Search-based Similarity-driven Behavioural SPL Testing

Xavier Devroey
PReCISE, University of Namur, Belgium
xavier.devroey@unamur.be

Gilles Perrouin
PReCISE, University of Namur, Belgium
gilles.perrouin@unamur.be

Axel Legay
INRIA Rennes Bretagne Atlantique, France
axel.legay@inria.fr

Pierre-Yves Schobbens
PReCISE, University of Namur, Belgium
pierre-yves.schobbens@unamur.be

Patrick Heymans
PReCISE, University of Namur, Belgium
patrick.heymans@unamur.be

ABSTRACT

Dissimilar test cases have been proven to be effective to reveal faults in software systems. In the Software Product Line (SPL) context, this criterion has been applied successfully to mimic combinatorial interaction testing in an efficient and scalable manner by selecting and prioritising most dissimilar configurations of feature models using evolutionary algorithms. In this paper, we extend dissimilarity to behavioural SPL models (FTS) in a search-based approach, and evaluate its effectiveness in terms of product and fault coverage. We investigate different distances as well as as single-objective algorithms, (dissimilarity on actions, random, all-actions). Our results on four case studies show the relevance of dissimilarity-based test generation for behavioural SPL models, especially on the largest case-study where no other approach can match it.

CCS Concepts

•Software and its engineering → Software product lines; Software testing and debugging; •Mathematics of computing → *Evolutionary algorithms;*

Keywords

Software Product Line Testing, Dissimilarity Testing, Featured Transition System

1. INTRODUCTION

During the two last decades, Software Product Line (SPL) engineering has developed many techniques to perform SPL testing [10, 17]. Many of those techniques use model-based testing approaches in order to reduce the testing efforts caused by variability inherent to SPLs. Among those approaches, we may cite Feature Diagram (FD) sampling techniques such as pairwise testing, which generates a set of tested products such that each possible pair of features of the SPL is present in at least one tested product [25, 34, 36].

More recently, the community has switched from the sole selection of products to test to the generation of test cases for SPLs. Delta-oriented SPL testing allows to incrementally build test cases for product lines while ensuring stable coverage [30]. In our previous work [14], we suggested to use Featured Transition System (FTS), a mathematical model defined by Classen *et al.* [8] used to compactly represent the behaviour of a SPL, as a base model for testing. This model-based testing process generates test cases based on the behavioural aspect of a SPL represented by the FTS. We formulated the test case selection from a FTS as an optimisation problem trying to maximise a coverage measure in the FTS (*e.g.,* states, actions, or transitions coverage) while minimizing/maximizing the number of products needed to execute all the test cases [14]. On a more theoretical aspect, Beohar and Mousavi [3] redefined an input output conformance (*ioco*) relation (used to ensure that given implementation is conform to a specification) [37] for FTSs.

We designed a framework to perform behavioural test cases selection for SPLs using FTS as a model of the behaviour of the SPL [14]. We already redefined classical coverage criteria for transition systems [14], implemented and evaluated some of them [15] in our Variability Intensive Behavioural teSting (VIBeS) framework [11]. In this paper, we introduce a new criteria based on a dissimilarity selection approach. The idea is to generate test cases that cover different behaviours and related feature configurations based on distance metrics. It has been empirically shown by Hemmati and Briand [20] that such test cases are more likely to discover bugs. In addition, dissimilarity was shown as effective in SPLs by Henard *et al.* [22] for selection of products to test (to mimic t-wise product selection for large SPLs) in a feature model. Concerning behavioural dissimilarity, Mondal *et al.* [32] showed on code that the dissimilarity-driven test case selection is slightly more effective than code coverage driven approaches. To our knowledge, there is no approach combining feature and behavioural similarity for SPLs.

In this paper, we offer:

- A dissimilarity-driven search-based algorithm that maximises the distance amongst FTS test cases and the distance amongst related feature configurations (product coverage). The algorithm is configurable to provide: *i)* different ways to combine objectives (adding or multiplying them, single/bi-objective dissimilarity) and *ii)*

VaMoS '16, January 27-29, 2016, Salvador, Brazil

DOI: http://dx.doi.org/10.1145/2866614.2866627

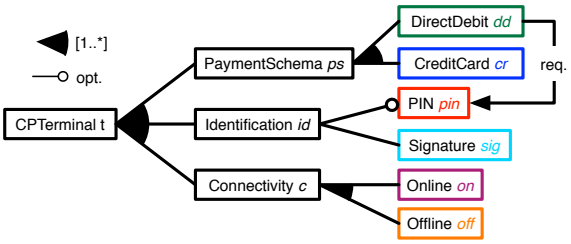

Figure 1: Card payment terminal FD

different distances at the behavioural level (Hamming, Levensthein, Jaccard, Dice, Anti-dice) [18], feature configuration distances are computed using the Jaccard index [24].

- The evaluation of this algorithm on four case studies, showing the effectiveness of our approach with respect to random and all-actions. We explore the influence of different distances (Hamming, Levensthein, Jaccard, Dice, Anti-dice) on the results, showing for example that, amongst the 120 configurations of the algorithm, the single-objective version using Hamming distance provides optimal performance on the largest case study.
- The implementation and evaluation results of our approach are available on the following website: https://projects.info.unamur.be/vibes/

In the remainder of this paper, Section 2 presents SPL modelling and dissimilar test cases generation; Section 3 presents our approach with its evaluation and results in Section 4; Section 5 presents the related work; and finally Section 6 concludes the paper as well as outlines future perspectives.

2. BACKGROUND

This section introduces the behavioural modelling of SPLs with Feature Diagrams (FDs) and Featured Transition Systems (FTSs), and dissimilarity testing.

2.1 Software Product Line Modelling

Variability Modelling. Software Product Line engineering is based on the idea that (software) products of a family (i.e, a product line) can be built by reusing assets, some common to all products and some specific to a subset of the product line. This variability is captured through the notion of *feature* and organised in a *Feature Diagram* (FD) [26]. For instance, Fig. 1 presents the FD of a card payment terminal. This machine t accepts card payment with a certain payment schema ps (direct debit dd and/or credit card cr), using a card owner authentication method (signature sig and optionally PIN code pin), and with a synchronous on or asynchronous off connection to the payment service.

Behavioural Modelling. Various modelling techniques exist to represent behavioural aspects of a SPL. Classen *et al.* [8] define *Featured Transition Systems* (FTSs) to compactly represent the behaviour of a SPL as a Transition System (TS) where transitions are labelled with feature expressions, *i.e.,* Boolean expressions over features representing the set of products able to fire the transition. Formally, a FTS is defined as a tuple $(S, Act, trans, i, d, \gamma)$,

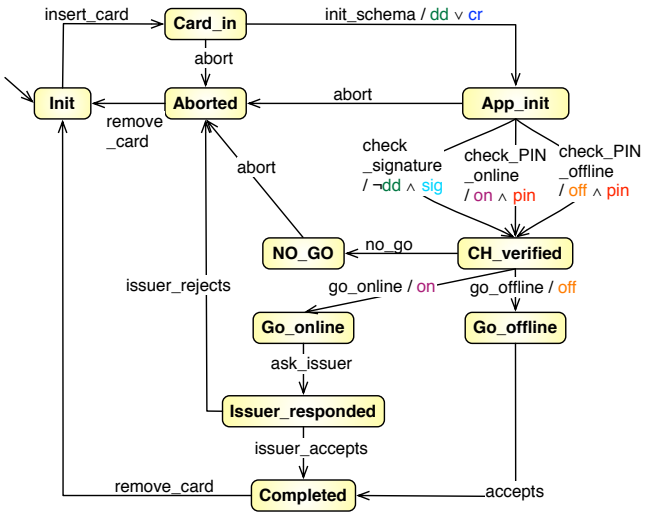

Figure 2: Card payment terminal FTS

where: (i) S is a set of states; (ii) Act a set of actions; (iii) $trans \subseteq S \times Act \times S$ is the transition relation (with $(s_1, \alpha, s_2) \in trans$ sometimes noted $s_1 \xrightarrow{\alpha} s_2$); (iv) i the initial state of the FTS; (v) d is a FD; (vi) $\gamma : trans \rightarrow [\![d]\!] \rightarrow \{\top, \bot\}$ is a total function labelling each transition with a Boolean expression over the features, which specifies the products that can execute the transition ($[\![d]\!]$ denotes the semantics of the FD d, *i.e.,* all the different products that may be derived from d). For instance, the FTS in Fig. 2 represents the behaviour of a card payment terminal SPL corresponding to the FD in Fig. 1: for instance $App\_Init \xrightarrow{check\_PIN\_online/pin \wedge on} CH\_verified$ is labelled with the $pin \wedge on$ feature expression meaning that only products with the PIN (pin) and on-line (on) feature may fire this transition.

2.2 Dissimilarity Testing

Dissimilarity testing is a technique used to select a subset of a set of test cases, which aims to maximise the fault detection rate, by increasing diversity among test cases [5,19]. This diversity is characterised by a *dissimilarity heuristic* defined over the different test cases. For instance, in behavioural model-based testing, one may define a distance between two test cases defined as sequences of actions (α_1, α_2, ..., α_n) in a TS as the number of α_i actions that differ from one test case to the other (*i.e.,* the number of actions in the first test case that are not in the second and vice versa). Hemmati *et al.* [19] empirically demonstrate that in single system model-based testing, dissimilar test suites find more faults than similar ones. Mondal *et al.* explored code coverage and test case diversity interact in fault-finding abilities of test suites [32]. Results are better for diversity-based test suites, though results are overlapping. The authors conclude that coverage and diversity may complement each other nicely in a multi-objective search-based scenario.

SPLs Dissimilar Test Case Generation. Henard *et al.* [22] applied dissimilarity testing to SPL in order to generate and prioritize products to test. The idea was to mimic the combinatorial interaction testing (CIT) generation for SPLs [29, 34], in which valid combinations of features are

covered at least once. CIT-based generation for large SPLs raises a computational challenge because of the number of features and constraints involved, forming a large and complex search space. To deal with this, a search-based (1+1) algorithm [16] was designed: an initial population of products (computed from a FD using a SAT solver) is evolved in order to maximise the distance amongst the combinations of features (*i.e.,* the products). This distance, based on the Jaccard index [24], serves as the fitness function, improving the global coverage of the population. This approach has shown good results for large values of interaction strength and large FDs (up to 7000 features) while allowing the tester to specify their test budget. The relevance of this strategy was independently confirmed by Al-Hajjaji *et al.* [1].

Considering this body of knowledge, we combine dissimilarity for SPLs at the product level, that maximise product coverage, with test case dissimilarity, that maximise behaviour coverage, to provide a bi-objective algorithm presented in the next section.

3. APPROACH

Prior to the introduction of our bi-objective algorithm, we define FTS test cases and explain how they can be generated randomly.

FTS Test Case. A test case tc is a sequence of actions $(\alpha_1, \ldots, \alpha_n)$ in a FTS (fts) that may be executed by at least one product in $[\![d]\!]$ (representing all the valid products of the FD d). Formally:

$$\exists p \in [\![d]\!] : fts_{|p} \overset{(\alpha_1,\ldots,\alpha_n)}{\Longrightarrow}$$

Where $fts_{|p}$ represents the projection of fts using p (*i.e.,* the transition system representing the behaviour of the product p obtained by evaluating the feature expressions in fts using p as $true/false$ assignment for the features) [8], and $fts_{|p} \overset{(\alpha_1,\ldots,\alpha_n)}{\Longrightarrow}$ is equivalent to $i \overset{(\alpha_1,\ldots,\alpha_n)}{\Longrightarrow}$, meaning that there exists a state $s_k \in S$ with a sequence of transitions labelled $(\alpha_1, \ldots, \alpha_n)$ from i to s_k in the projection of the FTS onto p. The set of FD products able to execute this test case is defined as:

$$prod(fts, tc) = \{p \in [\![d]\!] \mid fts_{|p} \overset{(\alpha_1,\ldots,\alpha_n)}{\Longrightarrow}\}$$

For instance, the test case $nogo = (insert\_card, init\_schema, check\_PIN\_online, no\_go, abort, remove\_card)$ may be executed by products with the *dd* or *cd* features and *on* and *pin* features, $prod(cpterminal, nogo)$ will contain 4 products.

As in other classical model-based testing approaches, the generated (abstract) test cases will be concretize using some mapping [28] in order to have a (concrete) test case directly executable by the system. This last step is beyond the scope of this paper and left for future work.

3.1 Random FTS Test Case Generation

In our previous work [15] we presented a first implementation of a random test case generation algorithm. This algorithm was not optimal as it performs validation *a posteriori*: it first generates a sequence of actions using the FTS without considering feature expressions and verifies afterwards that this sequence may be executed by a valid product of the product line using a SAT solver. In this paper, we improve this algorithm to directly consider sequences of actions executable by a valid product. Our random algorithm, which is presented in Algorithm 1, takes as input a FTS without deadlock (which may be verified beforehand using the ProVeLines FTS model checker [9]) and produces a random test case executable by at least one product of the SPL. The algorithm loops while we are not back to the initial state. At each iteration, a transition is selected such as if its associated action is added to the test case, at least one product of the product line may execute it (line 3), the action of this transition is then added to the test case (line 5).

```
   Input: fts: a FTS without deadlock
   Output: A random test case
   Data: tc: test case; next: state; t: transition
 1 begin
 2   tc = ();
 3   while next ≠ i do
 4     t = random({(s_i, a, s_j) ∈ fts.trans |
 5       prod(fts, tc + a) ≠ ∅});
 6     tc = tc + t.a;
 7     next = t.s_j;
 8   end
 9   return tc;
10 end
```

Algorithm 1: Random FTS test case selection algorithm

3.2 Bi-objective Test case Generation

Our bi-objective test case generation algorithm tries to maximise both the product and the behaviour coverage of a set of test cases. To do so, it will compute a *dissimilarity distance* between test cases based on the products able to execute each test case (using the FTS) and the actions appearing in those test cases. Two test cases are dissimilar (distance equals 1) if they do not share the same actions and they may be executed on dissimilar products. Formally, considering a FTS fts the dissimilarity between 2 test cases $tc_1 = (\alpha_1, \ldots, \alpha_n)$ and $tc_2 = (\beta_1, \ldots, \beta_n)$ derived from this FTS is defined as:

$$\begin{aligned} diss(fts, tc_1, tc_2) = \quad & diss_p(prod(fts, tc_1), prod(fts, tc_2)) \\ & \otimes diss_a((\alpha_1, \ldots, \alpha_n), (\beta_1, \ldots, \beta_n)) \end{aligned}$$

Where $diss_p : [\![d]\!] \times [\![d]\!] \to [0, 1.0]$ computes a dissimilarity distance between the products, $diss_a : Act^+ \times Act^+ \to [0, 1.0]$ computes a dissimilarity distance between the actions, and $\otimes : [0, 1.0] \times [0, 1.0] \to [0, 1.0]$ is an operator combining the products and actions distances to return a global dissimilarity distance between the two test cases. In our evaluation in Section 4, we use the multiplication ($\times$) and average (avg) operators.

Product Dissimilarity. To compute the product dissimilarity distance ($diss_p$), we use the Jaccard index [24] which shows good results in the work of Henard *et al.* [22]. This index is a *set-based* dissimilarity distance, computed using the following formula:

$$diss_p(s_1, s_2) = 1 - \frac{\#(s_1 \cap s_2)}{\#(s_1 \cup s_2)}$$

Actions Dissimilarity. The dissimilarity distance between two sequences of actions may be computed using *sequence-based* distances or *set-based* distances (like the Jaccard index) if we assimilate sequences of actions to sets of actions [19]. In our evaluation in Section 4, we consider both *set-based* distances (Hamming, Jaccard, dice, and anti-dice

```
Input: fts: a FTS without deadlock; k: the number of
       test cases; d: the duration
Output: A set of test cases
1  begin
2      s = {};
3      for i ∈ [0; k[ do
4          s.append(random(fts));
5      end
6      start = time();
7      while time() < start + time() do
8          sort(s);
9          candidate = s.copy().removeLast()
10             .append(random(fts));
11         if fit(fts, candidate) > fit(fts, s) then
12             s = candidate;
13         end
14     end
15     return s;
16 end
```

Algorithm 2: Search-based dissimilarity selection algorithm

distances) and a *sequence-based* distance: the Levenshtein (or edit) distance. The Levenshtein distance between two sequences indicates the number of insertion, deletion and replacement operations to perform on the first sequence to obtain the second one [18]. This number, divided by the maximal length between the two sequences, gives us a dissimilarity measure.

Fitness Function. The product and actions dissimilarity distances are used to compute the *fitness value* in a (1+1) without mutation nor crossover evolutionary algorithm [22] (Algorithm 2 explained hereafter) to characterize a set of test cases: $fit : FTS \times Act^+ \times \ldots \times Act^+ \rightarrow \mathbb{R}_+$:

$$fit : (fts, tc_1, \ldots, tc_k) \mapsto \sum_{j>i}^{k} dist(fts, tc_i, tc_j)$$

Dissimilarity Selection Algorithm. First, the algorithm initialises a random set of test cases (line 3) with k elements, this set is improved in order to maximise the fitness value during a given time d (line 7). At each iteration of the main loop, the current set is sorted using the dissimilarity distance between test cases (line 8). Sorting is performed by considering either *local distances* or *global distances*. For local distances, the sort computes the distances between every pair of test cases: $\forall i, j \in [0; k[, dist[i, j] = diss(fts, s[i], s[j])$. It will then select the pair of test cases such as $dist[i, j]$ is maximal and put them at the beginning of the list. This process loops until all elements of the set have been processed. The global distance works in the same way, except that the selected pair has also to be as dissimilar as possible from the previously selected pairs[1]. At the end of the sorting algorithm, the most globally or locally fittest elements are at the beginning of the list s. The next step is to replace the last element of this list by a new random test case (line 10), if the fitness value of this new set (*candidate*) is better than the previous one, this candidate becomes the new set of test cases (line 12). Hemmati *et al.* evaluated 320 similarity scenarios, including those where a new test case is not randomly generated and found the (1+1) strategy to be cost-effective [19]. Contrary to feature-based dissimilarity [1, 22], a SAT solver is only used for $prod(FTS, tc)$ and cannot affect generation.

[1]For more information about local and global sorting, the interested reader may refer to [22].

Table 1: Models characteristics with the number of states, transitions and actions, the average input/output degree of the states in the FTS, and the number of features and possible products of the FD.

Model	St.	Tr.	Act.	Avg. deg.	Feat.	Prod.
S. V. Mach.	9	12	13	1.44	9	24
Minepump	25	41	23	1.64	9	32
C. P. Term.	11	17	16	1.55	21	4,774
Claroline	106	2053	106	19.37	44	5.406.720

Table 2: Execution time ($d_{allactions}$) and number of test cases ($k_{allactions}$) measured for the 6 all-actions coverage test sets generation.

Model	Time ($d_{allactions}$)		T.c. count ($k_{allactions}$)	
	Avg.	*Std.*	*Avg.*	*Std.*
S. V. Mach.	1.03 sec.	0.093	3.86	0.35
Minepump	1.18 sec.	0.189	11.14	0.99
C. P. Term.	1.24 sec.	0.263	5.0	0.76
Claroline	3.42 sec.	1.814	52.86	2.95

4. EVALUATION

To assess our approach presented in Section 3 we define the following research questions: **RQ.1** How does the similarity-driven search based approach compare to all-actions and random test selection with respect to fault finding and product coverage? **RQ.2** How does the choice of a given distance influences the results?

The evaluation has been done on 4 case studies and implemented in VIBeS [11], our model-based testing framework dedicated to variability intensive systems[2].

4.1 Setup

We consider 4 models from different sources with different sizes as input to different test case selection processes. In order to avoid bias using random generation, we run the evaluation 6 times for each model and each configuration of the algorithm (6 configurations overall) presented in this section.

Models. The four case studies are: the Soda Vending Machine representing a SPL of beverage vending machine [7], the Minepump representing a SPL of pumps used to pump water from a mine while avoiding methane explosion [7], the card payment terminal presented in Fig. 2, and the Claroline-based system representing the navigational usage mined from an Apache weblog of a highly configurable course platform used at the University of Namur [13]. Table 1 presents the characteristics of the models: the number of states, transitions, and actions; the average input/output degree of the states; the number of features in the FD; and the number of possible products for this FD.

Test Cases Generation. For each model, we generate a set of test cases which satisfies the *all-actions* coverage criteria (*i.e.,* when executing all the test cases, all the actions of the system are executed at least once) and measure

[2]All the models and tools may be downloaded at https://projects.info.unamur.be/vibes/.

Table 3: Number of faults seeded in the models.

Model	F. Sta.		F. Tran.		F. Act.	
	Avg.	*Std.*	*Avg.*	*Std.*	*Avg.*	*Std.*
S. V. Mach.	4.6	0.8	5.9	1.0	5.7	1.0
Minepump	12.4	1.4	19.3	1.7	9.6	1.5
C. P. Term.	5.3	0.9	7.9	1.1	6.8	1.1
Claroline	52.1	2.9	896.8	13.3	32.3	2.9

the generation time ($d_{allactions}$). The number of test cases in this set ($k_{allactions}$) and the generation time ($d_{allactions}$), presented in table 2 for the 6 executions, are used as input for Algorithm 2. To assess time impact (d) on the results, we consider the following values: $1 \times d_{allactions}$, $2 \times d_{allactions}$, $10 \times d_{allactions}$, $100 \times d_{allactions}$ for each action dissimilarity distance described in Sec. 3.2 and using local and global distances for the sort in Algorithm 2. Rows and columns in table 4 show the parameters values used for Algorithm 2: the top rows indicate the actions dissimilarity distance ($diss_a$) and the operator ($\otimes$) used to combine it with the product distance ($diss_p$) or if the actions dissimilarity distance is used alone, *i.e.,* in a single-objective configuration of the algorithm (denoted by *Sing.* in the table); the leftmost columns indicate which sorting method is used (global or local) and the time considered for the algorithm ($1 \times d_{allactions}$, $2 \times d_{allactions}$, $10 \times d_{allactions}$, or $100 \times d_{allactions}$). A random set of $k_{allactions}$ test cases is also generated using Algorithm 1. In total, we generated 122 sets of test cases for each model.

Fault Injection and Test Sets Execution. To measure the quality of the generated test sets, we use fault seeding, a very popular method to assess the fault-finding ability of a test set [31]. As in our previous work [15], we randomly select states, transitions, and actions to mark them as faulty (*i.e.,* containing a fault), if a state/transition/action is selected more than once, it is only counted as 1 during the fault detection. We then execute the test set on the FTS and consider that a fault is revealed as soon as the faulty states is reached, the faulty transitions are fired, and the faulty actions are executed. Random selection has an upper bound of 66% of faults of the states, actions, and transitions of the FTS. This allows to compare in fine grain way the different approaches. Table 3 presents the average number of faults seeded in the different models during the evaluation.

4.2 Results

Fig. 3 presents the coverage distribution of the different sets of test cases generated using Algorithm 2 (◯), all-actions coverage (■), and random (♦) algorithms. The X axis is the percentage of faults (states, transitions, or actions) discovered when executing the set of test cases (fault cov.). The Y axis is the percentage of products covered by the set of test cases (prod. cov.): for a set s and a FD d, it corresponds to $\frac{\#prod(d,s)}{\#[\![d]\!]}$.

To characterize the sets of test cases (*i.e.,* , the solution space of our bi-objective generation), we compute a *reference front*, by taking the Pareto front of *all* the points in Fig. 3. This reference front contains all the sets of test cases maximising the fault and products coverages (*i.e.,* , the best solutions). We give hereafter for each model a podium with the 3 (or more if they have the same frequency) optimal configurations of Algorithm 2 providing solutions that are on the reference front:

- **Soda vending machine** (47 optimal solutions)
  1. Hamming avg., global, $t = 10$ ($freq. = 0.056$)
  2. Hamming avg., global, $t = 1$ ($freq. = 0.056$)
  3. Hamming avg., global, $t = 2$ ($freq. = 0.056$)
  4. Hamming avg., global, $t = 100$ ($freq. = 0.056$)
- **Minepump** (6 optimal solutions)
  1. Jaccard sing., global, $t = 2$ ($freq. = 0.222$)
  2. Jaccard avg., global, $t = 10$ ($freq. = 0.222$)
  3. Antidice sing., global, $t = 1$ ($freq. = 0.222$)
- **Card payment terminal** (64 optimal solutions)
  1. Levenshtein sing., global, $t = 2$ ($freq. = 0.029$)
  2. Antidice sing., global, $t = 10$ ($freq. = 0.029$)
  3. Levenshtein sing., global, $t = 2$ ($freq. = 0.029$)
  4. Antidice mul., global, $t = 2$ ($freq. = 0.029$)
  5. Antidice avg., global, $t = 2$ ($freq. = 0.029$)
- **Claroline** (1 optimal solution)
  1. Hamming sing., global, $t = 100$ ($freq. = 1.0$)

Finally, table 4 presents the hypervolume values for the Claroline model for the different sets of test cases. The hypervolume corresponds, for a set of test cases s, to the volume of the solution space dominated by s [4, 21]. A high value of hypervolume correspond to a set of test cases with a better fault and product coverage.

The raw results for the 6 executions of the different test case selection algorithms and their fault finding evaluation may be downloaded at http://projects.info.unamur.be/vibes

4.3 Discussion

Dissimilarity relevance. Regarding *RQ.1*, dissimilarity-based approaches are always able to obtain the optimal results in terms of fault finding ability and coverage. On the three small models, these results are sometimes matched by the all-actions and random approaches (Fig. 3). However, the latter appear less frequently: neither random nor all-actions are on the podium of optimal solutions in terms of frequency. Additionally on the Claroline case study, the only optimal solution found is a search-based one. We therefore confirm the good results of similarity-driven testing for single product testing [32] and configuration selection at the FD level [22] for behavioural test case selection in an SPL context. The most important finding is that being fully bi-objective is not necessarily an advantage: on the 13 approaches present in the frequency podiums, only 6 are bi-objective. Additionally, a single objective approach dominates alone the Claroline case. This may be due to the nature of the case study, which is not heavily constrained: it is easy to obtain by chance dissimilar configurations. All-actions performance may benefit of this situation as well. Time may be involved in the explanation: for a given amount of time, a bi-objective configuration will necessarily iterate less than a single-objective one. When bi-objective is optimal, the average (*avg*) composition operator gives the best results as only one *mul* approach appears on our podiums. Time given to the search-based algorithm has an (expected) influence on the quality of the results. This is apparent on

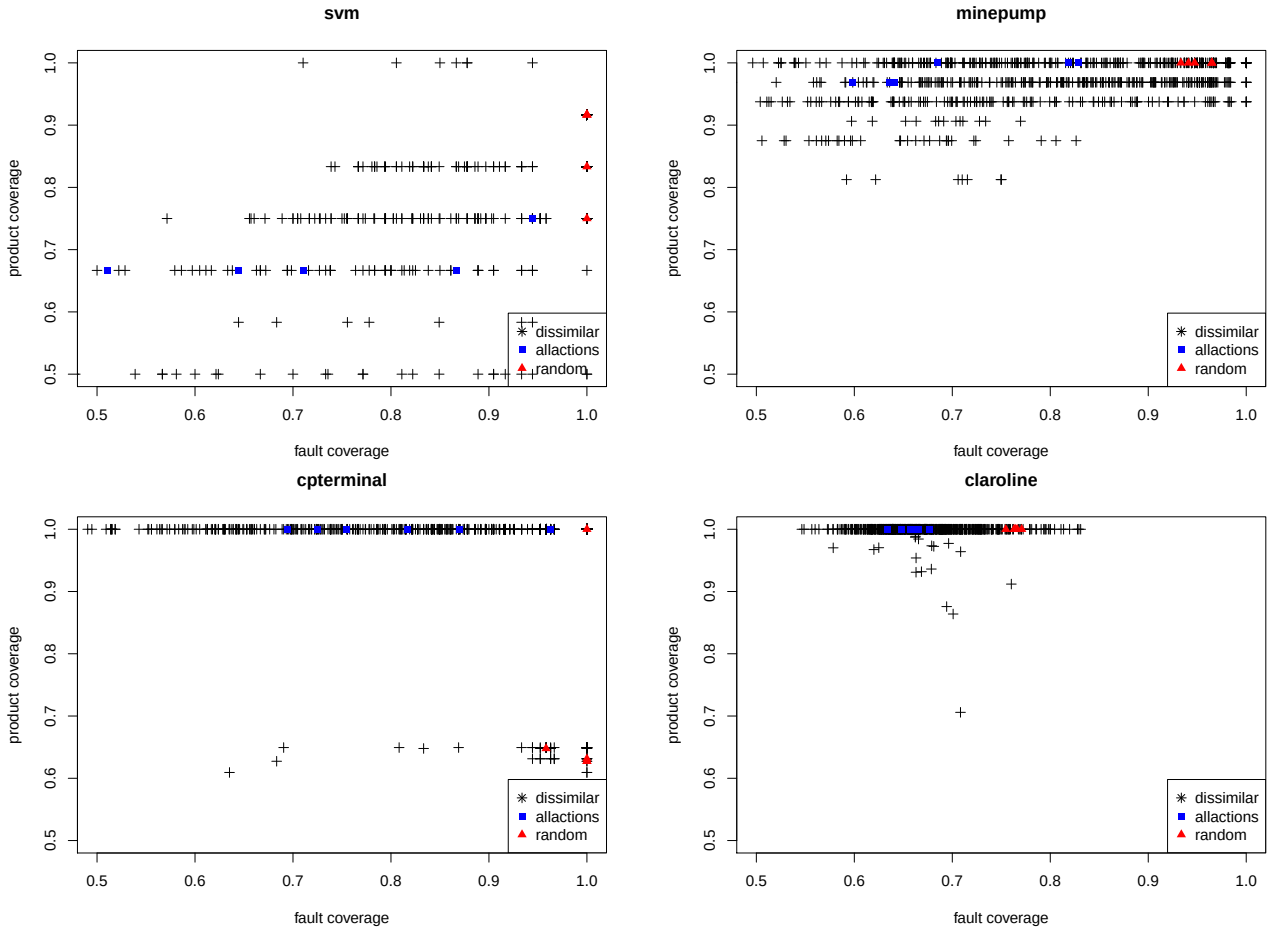

Figure 3: Coverage distribution for the sets of test cases for the soda vending machine (svm), Minepump, card payment terminal (cpterminal), and Claroline models.

the Claroline case where the best hypervolumes are obtained by approaches that are given $t = 100$.

Distance Impact. If we consider all the podiums, Hamming and Jaccard-based distances (Dice, Antidice, Jaccard) clearly win over Levenshtein. This may seem surprising since Levenshtein is the only one that is sequence-based taking into account the order of the actions. Levenshtein is more computationally expensive than Hamming and Jaccard-based distances, implying less iterations of the algorithm for a given amount of time. When employed alone (single-objective) on actions, it appears to be the second most performing distance on the Claroline case.

4.4 Threats to Validity

Internal Validity. Our evaluations have been applied to only 4 models. To mitigate this risk, we chose one example model (Soda Vending Machine), one academic models (Minepump), one hand-crafted model based on card payment documentation and norms (Card payment terminal), and one larger real world model (Claroline). Those models come from different sources and represents different kinds of systems: the card payment terminal, the Minepump, and the Soda Vending machine are embedded system designed by engineers and Claroline is a web-based platform where the model has been reverse engineered from a running instance.

Construct Validity. (i) To keep the comparison fair between the different test set generation algorithms, we use the same number of test cases and duration time of the all-actions test case generation to parametrize random and dissimilar test case generations. As the dissimilar test case generation is based on a (1+1) evolutionary algorithm [16], it is very sensitive to the maximal execution time (d). The all-actions test case generation may be very fast (for the Soda Vending machine for instance). In order to assess the time influence in the quality of generation for the evolutionary approach, we chose to repeat the test case generation with different d values. (ii) We choose to use a (1+1) evolutionary algorithm [16] to maximize the dissimilarity of the generated test set. This algorithm is simple and to parametrize, and it showed good results to select products to test [22]. Many other algorithms, like adaptive random testing [6], used to generate dissimilar test cases exist [19]. A comparison between those different algorithms is left for future work. (iii) The complete process described in Section 4.1 has been repeated 6 times for each model on a Ubuntu Linux machine (Linux version 3.13.0-65-generic, Ubuntu 4.8.2-19ubuntu1) with an Intel Core i3 (3.10GHz) processor and 4GB of mem-

Table 4: Hypervolumes values for the Claroline case-study

		Actions dissimilarity distance														
		Hamming			**Jaccard**			**Dice**			**Antidice**			**Levenshtein**		
	t	*Avg.*	*Mul.*	*Sing.*	*Avg.*	*Mul.*	*Sing.*	*Avg.*	*Mul.*	*Sing.*	*Avg.*	*Mul.*	*Sing.*	*Avg.*	*Mul.*	*Sing.*
	1	0.688	0.662	0.690	0.673	0.705	0.690	0.693	0.709	0.690	0.711	0.692	0.668	0.691	0.722	0.691
Loc.	*2*	0.674	0.699	0.705	0.679	0.673	0.684	0.667	0.689	0.670	0.688	0.690	0.659	0.666	0.662	0.677
sort	*10*	0.702	0.681	0.661	0.721	0.685	0.700	0.720	0.669	0.679	0.667	0.674	0.651	0.691	0.681	0.696
	100	0.667	0.693	0.672	0.672	0.720	0.713	0.691	0.703	0.678	0.696	0.685	0.655	0.655	0.678	0.687
	1	0.736	0.710	0.724	0.694	0.697	0.727	0.695	0.683	0.710	0.666	0.672	0.699	0.684	0.693	0.717
Glob.	*2*	0.711	0.708	0.755	0.722	0.718	0.715	0.723	0.670	0.729	0.677	0.705	0.728	0.708	0.687	0.733
sort	*10*	0.740	0.733	0.804	0.723	0.696	0.730	0.690	0.683	0.738	0.701	0.692	0.746	0.701	0.714	0.771
	100	0.794	0.800	0.831	0.747	0.729	0.756	0.750	0.714	0.755	0.747	0.740	0.758	0.747	0.730	0.781
All-act. crit.		0.771														
Rand. crit.		0.706														

ory. The complete experiment took approximately 4 days.

External Validity. We cannot guaranty that our models are representative of real behavioural models. The Soda Vending Machine, Minepump, and Card payment terminal models are small with few states and transitions. The Claroline model is a larger model reverse engineered from a running web application, this gave us a model with a very flexible navigation (*i.e.,* a large number of transitions) between states (representing pages) with very few exclusive constraints (*i.e.,* feature expressions) on the transitions. This has the side effect to allow many products to execute a large part of the FTS with few behaviours limited to a small subset of the SPL. However, the diversity of the models sources as well as the diversity of considered systems gives us good confidence in the possibilities of this approach. In our future work, we will apply our approach on other kinds of systems from various domains where the transitions from state to state are more constrained and/or where the SPL has specific behaviour exercised only by a small subset of the possible products.

5. RELATED WORK

Dissimilarity in Product Line Testing. To the best of our knowledge, our approach is the first to consider dissimilarity for behaviour and configurations in SPL testing, research on the topic has solely focused on generating dissimilar configurations of the feature model: Henard *et al.* [22] defined a product selection approach based on selected features dissimilarity to mimic the t-wise coverage for large systems and high values of t. This approach has been effective also on smaller systems (with fewer products) by Al-Hajjaji *et al.* [1]. As mentioned in Section 1, all these approaches are based on the pioneering works on dissimilarity testing at the code level (*e.g.,* [20, 32]).

Fault Injection. Fault seeding techniques is a very popular method used to assess the quality of a set of test cases. In our evaluation, we randomly tag states, actions and transitions as faulty. Other approaches include injection of known bugs in the system under test and mutation testing [31, 33]. In mutation testing, a program is modified (mutated) using a mutation operator (*e.g.,* replacing an *and* operator by a *or* operator). The number of modified programs (mutants) detected by the test set under evaluation is called mutation score and gives an indication on the fault finding ability of the test set [2]. Classical mutation testing works on code artefacts of an application. With the development of model-based techniques, mutation testing techniques working on model artefacts of an application (*e.g.,* to assess a set of abstract test cases) have been developed [35]. In our previous work, we suggested to use FTSs to represent all the mutants of a TS in a single model and therefore (re)define a set of mutations operators for TSs [12]. Other approaches also define mutation operators for FDs [23] or features to other artefacts mapping [27]. However, to the best of our knowledge, mutation operators for FTSs themselves do not exist (yet). Development of such operators are part of our future work in order to assess the presented and future approaches.

6. CONCLUSION AND FUTURE WORKS

In this paper, we have provided a configurable search-based approach supporting single and bi-objective similarity-driven test case selection for behavioural SPLs. Through 4 cases studies, we have demonstrated the superiority of dissimilarity over random and all-actions coverage. We discussed the fact that being bi-objective is not necessarily an advantage in our case studies. The examination of different types of distances has shown that Hamming and Jaccard-based distances are the most efficient.

Future work will investigate further single/bi-objective differences on larger and more constrained case studies. We will assess other selection algorithms like adaptive random testing [6]; together with other sequence-based distances such as global/local alignment, Needleman-Wunsch, and Smith-Waterman [19]; and other kinds of objectives such as test case cost and/or execution time.

Acknowledgements

The authors would like to thank Christopher Henard and Mike Papadakis for their valuable and helpful comments.

7. REFERENCES

[1] M. Al-Hajjaji, T. Thüm, J. Meinicke, M. Lochau, and G. Saake. Similarity-based Prioritization in Software Product-line Testing. In *SPLC - Vol. 1*, pages 197–206, 2014.

[2] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE TSE*, 32:608–624, 2006.

[3] H. Beohar and M. R. Mousavi. Input-output Conformance Testing Based on Featured Transition Systems. In *SAC*, pages 1272–1278, 2014.

[4] D. Brockhoff, T. Friedrich, and F. Neumann. Analyzing hypervolume indicator based algorithms. In *Parallel Problem Solving from Nature – PPSN X*, pages 651–660. 2008.

[5] E. G. Cartaxo, P. D. L. Machado, and F. G. O. Neto. On the use of a similarity function for test case selection in the context of model-based testing. *STVR*, 21(2):75–100, 2011.

[6] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse. Adaptive Random Testing: The ART of test case diversity. *JSS*, 83:60–66, 2010.

[7] A. Classen. Modelling with FTS: a Collection of Illustrative Examples. Technical Report P-CS-TR SPLMC-00000001, PReCISE Research Center, University of Namur, 2010.

[8] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. J.-F. Raskin. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE TSE*, 39:1069–1089, 2013.

[9] M. Cordy, A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. ProVeLines: A Product Line of Verifiers for Software Product Lines. In *SPLC - Vol. 2*, pages 141–146, 2013.

[10] P. A. da Mota Silveira Neto, I. do Carmo Machado, J. D. McGregor, E. S. de Almeida, and S. R. de Lemos Meira. A systematic mapping study of software product lines testing. *IST*, 53(5):407–423, 2011.

[11] X. Devroey and G. Perrouin. Variability Intensive system Behavioural teSting framework (VIBeS), 2014. version 1.1.2.

[12] X. Devroey, G. Perrouin, M. Cordy, M. Papadakis, A. Legay, and P.-Y. Schobbens. A Variability Perspective of Mutation Analysis. In *FSE*, pages 841–844, 2014.

[13] X. Devroey, G. Perrouin, M. Cordy, P.-y. Schobbens, A. Legay, and P. Heymans. Towards Statistical Prioritization for Software Product Lines Testing. In *VAMOS*, 2014.

[14] X. Devroey, G. Perrouin, A. Legay, M. Cordy, P.-Y. Schobbens, and P. Heymans. Coverage Criteria for Behavioural Testing of Software Product Lines. In *ISoLA*, pages 336–350, 2014.

[15] X. Devroey, G. Perrouin, and P.-Y. Schobbens. Abstract Test Case Generation for Behavioural Testing of Software Product Lines. In *SPLC - Vol. 2*, pages 86–93, 2014.

[16] S. Droste, T. Jansen, and I. Wegener. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science*, 276(1-2):51–81, apr 2002.

[17] E. Engström and P. Runeson. Software product line testing – A systematic mapping study. *IST*, 53:2–13, 2011.

[18] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. 1997.

[19] H. Hemmati, A. Arcuri, and L. Briand. Achieving scalable model-based testing through test case diversity. *ACM TOSEM*, 22:1–42, 2013.

[20] H. Hemmati and L. Briand. An Industrial Investigation of Similarity Measures for Model-Based Test Case Selection. In *ISSRE*, pages 141–150, 2010.

[21] C. Henard, M. Papadakis, M. Harman, and Y. L. Traon. Combining multi-objective search and constraint solving for configuring large software product lines. In *ICSE*, pages 517–528, 2015.

[22] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. *IEEE TSE*, 40:650–670, 2014.

[23] C. Henard, M. Papadakis, G. Perrouin, J. Klein, Y. Le Traon, and Y. L. Traon. Assessing Software Product Line Testing Via Model-Based Mutation: An Application to Similarity Testing. *ICSTW*, pages 188–197, 2013.

[24] P. Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin del la Société Vaudoise des Sciences Naturelles*, 37:547–579, 1901.

[25] M. F. Johansen, b. Haugen, and F. Fleurey. An Algorithm for Generating T-wise Covering Arrays from Large Feature Models. In *SPLC - Vol. 1*, pages 46–55, 2012.

[26] K. Kang, S. Cohen, J. Hess, W. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University, Software Engineering Institute, 1990.

[27] H. Lackner and M. Schmidt. Towards the Assessment of Software Product Line Tests: A Mutation System for Variable Systems. In *SPLC - Vol. 2*, pages 62–69, 2014.

[28] N. Li and J. Offutt. A test automation language framework for behavioral models. In *ICSTW*, pages 1–10, 2015.

[29] M. Lochau, S. Oster, U. Goltz, and A. Schürr. Model-based pairwise testing for feature interaction coverage in software product line engineering. *SQJ*, 20:567–604, 2011.

[30] M. Lochau, I. Schaefer, J. Kamischke, and S. Lity. Incremental Model-Based Testing of Delta-Oriented Software Product Lines. In *TAP*, pages 67–82, 2012.

[31] A. P. Mathur. *Foundations of software testing*. 2008.

[32] D. Mondal, H. Hemmati, and S. Durocher. Exploring Test Suite Diversification and Code Coverage in Multi-Objective Test Case Selection. In *ICST*, pages 1–10, 2015.

[33] M. Papadakis, C. Henard, and Y. le Traon. Sampling Program Inputs with Mutation Analysis: Going Beyond Combinatorial Interaction Testing. In *ICST*, pages 1–10, 2014.

[34] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. le Traon. Pairwise testing for software product lines: Comparison of two approaches. *SQJ*, 20:605–643, 2011.

[35] S. C. Pinto Ferraz Fabbri, M. E. Delamaro, J. C. Maldonado, and P. C. Masiero. Mutation analysis testing for finite state machines. In *ISSRE*, pages 220–229, 1994.

[36] M. Steffens, S. Oster, M. Lochau, and T. Fogdal. Industrial Evaluation of Pairwise SPL Testing with MoSo-PoLiTe. In *VAMOS*, pages 55–62, 2012.

[37] J. Tretmans. Model based testing with labelled transition systems. *Formal methods and testing*, pages 1–38, 2008.