher the fitness value of the given set of $m$ products, the higher the distances between the products, resulting in a potentially higher $t$-wise coverage.

Although evaluating the exact coverage would be a natural choice for a fitness function, say $f_c$, it would be computationally expensive for such a use. Indeed, for each product, it requires computing all the $t$-sets covered by this product. Let us consider an FM with $n$ features and a set of $m$ products. If $\binom{n}{k}$ denotes the binomial coefficient, $f_c$ requires to compute:

$$N = m\binom{n}{t} = \frac{mn!}{t!(n-t)!} \quad (1)$$

$t$-sets to evaluate the coverage of the whole set of products, which represents $N$ operations. On the contrary, $f$ requires:

$$N' = \binom{m}{2} = \frac{m(m-1)}{2}$$

distances computation plus the sum evaluation, which represents $m$ additions.

We assume that $2 \leq t \ll n$. Therefore, the time required to compute one particular distance between two given products is small compared to the coverage evaluation of these two products, i.e. $N \gg N'$. Indeed, $f$ depends neither on $t$ nor on $n$. We also assume that one will test fewer products than the number of features, and thus that $m \ll n$. Especially, in a realistic and industrial context (with large FMs), the testing process is usually subjected to time and budget limitations. It thus does not allow testing as much products as features. It results that $N \gg N'$ and even more while $t$ increases. Recall that we focus on $t$-wise, for high $t$-values. This fact implies a computationally lower cost for $f$ compared to $f_c$. As a result, $f$ is used as the fitness function[1] for the product generation.

### B. Scalable Search-based Product Generation

Classical constraint-based $t$-wise techniques, e.g. [12], are unable to scale to large FMs and to high values of $t$. This is mainly due to the number of $t$ feature combinations. The proposed approach, which is independ of $t$, is composed of two steps. The first one is the generation of valid products using a SAT solver, and the second one is the product selection. The search process is formed by iteratively repeating these two steps. A similar technique that combines constraint solving and search-based approaches in a scalable way has been proposed by Harman *et al.* [20] for mutation-based test generation.

*1) Generating Products:* A SAT solver is used to produce valid products. Once an FM is converted into a Boolean formula [15], the solver can generate valid products. As a result, a search space containing only valid products is formed.

[1]Additional elements showing how the similarity distance presented in Section III and how $f$ are relevant to appreciate the $t$-wise coverage of sets of products can be consulted at `http://research.henard.net/SPL/Resources/twise_similarity.pdf`.

Typically, a product is a correct solver configuration. To this end, the literals of the logical clauses (i.e. clauses represent the constraints of the FM) are assigned values. If the constraints are satisfied, one product is returned. However, assignments to the literals are done in a particular order which involves the following problem: no uniform exploration of the space of all the valid products is possible. Indeed, the order used by the solver to parse the logical clauses and literals enables their prediction. In that case, the approach always returns the same solution in a deterministic way. As a result, the products enumeration is driven by the order used by the solver.

To overcome this issue, and thus to get products in an *unpredictable* way, one solution is to randomize how the solver parses the logical clauses and the literals. It prevents from predicting the next product that will be returned. Additionally, it allows selecting products from the full space instead of enumerating them in a predictable order.

*2) Selecting Products:* Suppose we want to generate and prioritize a list $L$ (i.e. prioritized on the fly during the search-based generation) of $m$ products. From this perspective, the search-based method informally starts by selecting $m$ products in an unpredictable way. Then, these products are evaluated by the fitness function $f$ (see Section V-A). These products define the initial list $L$. Then, by using the distances computed while evaluating $f$, the worst product is determined. The worst product is the one which has the lowest participation in the fitness function. In other words, it is the last element of $L$. The next step consists of trying to replace this product by an unpredictable one got from the solver. This replacement is conserved if and only if the fitness of the resulting list increases. This whole process is repeated during a certain allowed amount of time $t$. It is noted that, at each iteration, the

**Algorithm 3** Search-based Product Generation$(m, t)$

**input:** $m, t$ ▷ Number of products to generate and execution time
**output:** $L$ ▷ List of generated products (prioritized)
$L \leftarrow []$
$S \leftarrow \emptyset$
**for** $i \leftarrow 1$ to $m$ **do**
  $P_{\text{unpredictable}} \leftarrow Request\ to\ the\ solver$
    ▷ If the solver cannot give a new product because it has already iterated over all the valid configurations, reinitialize it
  $S \leftarrow S \cup \{P_{\text{unpredictable}}\}$
**end for**
$s \leftarrow size(L)$
**while** the elapsed time is lower than $t$ **do**
  $fitness \leftarrow f(L.get(1), ..., L.get(s))$
  $L \leftarrow Greedy/Near\ Optimal\ prioritization(S)$ ▷ Facilitated by $f$
  $P_{\text{worst}} \leftarrow L.get(s)$ ▷ $P_{\text{worst}}$ verifies $min\left(\sum_{k=1}^{s} d(P_{\text{worst}}, L.get(k))\right)$
  **repeat**
    $P_{\text{unpredictable}} \leftarrow Request\ to\ the\ solver$
    ▷ If the solver cannot give a new product because it has already iterated over all the valid configurations, reinitialize it
  **until** $P_{\text{unpredictable}} \neq P_{\text{worst}}$
  $L.set(s, P_{\text{unpredictable}})$ ▷ The worst product is replaced
  $newFitness \leftarrow f(L.get(1), ..., L.get(s))$
  **if** $newFitness \leq fitness$ **then** ▷ The new fitness is not better
    $L.set(s, P_{\text{worst}})$ ▷ The worst product is taken back
  **end if**
**end while**
**return** $L$

resulting list $L$ is prioritized based on the distances computed by $f$. This approach is formalized in Algorithm 3.

This technique can be considered as a genetic algorithm without crossover. It can thus be seen as an adaptation of the (1+1) Evolutionary Algorithm [19]. Indeed, instead of removing a random product, the worst ranked product, in terms of fitness, is removed.

## VI. Empirical Study

In this section, the product generation and prioritization approaches are assessed. In test generation, we aim at selecting products providing the highest coverage. In test prioritization, the emphasis is on maximizing the coverage increase observed each time a product is tested. Empirical results regarding the stated research questions along with threats to the validity of this study are presented and analyzed. All the conducted experiments[2] are performed on a Quad Core@2.40 GHz with 24GB of RAM.

The study employs 124 FMs[3] divided into two categories. The first 120 FMs are small to medium size (with a number of features lower or equal to 1000); they are referred to as the *moderate* size FMs. A second subset is composed of 4 FMs of large size; they are referred to as the *large* FMs.

Regarding the moderate size FMs, 10 of them are real and 110 are artificially generated. The real FMs are taken from [21], [22] while the artificial ones are produced with the Software Product Line Online Tools (SPLOT) FM generator [21], [23]. All involved FMs are consistent (i.e. the constraints are possible to fulfill). Details about the moderate FMs are recorded in Table II. It presents: the number of features, the number of valid products[4] and the number of valid 2-sets. Concerning the large FMs, three FMs are real, taken from [22] and one is artificially created. The details of these FMs are recorded in Table III. This table presents, for each FM, the number of features and the number of valid $t$-sets. The number of products cannot be computed in a reasonable amount of time (in days) due to the high number of constraints and features of these FMs.

For the needs of the experiment, the $t$-sets of the moderate FMs are computed using the following procedure for each FM.

[2]The source code of the implemented approaches and the data used for the experiments are available at `http://research.henard.net/SPL/`.

[3]Handled via the SPLAR library [21] and the SAT solver Sat4j [16].

[4]Computed via a Binary Decision Diagram.

TABLE III
4 Large Feature Models

| | eCos 3.0 i386pc [22] | FreeBSD kernel 8.0.0 [22] | Generated FM | Linux kernel 2.6.28.6 [22] |
|---|---|---|---|---|
| #Features | 1,244 | 1,396 | 5,000 | 6,888 |
| #Valid 2-sets | 2,910,229 | 3,765,597 | 49,080,075 | 92,540,449 |
| #Valid 3-sets (≈) | 2.25E9 | 3.44E9 | 1.61E11 | 4.19E11 |
| #Valid 4-sets (≈) | 1.27E12 | 2.34E12 | 3.97E14 | 1.50E15 |
| #Valid 5-sets (≈) | 5.79E14 | 1.26E15 | 7.70E17 | 3.85E18 |
| #Valid 6-sets (≈) | 2.22E17 | 5.76E17 | 1.26E21 | 8.71E21 |

First, a list of all the features of the FM is recovered. Then, all the possible $t$-sets are enumerated and provided to the solver to determine whether they are valid or not. For the large FMs, computing the exact number of valid $t$-sets is a non-trivial and time consuming task. For instance, it took around 3 days to a 10-threaded program running on our system to compute the 92,540,449 valid 2-sets of the Linux FM. As $t$ increases, the number of valid $t$-sets explodes. As a result, for the large FMs and for $t \geq 3$, we estimate the number of $t$-sets. To this end, 1,000 $t$-wise sets are randomly sampled and checked. Since the total number of possible $t$-sets of an FM is known and equal to $\binom{2n}{t}$ for $n$ features ($2n$ because each feature is either selected or unselected), the valid $t$-sets can be directly estimated (law of large numbers). For example, if 800 $t$-sets out of 1,000 sampled are valid, the number of estimated valid $t$-sets is equal to $\frac{800*\binom{2n}{t}}{1{,}000}$.

### A. Product Generation Assessment (RQ1)

Here, we assess the ability of the proposed approaches to cover $t$-sets. A state of the art tool named SPLCAT [10], the most recent tool available handling large FMs, is employed to provide a basis for comparison.

However, SPLCAT and other approaches fail on large FMs for 2-wise. For moderate FMs, SPLCAT does not scale well to 3-wise or above (at least in a reasonable amount of time, in days). As a result, we can only compare our approach with SPLCAT for moderate FMs and 2-wise. As far as we know, our *search-based* approach is the only one which allows scaling to any $t$ value, even for large FMs. Since no other technique can serve as a basis for comparison for the large FMs, we compare the search-based approach with products selected in an unpredictable way from the SAT solver (Section V-B1).

TABLE II
120 Moderate Feature Models

| | Real FMs [21] | | | | | | | | | | Generated FMs | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Cellphone | Counter Strike Simple FM | SPL SimulES, PnP | DS Sample | Electronic Drum | Smart Home v2.2 | Video Player | Model Transformation | Coche Ecologico | Printers | 20 FMs | 20 FMs | 20 FMs | 20 FMs | 20 FMs | 10 FMs |
| #Features | 11 | 24 | 32 | 41 | 52 | 60 | 71 | 88 | 94 | 172 | 15 | 50 | 100 | 200 | 500 | 1,000 |
| #Valid products* (≈) | 14 | 18,176 | 73,728 | 6,912 | 331,776 | 3.87E9 | 4.5E13 | 1.65E13 | 2.32E7 | 1.14E27 | 209.55 | 1.02E8 | 8.56E15 | 3.19E20 | 8.43E80 | 1.81E153 |
| #Valid 2-sets | 151 | 833 | 1,448 | 2,592 | 3,746 | 6,189 | 7,528 | 13,139 | 11,075 | 42,638 | 300.65 | 4,103.2 | 17,367.8 | 71,760.15 | 4.67E5 | 1.88E6 |

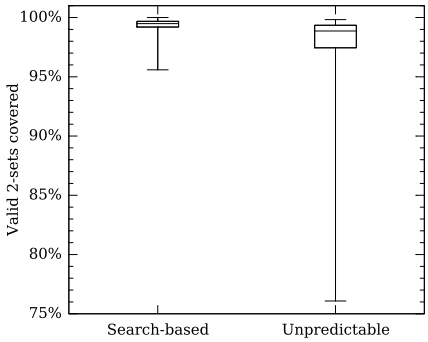



Fig. 1. Product Generation on the 120 Moderate FMs for $t = 2$ (1 Minute Execution for the Search-based Approach, 10 Runs for Each Approach)

In the following, this approach will be referred to as the *unpredictable* one and will also serve as a comparison basis.

*1) Moderate Feature Models:* Here, we compare the search-based approach with both the unpredictable approach and SPLCAT. As a consequence, this study is only based on the 2-wise coverage and considers only the moderate FMs.

*a) Experiment Setup:* To enable a fair comparison, the search-based and unpredictable approaches generate sets of products of the same size as those provided by SPLCAT. The search-based approach is allowed to run for one minute. The results are averaged on 10 runs for each FM.

*b) Experiment Results:* The results are presented in Figure 1. SPLCAT is not represented as it always achieves 100% of coverage. Following this figure, it appears that the proposed search-based approach, as an approximation technique, is close to SPLCAT. Indeed, in the best case, it is able to achieve 100% of 2-wise coverage with only 1 minute of processing time allowed. In the worst case, 95% is achieved. Besides, the search-based approach is much more stable than the unpredictable one, which can drop down to 76% of coverage in the worst case. Although 100% of coverage might be desirable, the focus of our approach, as explicitly stated in the introduction section, is the partial but scalable $t$-wise coverage.

Finally, the performance of SPLCAT varies. For FMs up to 200 features, SPLCAT requires less than a minute. However, it takes around 6.2 minutes for the FMs of more than 200 features, and around 159 minutes for the 1,000 features ones.

*2) Large Feature Models:* Scaling to large FMs is a quite difficult task, even for 2-wise. Neither SPLCAT nor the tools we found (see Section VII) are able to scale well to these sizes [10]. On the contrary, the search-based approach efficiently handles the $t$-wise combinations where no other approach is able to do so, by producing a partial coverage. Here, we evaluate the $t$-wise coverage ability of the search-based and unpredictable approaches on the large FMs for $t = 2, ..., 6$.

*a) Experiment Setup:* Evaluating the $t$-wise coverage requires computing the number of unique $t$-sets covered by the products. However, this is intractable in practice due to the combinatorial explosion (each products covers $\binom{n}{t}$ $t$-sets for $n$ features). Therefore, we sample valid $t$-sets from those covered by the products. The coverage is then estimated based on the sample, considering that the latter represents all the $t$-sets covered by the products. The sampling process is repeated 10 times per each examined $t$ value ($t = 2$ to $t = 6$) with samples of size 100,000. The search-based and unpredictable approaches are executed on all the large FMs to produce 5 times 50 and 100 products, with the time restriction of 30 minutes. Another experiment involves the generation of 1,000 products and the recording of the coverage over the runs of the search-based approach.

*b) Experiment Results:* The results are recorded in Table IV. This table presents the mean coverage achieved with respect to $t$-wise per FM and per approach. Additionally, it records the standard deviation of these values. A score above 90% with respect to 2-wise is achieved by both the approaches and for all the studied FMs when producing 50 products. With respect to 6-wise, scores of 40% to 50% are achieved. By producing 100 products, higher scores are achieved for both the approaches. It should be mentioned, based on the standard deviation values recorded in Table IV, that a small variation on the achieved coverage is observed. It is a fact indicating that the approaches are quite stable.

Generally, the search-based strategy provides a higher coverage compared to the unpredictable approach and especially for high values of $t$. This is true for all the $t$-wise coverage measures. Allowing more time to the search-based technique should increase the gap with the unpredictable approach since the iterations improve the set of products. However, the results are based on the selection of 50 and 100 products. Therefore, the maximum difference between the two approaches lies between the coverage of the unpredictable selection and the maximum possible coverage achievable with 50 or 100 products. Achieving 100% of $t$-wise coverage with 50 or

TABLE IV
T-WISE COVERAGE ACHIEVED (%) PER APPROACH ON THE LARGE FMS WITH 50 AND 100 PRODUCTS

| | | Search-Based | | Unpredictable | | Search-Based | | Unpredictable | |
|---|---|---|---|---|---|---|---|---|---|
| | | 50 products | | | | 100 products | | | |
| FM | $t$-wise | Mean | Std.Dev. | Mean | Std.Dev. | Mean | Std.Dev. | Mean | Std.Dev. |
| eCos | 2 | 99.12 | 0.09 | 98.19 | 0.40 | 99.62 | 0.04 | 99.44 | 0.24 |
| | 3 | 94.53 | 0.18 | 92.24 | 0.61 | 97.55 | 0.11 | 96.92 | 0.46 |
| | 4 | 83.62 | 0.28 | 80.85 | 0.54 | 91.40 | 0.22 | 90.51 | 0.56 |
| | 5 | 67.63 | 0.29 | 65.64 | 0.38 | 80.06 | 0.31 | 79.56 | 0.48 |
| | 6 | 50.11 | 0.29 | 49.36 | 0.26 | 64.79 | 0.34 | 65.05 | 0.30 |
| FreeBSD | 2 | 91.75 | 0.12 | 91.41 | 0.13 | 92.19 | 0.12 | 92.23 | 0.14 |
| | 3 | 85.75 | 0.18 | 83.99 | 0.20 | 87.59 | 0.16 | 87.05 | 0.15 |
| | 4 | 74.94 | 0.19 | 71.67 | 0.24 | 80.82 | 0.14 | 79.09 | 0.15 |
| | 5 | 58.54 | 0.20 | 55.18 | 0.24 | 69.74 | 0.15 | 67.07 | 0.15 |
| | 6 | 40.39 | 0.16 | 37.99 | 0.17 | 54.30 | 0.14 | 51.58 | 0.21 |
| FM Generated | 2 | 99.11 | 0.09 | 94.77 | 0.20 | 99.62 | 0.03 | 97.76 | 0.19 |
| | 3 | 94.53 | 0.18 | 83.91 | 0.22 | 97.55 | 0.11 | 91.10 | 0.27 |
| | 4 | 83.62 | 0.28 | 68.16 | 0.20 | 91.40 | 0.22 | 79.21 | 0.25 |
| | 5 | 67.63 | 0.29 | 50.89 | 0.17 | 80.06 | 0.31 | 63.75 | 0.23 |
| | 6 | 50.11 | 0.29 | 35.16 | 0.18 | 64.79 | 0.34 | 47.53 | 0.26 |
| Linux | 2 | 96.92 | 0.12 | 96.05 | 0.18 | 97.71 | 0.09 | 97.28 | 0.11 |
| | 3 | 91.96 | 0.16 | 90.49 | 0.20 | 94.60 | 0.18 | 93.82 | 0.17 |
| | 4 | 81.37 | 0.18 | 79.51 | 0.19 | 88.53 | 0.20 | 87.42 | 0.23 |
| | 5 | 64.42 | 0.17 | 62.75 | 0.20 | 77.38 | 0.22 | 76.13 | 0.21 |
| | 6 | 45.24 | 0.14 | 44.07 | 0.17 | 61.13 | 0.18 | 60.05 | 0.19 |

100 products seems to be impossible for the large FMs. It is expected that more products are required to achieve 100% of coverage for high values of $t$.

Table V records the coverage achieved by the search-based approach each 5,000 runs repetitions for 1,000 products with respect to 6-wise. Here, we observe that a higher level of coverage is achieved with more products. For instance, the search-based approach achieves 90,671% of 6-wise coverage for the Linux FM. It also shows that allowing more processing time to the approach allows reaching a higher coverage. Indeed, at each 5,000 runs, the coverage recorded is higher than the previous one. Here, the unpredictable approach, represented by the "0 run", is also the initialization stage of the search-based strategy (Alg. 3, lines 5 to 10). For example, considering the eCos FM, 94.191% of 6-wise coverage is achieved at the initialization. After 15,000 runs, it is 95.343%, which represents $\approx$ 2.475744E15 additional 6-sets covered compared to the unpredictable approach. Here, it should be mentioned that the number of valid $t$-sets is extremely high (see Table III) and thus, a small increase in the coverage represents a high increase in the number of additional valid $t$-sets covered. Finally, the 15,000 runs require about 10 to 20 hours of processing time per FM.

*3) Fitness Function:* So far, the presented results suggest that the search-based approach is effective and able to scale to large FMs. Scalability is reached thanks to the ability of the similarity fitness function to mimic the $t$-wise coverage. To illustrate this fact, Table V records the fitness function values with respect to 6-wise coverage for the large FMs as the search-based approach evolves. It shows that the fitness increases with the coverage over the runs of the approach. The same trend holds for all the FMs and values of $t$ considered in this study. Figure 2 illustrates the correlation between the fitness and the $t$-wise coverage for the Linux FM. Therefore, the quality assessment of a set of products can be performed without computing any $t$-set, thanks to the fitness function. Recall that computing the $t$-sets requires vast computational resources (Section V-A, Equation 1).

*4) RQ1:* The experiments conducted for the product generation approach emphasize the following outcomes. First, the similarity heuristic and the fitness function driving the approach form an efficient guide toward the products selection. That technique mimics $t$-wise coverage, does not depend at all on $t$ and thus avoids the combinatorial explosion due to the combinations of $t$ features. Second, the proposed technique is the first one, to the authors' knowledge, which scales well to large FMs while achieving a decent level of $t$-wise coverage (depending on the number of products desired). Finally, in addition to be a close approximation of SPLCAT, it is more flexible than the latter as it allows specifying the processing time and the number of desired products. These are characteristics conforming to an industrial context where the testing process is subjected to budget constraints.

TABLE V
6-WISE COVERAGE AND FITNESS EVOLUTION OVER TIME FOR THE SEARCH-BASED APPROACH ON THE LARGE FMS WITH 1,000 PRODUCTS

| | 0 run (=unpred.) | | 5,000 runs | | 10,000 runs | | 15,000 runs | |
|---|---|---|---|---|---|---|---|---|
| | Cov. | Fit. | Cov. | Fit. | Cov. | Fit. | Cov. | Fit. |
| eCos | 94.191% | 271,880 | 94.225% | 286,304 | 94.263% | 288,039 | 95.343% | 288,818 |
| FreeBSD | 76.236% | 294,184 | 76.395% | 299,962 | 76.465% | 300,892 | 76.494% | 301,634 |
| FM Generated | 82.986 | 258,763 | 84.492% | 263,243 | 84.605% | 263,974 | 84.778% | 264,362 |
| Linux | 89.411% | 296,661 | 90.404% | 298,709 | 90.640% | 299,114 | 90,671% | 299,363 |

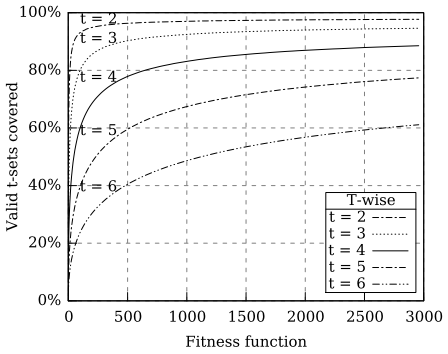



Fig. 2. Fitness Function Correlation with $t$-wise Coverage for the Linux FM

## B. Similarity-based Product Prioritization Assessment

The objective of this part is to evaluate the proposed prioritization approaches. The first experiment focuses on $t = 2$ for the moderate FMs, due to the limitations of SPLCAT (see Section VI-A and VI-A1). The second experiment demonstrates its ability to scale to any $t$ value for the large FMs.

To compare the prioritization approaches, the area under curve is evaluated [24]. This area is the numerical approximation of the integral of the coverage curve and is computed using the trapezoidal rule, i.e. $\int_a^b g(x)dx \approx (b-a)\frac{g(a)+g(b)}{2}$. Thus, for each prioritization method, if $cov(x)$ denotes the percentage of $t$-wise coverage achieved with the $x$-th product, then the area value is given by $\sum_{i=1}^{99}\int_i^{i+1} cov(x)dx = \sum_{i=1}^{99}\frac{cov(i)+cov(i+1)}{2}$. A higher area under curve value expresses a more effective prioritization.

*1) Moderate Feature Models:* This experiment focuses on comparing our prioritization techniques with SPLCAT for $t = 2$. This tool does not prioritize the products, but it tries to cover the maximum of 2-sets each time a product is added. The resulting products can thus be considered as ordered for covering faster the highest amount of 2-sets.

*a) Experiment Setup:* For each moderate FMs, three different sets of products are used to apply the prioritization techniques. The first set is the set of products produced by SPLCAT (Case I). The second one is a set of $n$ products, where $n = \frac{\#features}{2}$, selected with the unpredictable method (Case II). Finally, the last set is composed of the products generated by SPLCAT plus the same amount of products selected by the unpredictable method (Case III). Using these different sets allows ensuring that the prioritization approaches are relevant whatever the nature of the products.

All these sets of products are randomized before executing the prioritization techniques. This practice ensures that our

TABLE VI
PRIORITIZATION RESULTS: AREA UNDER CURVE (SCALE 1:1,000)

| | Case I | Case II | Case III | Case IV with 100 products | | | | | Case IV with 500 products | | | | | Case V with 100 products | | | | | Case V with 500 products | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Technique \ T-wise | 2 | 2 | 2 | 2 | 3 | 4 | 5 | 6 | 2 | 3 | 4 | 5 | 6 | 2 | 3 | 4 | 5 | 6 | 2 | 3 | 4 | 5 | 6 |
| Random | 8.51 | 8.37 | 9.24 | 9.23 | 8.34 | 7.10 | 5.57 | 4.05 | 49.06 | 47.69 | 45.09 | 40.88 | 35.22 | 8.65 | 7.33 | 5.67 | 4.02 | 2.67 | 47.47 | 44.70 | 40.42 | 34.40 | 27.40 |
| Greedy | 8.84 | 8.50 | 9.35 | 9.28 | 8.43 | 7.17 | 5.61 | 4.07 | 49.16 | 47.77 | 45.15 | 40.97 | 35.32 | 8.89 | 7.68 | 6.13 | 4.45 | 3.03 | 47.76 | 45.28 | 41.44 | 36.14 | 29.55 |
| Near Optimal | 8.93 | 8.60 | 9.44 | 9.33 | 8.48 | 7.22 | 5.65 | 4.11 | 49.23 | 47.92 | 45.46 | 41.32 | 35.66 | 9.06 | 8.00 | 6.56 | 4.91 | 3.37 | 48.15 | 46.19 | 42.95 | 38.03 | 31.71 |
| SPLCAT | 8.88 | | | | | | | | | | | | | | | | | | | | | | |

approaches are independent of the original order. On each of the three cases and for each FM, a random prioritization is averaged 10 times. Cases II and III are independently repeated 10 times to avoid any bias from the initial set of products.

*b) Experiment Results:* Table VI presents the area under curve for each case and technique. Recall that a higher surface value indicates a better prioritization. With respect to Table VI and focusing on Case I, we observe the following ordering: Random $<$ Greedy $<$ SPLCAT $<$ Near Optimal. For Case II and Case III, the order Random $<$ Greedy $<$ Near Optimal is observed. Thus, in all the cases, the Near Optimal prioritization provides the best prioritization as its area under curve is the greatest one.

Figure 3 illustrates this behavior. For each case, the results are averaged on all the FMs by normalizing the number of products selected from 0 to 100%. For instance, with respect to Case I, the Near Optimal prioritization approach enables covering more than 90% of the 2-set with only 28% of the products. On the contrary, the random prioritization needs more than 40% of the products. For Case II and Case III (Figure 3b and 3c), the same trends are observed. These results emphasize that the prioritization techniques are either able to perform similarly (Greedy) or better (Near Optimal) as SPLCAT.

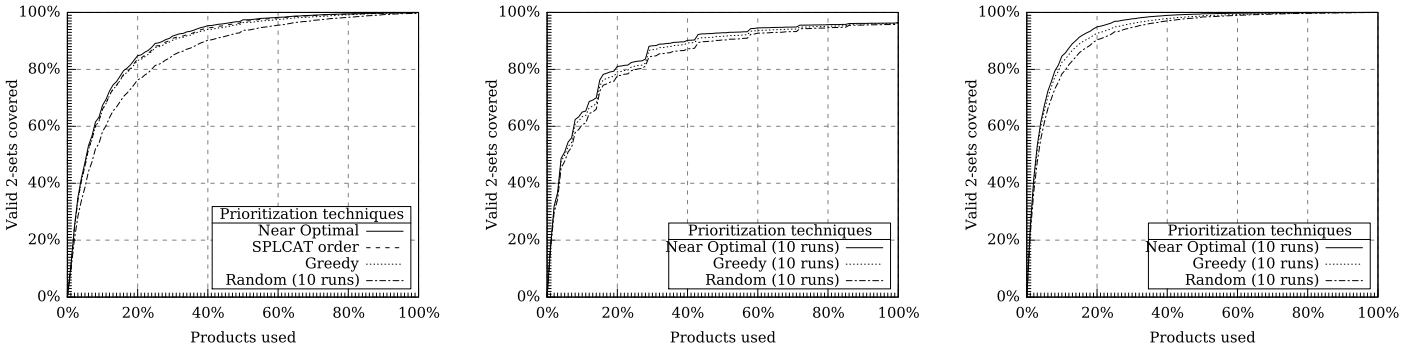
*2) Large Feature Models:* This experiment focuses on the evaluation of the Near Optimal and Greedy prioritization on the large FMs and for $t = 2$ to $t = 6$.

*a) Experiment Setup:* We generate two sets of 100 and 500 products containing dissimilar products (Case IV) and two sets of the same sizes containing half similar and dissimilar products (Case V). We choose these two kinds of sets of products since the prioritization approaches are similarity-driven and can thus be influenced by the nature of the used sets. Indeed, applying these approaches on sets containing dissimilar products can be less effective than applying them on sets containing similar products. We randomize each set of products and execute the Greedy and Near Optimal prioritizations on each of them. We also produce 10 random orderings to compare with our approaches. This practice shows that the prioritization techniques are not affected by random orders.

*b) Experiment Results:* As for the results presented in Section VI-B1b, we evaluate the area under curve. The results are recorded in Table VI. The random is averaged on 10 runs for each value of $t$. The presented values are averaged on the 4 large FMs. We observe the following ordering for both Case IV and Case V: Random $<$ Greedy $<$ Near Optimal. This shows that the prioritizations approaches are relevant for finding the dissimilarities in the sets containing both similar and dissimilar products. The Near Optimal prioritization tends to be the most relevant approach.

As expected, when products are already dissimilar (Case IV), the gain is lesser than when the set of products is any (Case V). Additionally, Figure 4 presents the $t$-wise coverage difference between the Near Optimal prioritization and the random ordering for 500 products, averaged on the 4 FMs. For Case IV (Figure 4a) and $t = 4$, 3% of difference is observed with 30 products selected. For Case V (Figure 4b), 14% of difference is observed with 100 products for $t = 6$.

*3) RQ2:* The experiments conducted for the prioritization bring out the following observations. First, our approaches compete with existing ones that could produce a kind of ordering, like SPLCAT. Second, the most relevant products

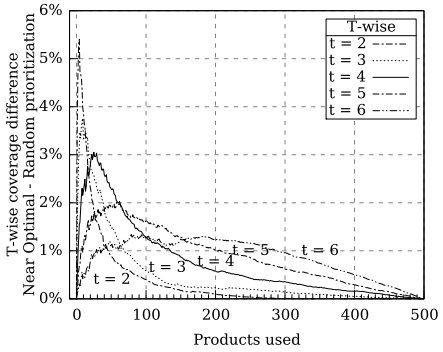



(a) Case I: SPLCAT products

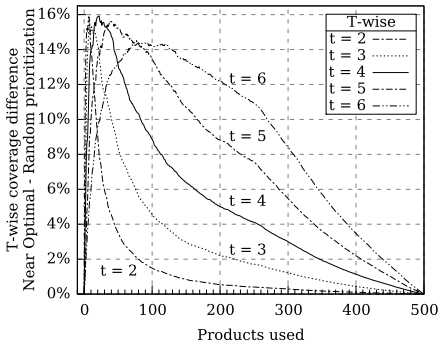
(b) Case II: unpredictable products

(c) Case III: SPLCAT + unpredictable products

Fig. 3. Prioritization on Moderate FMs ($t = 2$)




(a) Dissimilar Set of Products




(b) Similar + Dissimilar Products

Fig. 4. Near Optimal VS Random Prioritization on Large FMs

contributing to $t$-wise coverage are the most dissimilar ones. This is enabled by the similarity heuristic. Finally, the proposed prioritization approaches are able to prioritize any set of products, by looking for the dissimilarities. This is performed without computing any $t$-sets and regardless of the value of $t$.

### C. Threats to Validity

Although we used various FMs, there is an *external validity* threat. Indeed, we cannot ensure that the proposed strategies will provide similar results on different sets of FMs (larger or more constrained). To reduce this threat, we used a relatively large set of 124 FMs of different sizes, combined real and generated FMs to cope with a variety of situations. Additionally, potential errors in our implementation could affect the presented results and lead to *internal validity* threats. To overcome these threats, we divided the implementation into sub stages to have a better control on each of the steps composing the proposed approaches. The comparison with SPLCAT also gave us confidence in our implementation. Besides that, to prevent as possible a *construct validity* threat, we sampled each technique on 10 runs. To enable reproducibility and to reduce the above-mentioned threats, we made our implementation and the experiment data publicly available.

## VII. Related Work

### A. Prioritization

As surveyed by Nie and Leung [25], efforts have been made to prioritize test suites. For instance, Bryce and Colbourn [26] use search-based techniques (e.g. hill climbing) to select the "best test" in terms of $t$-wise coverage. We share their motivation of focusing on the most relevant test. Additionally, the proposed techniques offer improvements over the "natural" ordering provided by the AETG algorithm [5] in line with our experimentations. However, computing $t$-wise coverage for each product is expensive, especially for constrained cases, which are not taken into account in their approach and thus unsuitable in SPLs context. Some approaches combine $t$-wise prioritization and generation (see below). Finally, there are SPL-dedicated efforts, also in the context of test generation, but not directed to $t$-wise, such as Uzuncaova *et al.* [27] work.

### B. T-wise generation

SPL $t$-wise testing approaches typically fall into two categories: *constraint-based* and *search-based*.

*1) Constraint-based Approaches:* Since $t$-wise testing of SPLs is made difficult by the presence of constraints [28], the use of constraint solving solutions have been investigated. In Perrouin *et al.* work [11], a solution based on Alloy, a SAT solver, was devised. The approach was non-predictable in terms of generated solutions and strategies to improve scalability were proposed. Oster *et al.* [8] optimized the problem upfront by flattening the FM and using CIT algorithms [5], [29] within a dedicated constraint solver, producing predictable solutions. Both cannot handle thousand-sized FMs. Recently, SPLCAT [10], used as a reference throughout this paper has been proposed. It also produces predictable solutions and handles larger FMs, but it does not scale to the Linux FM. An optimization of SPLCAT has been recently proposed [12]: it handles larger FMs than SPLCAT but is limited to $t = 3$. While prioritization is induced in some approaches [8], [10], it is not explicit and thus not applicable directly. Logic was also used. Calvagna *et al.* explain how to deal with constraints in CIT [30] by encoding them in first order logic, offer various reductions algorithms to simplify them and use a model checker to solve them. Since this work was not related to FMs, it is difficult to assess its scalability. Hervieu *et al.* [31] also use reduction techniques in the aim of finding the minimal test suite in a Prolog-based implementation. However, this approach does not scale well to FMs of over 200 features, according to our experiments.

*2) Search-based Approaches:* Due to the computational complexity of $t$-wise testing of SPLs, using search-based heuristics is an option. However, we are only aware of two approaches [32], [33]. Garvin *et al.* [32] report on their experience applying and improving an extension to the AETG algorithm [5] using simulated annealing. The simulated annealing approach incrementally populates a *constrained covering array* [28] (which can be simply viewed as a table where lines represent products and columns features, like Table I) by making some moves, i.e. changing the valuation of the features. Each move is controlled by a SAT solver to ensure it is legal with respect to the FM constraints. Moves are guided by a fitness function defined over the remaining pairs to be covered: the fewer pairs to be covered, the lower the proba-

bility to make a move. As we have seen, using pairs coverage as a fitness function induces scalability issues which may be intractable for very large FMs or high $t$ values. Similarly to ours, Ensan *et al.* devised a genetic algorithm approach to generate SPL test suites [33]. They took a fine-grained perspective where each gene is a feature to be mutated and where crossover is applied, inducing possible invalid products which need to be removed. Their fitness function indirectly measures coverage by evaluating the variability points to be bound and the constraints concerned by the features of a product. Rather, we adopt a coarse-grained approach which copes better with large FMs ([33] does not scale over 300 features) and does not produce invalid configurations (since a product is always replaced as a whole). As opposed to other approaches, Ensan *et al.* and our approaches yield partial $t$-wise coverage due to the choice of the fitness function. This, however, allows dealing more easily with time and cost constraints, looking for a "good enough" test suite.

## VIII. Conclusion

T-wise testing aims at finding bugs due to interactions amongst faulty features, which is particularly relevant in an SPL context. However, full $t$-wise testing is NP-complete and scalability an issue: no approach is able to handle high values of $t$ ($\geq 3$) for large feature models in a reasonable amount of time (in days). Moreover, there is no suitable technique supporting the selection of only a fixed number of products, according to a limited budget. In this paper, we tackled these problem by proposing (a) approaches to prioritize products while maximizing the $t$-wise coverage and (b) a scalable and flexible search-based technique to generate products under budget and time constraints for large feature models. Both these techniques are computationally independent from $t$.

Our experiments, performed on 124 feature models for $t = 2$ to $t = 6$, show the feasibility and the scalability of our solutions. We managed to deal with the largest feature models available, such as the Linux kernel ($\approx$ 7,000 features, $\approx$ 200,000 constraints and $\approx$ 8.71E21 valid 6-sets) with up to 90.671% of 6-wise coverage achieved with 1,000 products. Thus, by enabling a partial but scalable $t$-wise coverage and by introducing flexibility in the testing process, our approaches pave the way to a potentially $t$-unrestricted combinatorial interaction testing.

## References


[1] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison Wesley, USA, 2001.

[2] D. Benavides, S. Segura, and A. R. Cortés, "Automated analysis of feature models 20 years later: A literature review," *Inf. Syst.*, vol. 35, no. 6, pp. 615–636, 2010.

[3] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Softw. Engineering Institute, Tech. Rep., Nov. 1990.

[4] J. McGregor, "Testing a software product line," in *Testing Techniques in Software Engineering*. Springer, 2010, vol. 6153, pp. 104–140.

[5] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG System: an approach to testing based on combinatorial design," *IEEE Trans. Softw. Eng.*, vol. 23, no. 7, pp. 437–444, 1997.

[6] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Trans. Softw. Eng.*, vol. 30, no. 6, pp. 418–421, 2004.

[7] R. Kuhn, Y. Lei, and R. Kacker, "Practical combinatorial testing: Beyond pairwise," *IT Professional*, vol. 10, pp. 19–23, 2008.

[8] S. Oster, F. Markert, and P. Ritter, "Automated incremental pairwise testing of software product lines," in *SPLC*. Springer, 2010, pp. 196–210.

[9] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Le Traon, "Pairwise testing for software product lines: A comparison of two approaches," *Softw. Qual. Journal*, 2011.

[10] M. F. Johansen, Ø. Haugen, and F. Fleurey, "Properties of realistic feature models make combinatorial testing of product lines feasible," in *MODELS*. Springer, 2011, pp. 638–652.

[11] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. l. Traon, "Automated and scalable t-wise test case generation strategies for software product lines," in *ICST*. IEEE Computer Society, 2010, pp. 459–468.

[12] M. F. Johansen, O. Haugen, and F. Fleurey, "An Algorithm for Generating t-wise Covering Arrays from Large Feature Models," in *SPLC*. ACM, to appear, 2012.

[13] M. Steffens, S. Oster, M. Lochau, and T. Fogdal, "Industrial evaluation of pairwise spl testing with moso-polite," in *VaMoS*, U. W. Eisenecker, S. Apel, and S. Gnesi, Eds. ACM, 2012, pp. 55–62.

[14] H. Hemmati and L. Briand, "An industrial investigation of similarity measures for model-based test case selection," in *ISSRE*. San Jose, CA, USA: IEEE, 2010, pp. 141–150.

[15] M. Mendonça, A. Wasowski, and K. Czarnecki, "Sat-based analysis of feature models is easy," in *SPLC*, San Francisco, CA, USA, 2009, pp. 231–240.

[16] D. Le Berre and A. Parrain, "The sat4j library, release 2.2, system description," *Journal on Satisfiability, Boolean Modeling and Computation(JSAT)*, vol. 7, pp. 59–64, 2010.

[17] P. Jaccard, "Étude comparative de la distribution florale dans une portion des alpes et des jura," *Bulletin del la Société Vaudoise des Sciences Naturelles*, vol. 37, pp. 547–579, 1901.

[18] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *STVR*, 2010.

[19] S. Droste, T. Jansen, and I. Wegener, "On the analysis of the (1+1) evolutionary algorithm," *Theor. Comput. Sci.*, vol. 276, no. 1-2, pp. 51–81, 2002.

[20] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *ESEC/FSE*. ACM, 2011, pp. 212–222.

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

[22] http://code.google.com/p/linux-variability-analysis-tools/source/browse/?repo=formulas.

[23] M. Mendonca, M. Branco, and D. Cowan, "S.p.l.o.t.: software product lines online tools," in *OOPSLA*. ACM, 2009, pp. 761–762.

[24] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Trans. Softw. Eng.*, vol. 32, no. 9, pp. 733–752, Sep. 2006.

[25] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, no. 2, pp. 11:1–11:29, Feb. 2011.

[26] R. Bryce and C. Colbourn, "One-test-at-a-time heuristic search for interaction test suites," in *GECCO*. ACM, 2007, pp. 1082–1089.

[27] E. Uzuncaova, D. Garcia, S. Khurshid, and D. Batory, "Testing software product lines using incremental test generation," in *ISSRE*. IEEE Computer Society, 2008, pp. 249–258.

[28] M. Cohen, M. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *ISSTA*, vol. 4961/2008, 2007, pp. 129–139.

[29] Y. Lei and K. Tai, "In-parameter-order: a test generation strategy for pairwise testing," in *IEEE High Assurance Systems Engineering Symp.*, 1998, pp. 254–261.

[30] A. Calvagna and A. Gargantini, "A logic-based approach to combinatorial testing with constraints," in *Tests and Proofs*, ser. LNCS, vol. 4966. Springer, 2008, pp. 66–83.

[31] A. Hervieu, B. Baudry, and A. Gotlieb, "Pacogen: Automatic generation of pairwise test configurations from feature models," in *ISSRE*. IEEE, 2011, pp. 120–129.

[32] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "Evaluating improvements to a meta-heuristic search for constrained interaction testing," *Empirical Software Engineering*, vol. 16, no. 1, pp. 61–102, 2011.

[33] F. Ensan, E. Bagheri, and D. Gasevic, "Evolutionary search-based test generation for software product line feature models," in *CAISE*. Gdansk, Poland: Springer, 2012.