Mutation-based Generation of Software Product Line Test Configurations

Christopher Henard, Mike Papadakis, and Yves Le Traon

Interdisciplinary Centre for Security, Reliability and Trust (SnT)
University of Luxembourg, Luxembourg
{christopher.henard, michail.papadakis, yves.letraon}@uni.lu

Abstract. Software Product Lines (SPLs) are families of software products that can be configured and managed through a combination of features. Such products are usually represented with a Feature Model (FM). Testing the entire SPL may not be conceivable due to economical or time constraints and, more simply, because of the large number of potential products. Thus, defining methods for generating test configurations is required, and is now a very active research topic for the testing community. In this context, mutation has recently being advertised as a promising technique. Mutation evaluates the ability of the test suite to detect defective versions of the FM, called mutants. In particular, it has been shown that existing test configurations achieving the mutation criterion correlate with fault detection. Despite the potential benefit of mutation, there is no approach which aims at generating test configurations for SPL with respect to the mutation criterion. In this direction, we introduce a search-based approach which explores the SPL product space to generate product test configurations with the aim of detecting mutants.

Keywords: Software Product Lines, Test Configuration Generation, Search-Based Software Engineering, Mutation, Feature Models

1 Introduction

Software Product Lines (SPLs) extend the concept of reusability by allowing to configure and build tailored software product through a combination of different features [1]. Each feature represents a functionality or an abstraction of a functional requirement of the software product and is itself built from components, objects, modules or subroutines. Thus, an SPL is defined as a family of related software products that can easily be configured and managed, each product sharing common features while having specific ones. The possible products of an SPL are usually represented through a Feature Model (FM) which defines the legal combination between the features of the SPL, facilitates the derivation of new products and enables the automated analysis of the product line [2].

SPLs bring many benefits such as code resuability, a faster time to market, reduced costs and a flexible productivity [3]. However, SPLs are challenging to

test due to the large amount of possible software that can be configured [4]. For instance, 20 optional features lead to 2^{20} possible products to configure, meaning more than a million of different software product that should be tested independently. Such a testing budget is usually unavailable for economic, technical or time reasons, preventing the SPL from being exhaustively tested. Thus, defining methods for generating test suites while giving enough confidence in what is tested is required, and is now a very active research topic for the testing community [5,6,7]. In this respect, Combinatorial Interaction Testing (CIT) [8] is a popular technique that has been applied to SPLs to reduce the size of the test suites. CIT operates by generating only the product configurations exercising feature interactions. While CIT has been shown to be effective for disclosing bugs [9,10], recent work has shown mutation as a promising alternative to the CIT criterion, also correlating with fault detection [11] for existing test suites.

Mutation evaluates the effectiveness of a test suite in terms of its ability to detect faults [12]. It operates by first creating defective versions of the artifact under test, called mutants and then by evaluating the ability of the test suite to detect the introduced mutants. Mutation has been identified as a powerful technique in several work, e.g., [13,14]. In this paper, defective versions of the FM are produced. A mutant is thus an altered version of the rules defining the legal feature associations. Such mutants are useful as they represent faulty implementations of the FMs that should be tested. Thus, while CIT measures the number of feature interactions of the FM exercised by the test suite, mutation measures the number of mutants detected by the test suite. However, and despite the potential benefit of mutation, there is no approach with the purpose of generating product configurations for SPL with respect to the mutation criterion.

Towards this direction, we devise the first approach which generates SPL test configurations using mutation of the FM. Since the SPL product space is too large to be exhaustively explored, we introduce a search-based technique based on the (1+1) Evolutionary Algorithm (EA) [15,16] in conjunction with a constraint solver in order to only deal with products that are conform to the FM. In order to guide the search towards the detection of mutants, four search operators are proposed to both add and remove test configurations from the test suite. The proposed approach solves the challenge of generating a test suite with respect to the mutation criterion. Experiments on 10 FMs show the ability of the proposed approach to generate test suites while with the purpose of mutation.

The remainder of this paper is organized as follows. Section 2 introduces the background concepts underlying the proposed approach. Section 3 describes the approach itself. Section 4 presents the conducted experiments. Finally, Section 5 discusses related work before Section 6 concludes the paper.

2 Background

2.1 Software Product Line Feature Models

A Feature Model (FM) encompasses the different features of the SPL and the constraints linking them. Thus, it defines the possible products that can be

configured in an SPL. For instance, consider the FM of Figure 1. It contains 9 features. Some features are mandatory, which means included in every software product, e.g., `Draw`. There are other type of constraints for the features, such as implications or exclusion. For example, the presence of the `Color` feature in the software product requires the `Color Palette` one to be present too.

The FM can be represented as a boolean formula. In this paper, each constraint is represented in Conjunctive Normal Form (CNF). Such formulas are a conjunction of n clauses $C_1, ..., C_n$, where a clause is a disjunction of m literals. Here, a clause represents a constraint between features of the FM and a literal represent a feature that is selected (f_j) or not ($\overline{f_j}$):

$$FM = \underbrace{\bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{m} l_j \right)}_{\text{constraints}}, \text{ where } l_j = f_j \text{ or } \overline{f_j}.$$

For instance, the FM of Figure 1 encompasses $n = 18$ constraints represented as follows in Conjunctive Normal Form:

$f_1, (\overline{f_2} \vee f_1), (\overline{f_1} \vee f_2), (\overline{f_3} \vee f_1), (\overline{f_1} \vee f_3), (\overline{f_4} \vee f_1), (\overline{f_5} \vee f_1), (\overline{f_1} \vee f_5), (\overline{f_6} \vee f_3), (\overline{f_7} \vee f_3), (\overline{f_3} \vee f_6 \vee f_7), (\overline{f_8} \vee f_5), (\overline{f_9} \vee f_5), (\overline{f_5} \vee f_8 \vee f_9), (\overline{f_8} \vee \overline{f_9}), (\overline{f_7} \vee f_4), (\overline{f_4} \vee \overline{f_8}), (\overline{f_9} \vee f_4)$,
where $RasterGraphicsEditor \mapsto f_1$, $Draw \mapsto f_2$, $Selection \mapsto f_3$, $ColorPalette \mapsto f_4$, $Rendering \mapsto f_5$, $Rectangular \mapsto f_6$, $ByColor \mapsto f_7$, $BlackWhite \mapsto f_8$, $Color \mapsto f_9$.

Thus, with respect to the FM of Figure 1, the corresponding boolean formula is a conjunction between all the constraints:

$$FM = f_1 \wedge (\overline{f_2} \vee f_1) \wedge (\overline{f_1} \vee f_2) \wedge (\overline{f_3} \vee f_1) \wedge (\overline{f_1} \vee f_3) \wedge (\overline{f_4} \vee f_1) \wedge (\overline{f_5} \vee f_1) \wedge (\overline{f_1} \vee f_5) \wedge (\overline{f_6} \vee f_3) \wedge (\overline{f_7} \vee f_3) \wedge (\overline{f_3} \vee f_6 \vee f_7) \wedge (\overline{f_8} \vee f_5) \wedge (\overline{f_9} \vee f_5) \wedge (\overline{f_5} \vee f_8 \vee f_9) \wedge (\overline{f_8} \vee \overline{f_9}) \wedge (\overline{f_7} \vee f_4) \wedge (\overline{f_4} \vee \overline{f_8}) \wedge (\overline{f_9} \vee f_4).$$

2.2 Software Product Line Test Configurations

We denote as a *test configuration* (TC) or *product configuration* to test the list of features of the FM that are present or not in a given product. For instance, with

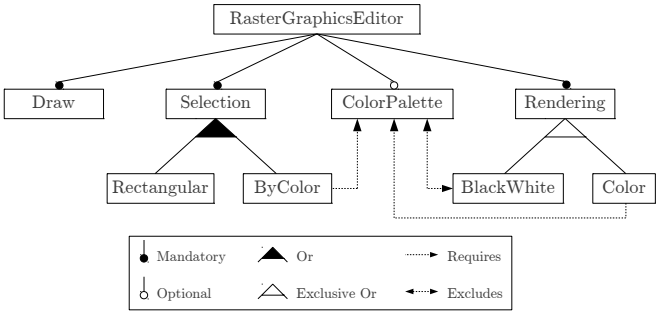


Fig. 1. A Feature Model of a raster graphic editor Software Product Line

respect to Figure 1, $TC_1 = \{f_1, f_2, f_3, f_4, f_5, \overline{f_6}, f_7, \overline{f_8}, f_9\}$ is a TC representing the software product proposing all the features except the rectangular selection and the black and white rendering. This TC satisfies the constraints of the FM that are described in the previous subsection. On the contrary, $TC_2 = \{f_1, \overline{f_2}, f_3, f_4, f_5, \overline{f_6}, f_7, \overline{f_8}, f_9\}$ violates the constraint which specifies that $f_2 =$ `Draw` is a mandatory feature. In the remainder of this paper, only TCs satisfying the constraints of the FM are considered. To this end, a satisfiability (SAT) solver is used. Finally, we denote as *test suite* (TS) a set of TCs.

3 Mutation-based Generation of Software Product Line Test Configurations

The approach for generating TCs starts by creating mutants of the SPL FM. Then, a search-based process based on the (1+1) Evolutionary Algorithm (EA) [15,16] makes use of both the FM and the mutants to produce a set of test configurations. The (1+1) EA is a hill climbing approach which has been proven to be effective in several studies [17,18]. The overview of the approach is depicted in Figure 2. The following sections describe the different steps of the approach.

3.1 Creation of Mutants of the Feature Model

The first step of the approach creates altered versions of the FM. Each altered version is called a mutant and contains a defect within the boolean formula of the FM. For instance, the two following mutants are produced from the FM example of Figure 1:

$$M_1 =$$
$$\overset{\blacktriangledown}{\overline{f_1}} \wedge (\overline{f_2} \vee f_1) \wedge (\overline{f_1} \vee f_2) \wedge (\overline{f_3} \vee f_1) \wedge (\overline{f_1} \vee f_3) \wedge (\overline{f_4} \vee f_1) \wedge (\overline{f_5} \vee f_1) \wedge (\overline{f_1} \vee f_5) \wedge (\overline{f_6} \vee f_3) \wedge (\overline{f_7} \vee f_3) \wedge (\overline{f_3} \vee f_6 \vee f_7) \wedge (\overline{f_8} \vee f_5) \wedge (\overline{f_9} \vee f_5) \wedge (\overline{f_5} \vee f_8 \vee f_9) \wedge (\overline{f_8} \vee \overline{f_9}) \wedge (\overline{f_7} \vee f_4) \wedge (\overline{f_4} \vee f_8) \wedge (\overline{f_9} \vee f_4).$$

$$M_2 =$$
$$f_1 \wedge (\overline{f_2} \overset{\blacktriangledown}{\wedge} f_1) \wedge (\overline{f_1} \vee f_2) \wedge (\overline{f_3} \vee f_1) \wedge (\overline{f_1} \vee f_3) \wedge (\overline{f_4} \vee f_1) \wedge (\overline{f_5} \vee f_1) \wedge (\overline{f_1} \vee f_5) \wedge (\overline{f_6} \vee f_3) \wedge (\overline{f_7} \vee f_3) \wedge (\overline{f_3} \vee f_6 \vee f_7) \wedge (\overline{f_8} \vee f_5) \wedge (\overline{f_9} \vee f_5) \wedge (\overline{f_5} \vee f_8 \vee f_9) \wedge (\overline{f_8} \vee \overline{f_9}) \wedge (\overline{f_7} \vee f_4) \wedge (\overline{f_4} \vee f_8) \wedge (\overline{f_9} \vee f_4).$$

In M_1, a literal has been negated whereas in M_2, an operator OR has been replaced by an AND one. It should be noted that the proposed approach is independent from the way the mutants have been created and from the changes they operate compared to the original FM.

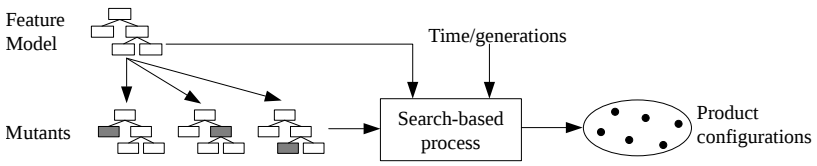


Fig. 2. Overview of the approach for generating test configurations.

3.2 The Search-based Process

Once the mutants are created, the search-based process starts to generate a set of test configurations. The different steps of the approach are described in Algorithm 1 and detailed in the following. First, an initial population is created and its fitness is evaluated (line 1 and 2). Then, the population is evolved (line 3 to 10): search-operators try to improve the population by adding or removing test configurations.

Individual An individual I or potential solution to the problem is a set of $k \geq 1$ test configurations that are conform to the FM: $I = \{TC_1, ..., TC_k\}$.

Population The population P is composed of only one individual: $P = \{I\}$.

Initial Population The individual of the initial population is initialized by generating randomly a test configuration that is conform to the FM by using a SAT solver.

Fitness Evaluation The fitness f of an individual I is calculated by evaluating how many mutants are not satisfied by at least one of the test configurations of I. This is called *mutation score*. More formally, if we denote as $M = \{M_1, ..., M_m\}$ the m mutants of the FM, the fitness f of an individual I is evaluated as follows:

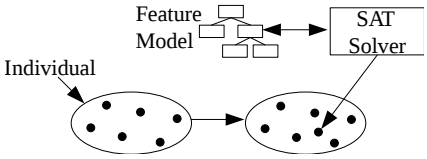
$$F(I) = \frac{|\{M_i \in M \,|\, \exists TC_j \in I \,|\, TC_j \text{ does not satisfy } M_i\}|}{m} = \text{mutation score},$$

where $|A|$ denotes the cardinality of the set A. It should be noted that all the test configurations considered are satisfying the FM since they belong to I.

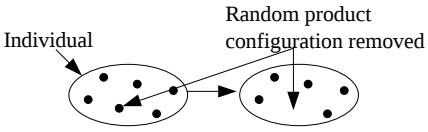
Search Operators The approach makes use of four search operators that operate on an individual I. The operators are divided into two categories: operators that *add* a new test configuration and operators that *remove* a test configuration. The operators are depicted in Figure 3.

Algorithm 1 Generation of SPL Test Configurations

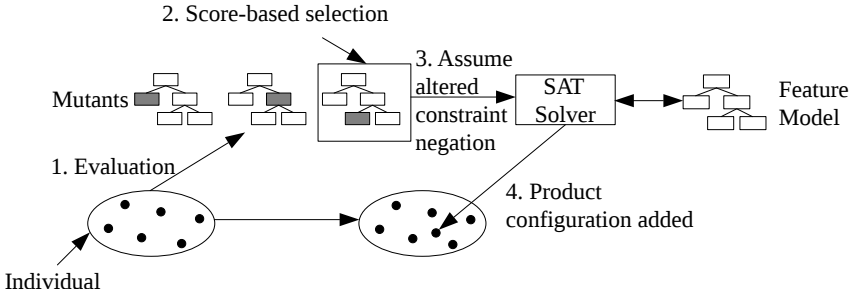
```
1: Create an initial population P with one individual I : P = {I} containing one test configuration
2: Evaluate the fitness f of I : f = F(I)
3: while budget (time, number of generations) do
4:     Select a search operator with a probability p
5:     Generate a new individual I' using the selected search operator
6:     Evaluate the fitness f' = F(I')
7:     if f' ≥ f then
8:         I = I'
9:     end if
10: end while
11: return I
```



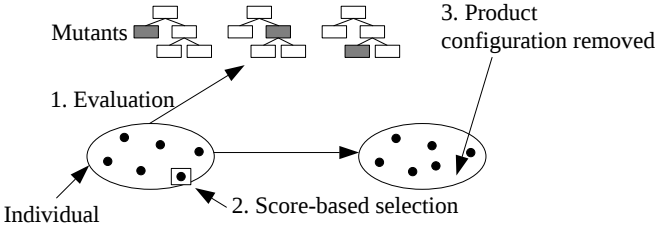
(a) **Add a random test configuration**.



(b) **Remove a random test configuration**



(c) **Smart add of a test configuration**



(d) **Smart remove of a test configuration**

Fig. 3. The search operators used to generate a test configuration set.

- **Add a random test configuration**. This operator is presented in Figure 3(a). It adds to the considered individual a test configuration randomly chosen from the space of all the test configurations of the FM .
- **Remove a random test configuration**. This operator is depicted in Figure 3(b). It randomly removes a test configuration from the individual.
- **Smart add of a test configuration**. This operator is presented in Figure 3(c). First, the altered constraints of the mutants are collected. Then, for each constraint, the number of test configurations from I that do not satisfy it is evaluated. This can be view as a mutant constraint score. Then, using this score, a proportionate selection is performed in order to choose one of these constraints. The idea is to promote the constraint that is the less not satisfied by the test configurations of I. Then, the operators tries to select a test configuration which is at the same time satisfying the FM and the

negation of the selected constraint. Doing so will result in a test configuration that is able to violate a clause of the mutant and thus do not satisfy it.
- **Smart remove of a test configuration**. This operator is illustrated in Figure 3(d). For each test configuration of I, it is evaluated the number of mutants that are not satisfied. This can be view as a test configuration score. Then, using this score, a proportionate selection is performed in order to choose which test configuration to remove from I. The idea is to promote the removal of test configurations that are not satisfying the less amount of mutants.

4 Experiments

In this section, the proposed search-based approach, that we will denote as SB is evaluated on a set of FMs. The objective of these experiments is to answer the two following research questions:

- [RQ1] *Is the proposed approach capable of generating test configurations leading to an improved mutation score?*
- [RQ2] *How does the proposed approach compare with a random one in terms of mutation score and number of test configurations generated?*

The first research question aims at evaluating whether the mutation score is increasing over the generations of SB and if at a point it is able to converge. We expect to see the mutation score increasing over the generations and stabilize at a time. In practice, it means that the approach is capable of improving the solution and reach a good enough mutation score.

The second question amounts to evaluate how SB compares with a naive approach. Since no other technique exists to perform a mutation-based generation of TCs for SPLs, we compare it to a random one. To this end, two bases of comparison are used. The first one is the evaluation of the mutation score when generating the same number of test configurations with both approaches. The second baseline evaluates the number of configurations required by the random approach to achieve the same level of mutation score as SB. It is expected that a higher mutation score than random for the same number of configurations will be observed and we expect a random generation to necessitate more configurations than SB to achieve a given mutation score.

In order to answer these two questions, an experiment is performed on 10 FMs of various size taken from the Software Product Line Online Tools (SPLOT) [19], which is a widely used repository in the literature. The FMs used in the experiments are described in Table 1. For each FM, it presents the number of features, the number of constraints, the number of possible products and the number of mutants used. The mutants have been created using the mutants operator presented in Table 2 and taken from [20,21]. The mutants leading to an invalid FM formula (e.g., $a \wedge \neg a$) and equivalent mutants (mutants that can never be detected because they are always satisfied by any test configuration) are

Table 1. The feature models used for the experiments.

Feature Model	Features	Constraints	Possible products	Mutants
Cellphone	11	22	14	119
Counter Strike	24	35	18,176	208
SPL SimulES, PnP	32	54	73,728	291
DS Sample	41	201	6,912	1,086
Electronic Drum	52	119	331,776	664
Smart Home v2.2	30	82	3.87×10^{9}	434
Video Player	71	99	4.5×10^{13}	582
Model Transformation	88	151	1.65×10^{13}	851
Coche Ecologico	94	191	2.32×10^{7}	1,030
Printers	172	310	1.14×10^{27}	1,829

Table 2. Mutation operators used in the experiments in order to alter feature models.

Mutation Operator	Action
Literal Omission (LO)	A literal is removed
Literal Negation (LN)	A literal is negated
OR Reference (OR)	An OR operator is replaced by AND

not considered in this work. Finally, in order to generate random configurations from the FM and the mutants, the PicoSAT SAT solver [22] is used.

4.1 Approach Assessment (RQ1)

Setup SB has been performed 30 times independently per FM with 1,000 generation with an equal probability $p = 0.25$ to apply one of the four operators.

4.2 Results

The results are recorded in Figure 4 and Table 3. The figure presents the evolution of the mutation score averaged on all the FM and all the 30 runs while the table presents detailed results per FM. With respect to Figure 4, one can see the ability of the approach to improve the mutation score over the generations and stabilize around 0.8. With respect to Table 3, one may observe that the approach is able to improve the mutation score for each of the considered FM, with improvements of 68% in average for the DS Sample FM. Besides, there are very small (0.03) or non-existent variations among the different final mutation score achieved over the 30 runs, fact demonstrating the ability of SB to reach a good solution at each execution of the approach. Finally, it should be noticed that SB achieves the above-mentioned results using only a small number of generations (1,000 generations). This is an achievement since search-based techniques usually require thousands of executions in order to be effective [18].

Table 3. Comparison between the initial and final mutation score achieved by the proposed search-based approach on the 30 runs for 1,000 generations.

	Generation 1			Generation 1,000		
Feature Model \ Mutation score	min	max	avg	min	max	avg
Cellphone	0.39	0.66	0.5	0.79	0.79	0.79
Counter Strike	0.37	0.56	0.45	0.79	0.79	0.79
SPL SimuleES, PnP	0.42	0.62	0.49	0.7	0.7	0.7
DS Sample	0.17	0.27	0.22	0.9	0.9	0.9
Electronic Drum	0.38	0.56	0.44	0.78	0.78	0.78
Smart Home v2.2	0.45	0.66	0.54	0.89	0.89	0.89
Video Player	0.36	0.55	0.45	0.69	0.72	0.71
Model Transformation	0.41	0.61	0.5	0.86	0.86	0.86
Coche Ecologico	0.44	0.57	0.49	0.8	0.8	0.8
Printers	0.35	0.45	0.41	0.74	0.75	0.75

4.3 Answering RQ1

The results presented in the previous section demonstrate the ability of SB to both improve the mutation score over the generations and converge towards an acceptable mutation score. Indeed, some mutants may not be detectable if they are either leading to an invalid formula or an equivalent to the original FM formula (i.e., there is no test configuration that cannot satisfy it), thus limiting the maximum score achievable by the approach. In this work, we only focus on the process of generating the test suite to maximize the mutation score. Finally, we observe improvements in the mutation score of over 60% and a quick convergence, with very small variations between each of the 30 runs, thus giving confidence in the validity of the search approach.

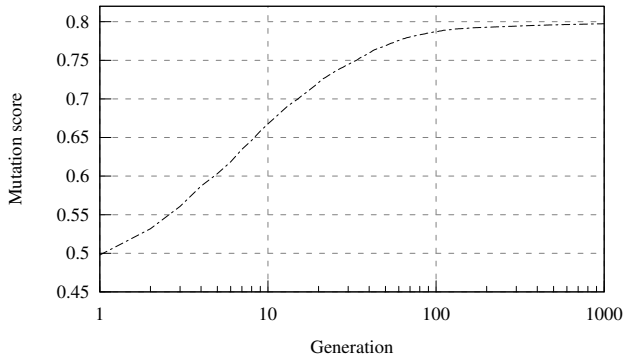


Fig. 4. Evolution of the mutation score over the 1,000 generations of the proposed approach averaged on all the feature models for the 30 runs.

Table 4. Comparison between the search-based approach (SB) and a random one on the following basis: (a) same number of test configurations and (b) same mutation score (MS). Each approach has been performed 30 times independently. #Conf denotes the number of test configurations. The execution time is in seconds.

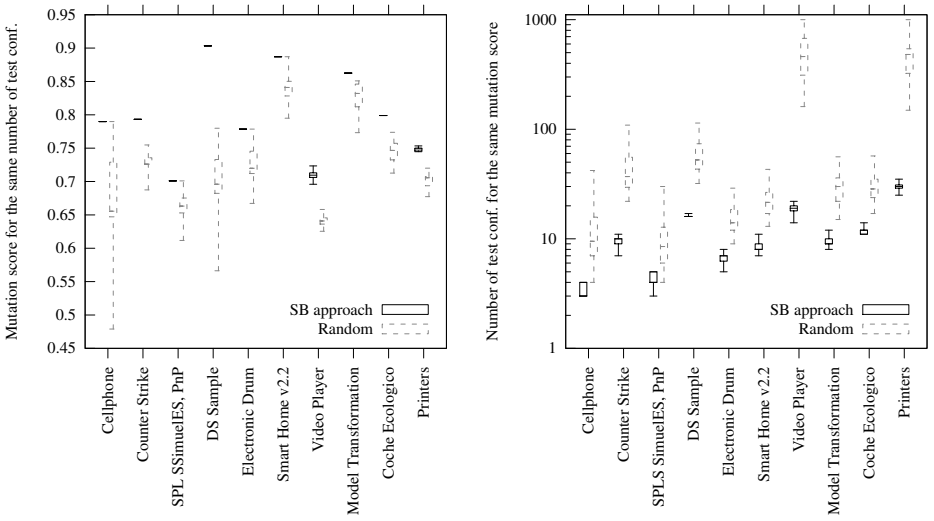
		SB approach			Rand. same #Conf		Rand. same MS	
Feature Model	30 runs	#Conf	MS	Time	MS	Time	#Conf	Time
Cellphone	min	3	0.79	2	0.48	0	4	0
	max	4	0.79	3	0.79	0	42	0
	avg	3.46	0.79	2.66	0.67	0	12.4	0
Counter Strike	min	7	0.8	9	0.68	0	22	0
	max	11	0.8	11	0.75	1	109	2
	avg	9.53	0.8	10.6	0.72	0.16	43.73	0.56
SPL SimulES, PnP	min	3	0.7	11	0.61	0	4	0
	max	5	0.7	13	0.7	1	30	1
	avg	4.36	0.7	11.9	0.66	0.1	9.66	0.16
DS Sample	min	16	0.9	46	0.56	0	32	1
	max	17	0.9	49	0.77	1	114	8
	avg	16.03	0.9	46.8	0.70	0.2	60.26	2.9
Electronic Drum	min	5	0.78	22	0.66	0	9	0
	max	8	0.78	27	0.77	1	29	1
	avg	6.83	0.78	24.8	0.72	0	15.46	0.3
Smart Home v2.2	min	7	0.88	26	0.79	0	13	0
	max	11	0.88	30	0.88	1	43	2
	avg	8.36	0.88	28	0.84	0.1	22.7	0.66
Video Player	min	14	0.69	53	0.62	0	161	19
	max	22	0.72	65	0.65	1	1,000*	532
	avg	18.86	0.71	59	0.64	0.5	518	183
Model Transfo.	min	8	0.86	54	0.77	0	15	0
	max	12	0.86	67	0.85	1	56	4
	avg	9.36	0.86	59.2	0.82	0.2	31.13	1.86
Coche Ecologico	min	11	0.8	75	0.71	0	17	1
	max	14	0.8	89	0.77	1	57	7
	avg	11.76	0.8	80	0.74	0.	31.36	2.9
Printers	min	25	0.74	443	0.67	2	149	110
	max	35	0.75	567	0.72	3	1,000*	4,928
	avg	30	0.75	513	0.70	2.4	481	1,264

*The number of test configurations required by random to achieve the same mutation score as SB has been limited to 1,000.

4.4 Comparison with Random (RQ2)

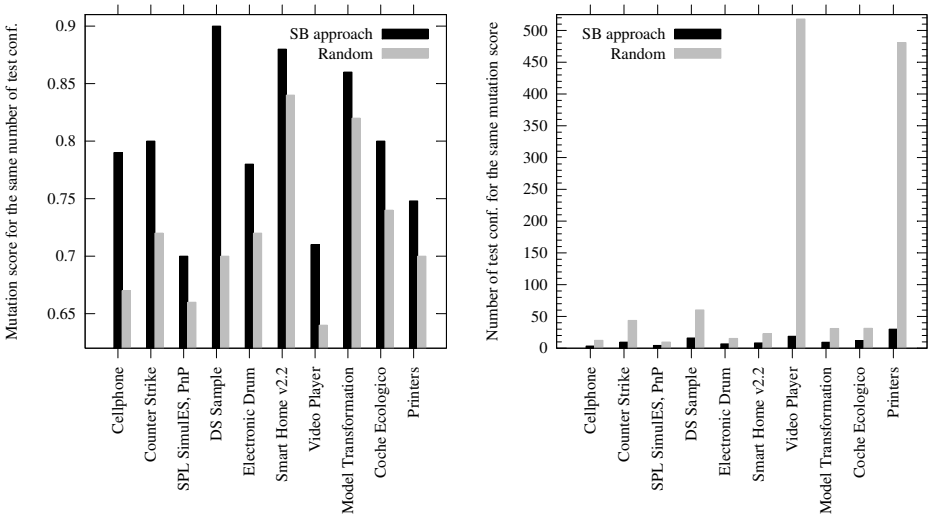
Setup SB has been performed 30 times independently per FM with 1,000 generation allowed. An equal probability $p = 0.25$ to apply one of the four operators has been set. For each run of SB, a random one has been conducted in order to (a) evaluate the mutation score achieved when randomly generating the same number of TCs as the number proposed by SB, and (b) evaluate the amount of generated TCs required by the random approach in order to achieved the same mutation score. In the latter case, a limit of 1,000 TCs has been set.

Results The results are recorded in Table 4. It presents the minimum, maximum and average number of TCs, mutation score (MS) achieved and execution time in seconds for the following approaches: SB, random based on the same number



(a) Baseline number of configurations (b) Baseline mutation score

Fig. 5. Search-based approach VS Random: distribution of the mutation score and number of test configurations on the 30 runs.



(a) Baseline number of configurations (b) Baseline mutation score

Fig. 6. Search-based approach VS Random: average values of the mutation score and number of test configurations on the 30 runs.

of test configurations as SB and random based on the same mutation score as SB. Besides, Figure 5 depicts the distribution of the values over the 30 runs and Figure 6 presents the average values. From these results, one can see that SB is quite stable, with small variations in both the mutation score and number of configurations achieved (5(b) and 5(a)). Compared to random based on the same number of configurations, SB always performs better in terms of mutation score. For instance, for the DS Sample FM, there is a difference of 0.34 on minimum mutation score achieved and 0.2 on the average one (Table 4). Regarding the comparison based on the mutation score, the random approach requires much more TCs to achieve the same mutation score. For instance, with respect to the Video Player FM, the random approach requires in average more than 500 TCs to reach a mutation score of 0.71 while SB only needs less than 20 (Figure 6(a)). In addition, there were some cases, e.g., the Printers FM where the random approach was not able to achieved the same mutation score as the one reached by SB, requiring more than 1,000 TCs and more execution time than SB.

Answering RQ2 Our results show that SB outperforms the random approach. We observed a difference between random and SB of up to 34 % in favor of SB. Additionally, the random technique requires much more test configurations to achieve a given mutation score. In some cases, it is not even able to terminate, requiring more than 20 times more TCs. This shows the ability of SB to generate TCs while at the same time maximizing the mutation score that can be achieved.

4.5 Threats to Validity

The experiments performed in this paper are subject to potential threats towards their validity. First, the FMs employed are only a sample and thus the generalization of theses results to all possible FMs is not certain. In particular, using different models might lead to different results. In order to reduce this threat, we selected 10 FMs of different size and complexity. Thus, we tried to use a diversify and representative set of subjects. A second potential threat can be due to the experiments themselves. First, there is a risk that the observed results happened by chance. To reduce this threat, we have repeated the execution of both the proposed approach and the random one 30 times per FM. Doing so allows reducing risks due to random effects. Another threat can be due to the SAT solver used. Indeed, there is a risk that another solver will lead to different results. We choose the PicoSAT solver as it was easy to modify it to produce random solutions. The same threat holds for the mutation operator used. We tried to employ various mutation operators that are relevant for FM formulas. This paper aims at generating test configurations with the aim of detecting mutants. The ability of finding faults is not evaluated. Regarding the mutation score achieved, it is expected that giving more time to the search-based approach will provide better results. Even if small differences are observed in the mutation score compared to the random approach, this can be in practice leading to finding more faults [11,17]. Finally, the presented results could be erroneous due to potential

bugs within the implementation of the described techniques. To minimize such threats, we divided our implementation into separated modules. We also make publicly available the source code and the data used for the experiments.

5 Related Work

The use of metaheuristic search techniques for the purpose of the automatic generation of test data has been a burgeoning interest for researchers [23]. In this context, several work have investigated the product configuration generation for SPLs using a metaheuristic search. For instance, in [24], Ensan *et al.* aim at generating a test suite with a genetic algorithm by exploring the whole SPL product space, including product that do not fulfill the FM constraints. In this work, we only explore the space of product satisfying the FM, by using a SAT solver. The importance but also the overhead induced by constraint solvers has been shown in [4,25] and used in some work for the purpose of CIT. For instance, in [26], Garvin *et al.* proposed a simulated annealing approach for generating configurations based on CIT. In [27], a multi-objective genetic algorithm in conjunction with a SAT solver was proposed by the authors with CIT as one of the objective to fulfill. There are also work exploring the whole search space to compute the optimal test suite according to CIT, such as [28]. In this work, we propose a simple hill-climbing-based approach in conjunction with the SAT solver. The difference is that we do not consider CIT as an objective. The product configuration generation process is guided by the mutation score. An exact solving technique can only be used for moderate size search spaces. In this paper, we focus on larger SPLs where the product space cannot be fully explored. Finally, there are work focusing on test case generation for each software product [29].

Mutation has been widely used for the purpose of testing and test generation, e.g., [30,31]. With respect to the mutation of models such as FMs for SPL, Henard *et al.* [20] introduced some operators and used mutants of FM in order to evaluate the ability of a given test suite to find them. While the concept of mutation score was used, it was only a way to evaluate a given test suite. In this paper, the mutation score is used to guide the search for generating test suites. Besides, it has been shown that measuring the mutation score of a test suite with respect to a CIT model like FMs rather than measuring the number of interactions covered gives a stronger correlation to code-level faults [11]. In this paper, we used FMs represented as boolean formulas from which we created mutants. There are several work who investigated the mutation of logic formulas. For instance, Gargantini and Fraser devised a technique to generate tests for possible faults of boolean expressions [32]. In this work, the smart add search operator also aims at triggering the fault introduced in the mutant by using a SAT solver. In the context of logic-based testing, Kaminski *et al.* [33] proposed an approach to design tests depending on logical expressions. In this paper, mutation operators are applied on the logic formula of the FM. The objective is only to generate test configurations according to the mutants of the FM formula.

6 Conclusion and Future Work

This paper devised an approach for generating test configurations for SPLs based on mutation. The novelty of the proposed technique is the use of mutation of the FM to guide the search, thus focusing on possible faulty implementation of the FM that should be tested. To the authors knowledge, it is the first approach that is performing so. The conducted experiments show the benefit of the approach compared to a random one as it is able to both reduce the test suite size while significantly increasing the mutation score. To enable the reproducibility of our results, our implementation and the FMs used are publicly available at:

`http://research.henard.net/SPL/SSBSE_2014/`.

Future work spans in the three following directions. First, we will investigate the influence of the parameters and study different variants of the algorithm. In particular, we will compare with standard genetic algorithms. Second, we will propose new operators for improving the search process. Finally, we will undertake supplementary experimentations to further validate the presented findings.

References

1. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering - Foundations, Principles, and Techniques. Springer (2005)
2. Benavides, D., Segura, S., Cortés, A.R.: Automated analysis of feature models 20 years later: A literature review. Inf. Syst. **35**(6) (2010) 615–636
3. Knauber, P., Muñoz, J.B., Böckle, G., do Prado Leite, J.C.S., van der Linden, F., Northrop, L.M., Stark, M., Weiss, D.M.: Quantifying product line benefits. In: PFE. (2001) 155–163
4. Perrouin, G., Sen, S., Klein, J., Baudry, B., Traon, Y.L.: Automated and scalable t-wise test case generation strategies for software product lines (2010)
5. do Carmo Machado, I., McGregor, J.D., de Almeida, E.S.: Strategies for testing products in software product lines. ACM SIGSOFT Software Engineering Notes **37**(6) (2012) 1–8
6. Engström, E., Runeson, P.: Software product line testing - a systematic mapping study. Information & Software Technology **53**(1) (2011) 2–13
7. da Mota Silveira Neto, P.A., do Carmo Machado, I., McGregor, J.D., de Almeida, E.S., de Lemos Meira, S.R.: A systematic mapping study of software product lines testing. Information & Software Technology **53**(5) (2011) 407–423
8. Nie, C., Leung, H.: A survey of combinatorial testing. ACM Comput. Surv. **43**(2) (2011) 11
9. Kuhn, D.R., Wallace, D.R., Gallo, A.M.: Software fault interactions and implications for software testing. IEEE Trans. Software Eng. **30**(6) (2004) 418–421
10. Petke, J., Yoo, S., Cohen, M.B., Harman, M.: Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In: ESEC/SIGSOFT FSE. (2013) 26–36
11. Papadakis, M., Henard, C., Traon, Y.L.: Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing. ICST (2014)

12. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Software Eng. **37**(5) (2011) 649–678
13. Offutt, J.: A mutation carol: Past, present and future. Information & Software Technology **53**(10) (2011) 1098–1107
14. Papadakis, M., Malevris, N.: Mutation based test case generation via a path selection strategy. Information & Software Technology **54**(9) (2012) 915–932
15. Droste, S., Jansen, T., Wegener, I.: On the analysis of the (1+1) evolutionary algorithm. Theor. Comput. Sci. **276**(1-2) (2002) 51–81
16. Lehre, P.K., Yao, X.: Runtime analysis of the (1 + 1) ea on computing unique input output sequences. Inf. Sci. **259** (2014) 510–531
17. Henard, C., Papadakis, M., Perrouin, G., Klein, J., Heymans, P., Le Traon, Y.: Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. IEEE Trans. Software Eng (2014)
18. Harman, M., McMinn, P.: A theoretical and empirical study of search-based testing: Local, global, and hybrid search. IEEE Trans. Software Eng. **36**(2) (2010) 226–247
19. Mendonça, M., Branco, M., Cowan, D.D.: S.p.l.o.t.: software product lines online tools (2009)
20. Henard, C., Papadakis, M., Perrouin, G., Klein, J., Traon, Y.L.: Assessing software product line testing via model-based mutation: An application to similarity testing. In: ICST Workshops. (2013) 188–197
21. Kaminski, G.K., Praphamontripong, U., Ammann, P., Offutt, J.: A logic mutation approach to selective mutation for programs and queries. Information & Software Technology **53**(10) (2011) 1137–1152
22. Biere, A.: Picosat essentials. JSAT **4**(2-4) (2008) 75–97
23. McMinn, P.: Search-based software test data generation: a survey. Softw. Test., Verif. Reliab. **14**(2) (2004) 105–156
24. Ensan, F., Bagheri, E., Gasevic, D.: Evolutionary search-based test generation for software product line feature models. In: CAiSE. (2012) 613–628
25. Cohen, M.B., Dwyer, M.B., Shi, J.: Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. IEEE Trans. Software Eng. **34**(5) (2008) 633–650
26. Garvin, B.J., Cohen, M.B., Dwyer, M.B.: Evaluating improvements to a meta-heuristic search for constrained interaction testing. Volume 16. (2011) 61–102
27. Henard, C., Papadakis, M., Perrouin, G., Klein, J., Traon, Y.L.: Multi-objective test generation for software product lines. In: SPLC. (2013) 62–71
28. Lopez-Herrejon, R.E., Chicano, F., Ferrer, J., Egyed, A., Alba, E.: Multi-objective optimal test suite computation for software product line pairwise testing. In: ICSM. (2013) 404–407
29. Xu, Z., Cohen, M.B., Motycka, W., Rothermel, G.: Continuous test suite augmentation in software product lines. In: SPLC. (2013) 52–61
30. Harman, M., Jia, Y., Langdon, W.B.: Strong higher order mutation-based test data generation. In: SIGSOFT FSE. (2011) 212–222
31. Fraser, G., Zeller, A.: Mutation-driven generation of unit tests and oracles. IEEE Trans. Software Eng. **38**(2) (2012) 278–292
32. Gargantini, A., Fraser, G.: Generating minimal fault detecting test suites for general boolean specifications. Information & Software Technology **53**(11) (2011) 1263–1273
33. Kaminski, G., Ammann, P., Offutt, J.: Improving logic-based testing. Journal of Systems and Software **86**(8) (2013) 2002–2012