Abstract Test Case Generation for Behavioural Testing of Software Product Lines

Xavier Devroey, Gilles Perrouin* and Pierre-Yves Schobbens
PReCISE Research Center, Faculty of Computer Science,
University of Namur, Belgium
xavier.devroey@unamur.be, gilles.perrouin@unamur.be, pierre-yves.schobbens@unamur.be

ABSTRACT

In Model Based Testing (MBT), test cases are generated automatically from a partial representation of expected behaviour of the System Under Test (SUT) (i.e., the model). For most industrial systems, it is impossible to generate all the possible test cases from the model. The test engineer recourse to generation algorithms that maximize a given coverage criterion, a metric indicating the percentage of possible behaviours of the SUT covered by the test cases. Our previous work redefined classical Transition Systems (TSs) criteria for SPLs, using Featured Transition Systems (FTSs), a mathematical structure to compactly represent the behaviour of a SPL, as model for test case generation. In this paper, we provide one *all-states* coverage driven generation algorithm and discuss its scalability and efficiency with respect to random generation. All-states and random generation are compared on fault-seeded FTSs.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Reliability, Verification

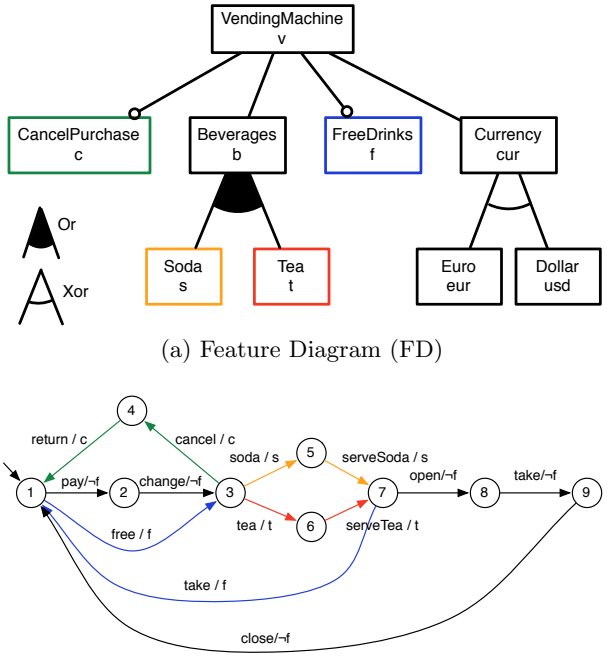
Keywords

Test Case Generation, Software Product Line, Model-Based Testing

1. INTRODUCTION

Software Product Line (SPL) engineering [29] is a branch of software engineering concerned about how to manage variability for a set of software (i.e., products) sharing common assets. Somme assets commons to all the products, some specific to a subset of products. Those assets are regrouped in features. Features are commonly organized in a *Feature Diagram* (FD) [19] which represents all the possible products of the SPL by expressing relationships and constraints between features. For instance, Fig. 1a presents the FD of a soda vending machine. As for any software engineering paradigm, providing efficient Quality Assurance (QA) (e.g. model-checking and testing) techniques is essential to SPL engineering success. Devising an approach to SPLs QA requires to deal with the combinatorial explosion problem as the number of products to consider for validation is growing exponentially with the number of features. In the worst case, no more than 270 features are needed to derive as many products as there are atoms in the universe. SPL testing has been identified as a research area for more than a decade [24] but only gained momentum recently [13]. Despite advances made in handling individual products testing and tests reuse, there is still a lack of strategies to select test cases at the SPL level [22].

In a Model-Based Testing approach for single system, the



(a) Feature Diagram (FD)

(b) Featured Transition System (FTS)

Figure 1: The soda vending machine example [7]

*FNRS Postdoctoral Researcher

SPLC'14 , September 15 - 19, Florence, Italy

http://dx.doi.org/10.1145/2647908.2655971.

test engineer designs a models of the System Under Test (SUT) (or a subset of the SUT) with the expected behaviour of the system he wants to test [33]. Given some coverage criterion, an adequacy measure, the test engineer may use a test case generation tool in order to generate a set of test cases (i.e., a test suite) which will satisfy the coverage criterion. For instance, the all-state coverage criterion for a state machine model requires a test suite which, when all test cases have been executed, will pass trough every state of the model. MBT has been adapted to SPL in order to provide efficient techniques to select relevant products to test [16, 27]. In this paper, we are focused on the behavioural aspect of SPLs.

In our previous work [12], we introduce coverage criteria for Featured Transition Systems (FTSs), a mathematical and compact representation of the behaviour of a SPL. FTSs have originally been proposed to efficiently perform model checking on SPLs [7]. We suggest to build a general framework to perform behavioural model-based testing of SPLs using FTSs as a common representation of the behaviour of the SPL [10]. We are confident that the advances made by the model checking community may be beneficial to perform testing activities (e.g., test case selection) at the SPL level.

In this paper, we present an algorithm to build a test suite satisfying the all-states coverage criterion at the SPL level. We discuss its scalability and efficiency with respect to random generation. All-states and random generation are compared on fault-seeded FTSs.

The remainder of this paper is organised as follows: Section 2 introduces our FTS approach for modelling software product lines, as well as coverage criterion exercised in this paper. Section 3 details our coverage generation algorithms and Section 4 compares them with respect to randomly injected faults. Section 5 discusses related work while Section 6 wraps up with conclusions.

2. BACKGROUND

Classen et al. [7] propose Featured Transition Systems (FTSs) to compactly represent the behaviour of a SPL. Basically a FTS corresponds to a Transition System (TS) with transitions tagged with feature expressions defining which product(s) of the SPL may fire a transition. Fig. 1b presents a FTS for the soda vending machine SPL described in Fig. 1a. For instance: f in Fig. 1b indicates that only products that have the $free$ feature may fire the $free$ and $take$ blue transitions. Formally, an FTS is a tuple $(S, Act, trans, i, d, \gamma)$, where

- S is a set of states;
- Act a set of actions;
- $trans \subseteq S \times Act \times S$ is the transition relation (with $(s_1, \alpha, s_2) \in trans$ sometimes noted $s_1 \xrightarrow{\alpha} s_2$);
- $i \in S$ is the initial state;
- d is a FD; and $\gamma : trans \rightarrow [\![d]\!] \rightarrow \{\top, \bot\}$ is a total function labelling each transition with a boolean expression over the features, which specifies the products that can execute the transition ($[\![d]\!]$ corresponds to the semantic of the FD d, i.e., all the different products that may be derived from d). To improve readability, this expression (called feature expression) for a particular transition is represented using classical boolean operators over features. E.g., the feature expression for the transition $t = 1 \xrightarrow{pay} 2$ in Fig. 1b (noted γt) is $\neg f$, denoting all the products of the SPL that do not have the free feature.

A model checker for FTSs has been implemented in ProVeLine [9], a product line of model checkers for the verification of behavioural models of SPLs.

2.1 Abstract Test Case

To select relevant test cases, we ha to define the notion of abstract test case in a FTS [12]. In opposition to "abstract test case", the notion of "concrete test case" refers to code to execute or actions to test the final product, it corresponds to an instantiated test case for a particular product. Abstract test cases are defined at the *SPL level*. An abstract test case is a trace $atc = (\alpha_1, \ldots, \alpha_n)$ (i.e., finite sequence of actions) in the FTS. It is valid iff:

$$fts \overset{(\alpha_1,\ldots,\alpha_n)}{\Longrightarrow}$$

Where $fts \overset{(\alpha_1,\ldots,\alpha_n)}{\Longrightarrow}$ is equivalent to $i \overset{(\alpha_1,\ldots,\alpha_n)}{\Longrightarrow}$, meaning that there exists a state $s_k \in S$ with sequence of transitions labelled $(\alpha_1, \ldots, \alpha_n)$ from i to s_k. This definition is similar to classical test case definitions for TS test models [23]. However, for a FTS, it is possible to extract sequences of actions that cannot be executed by any product of the SPL (e.g., in Fig. 1b the related transitions contains mutually exclusive features: $pay, change, tea, serveTea, take$). A test case is *executable* if there exists at least one product in the product line able to execute it.

2.2 Coverage Criteria

In our previous work [12], we redefine classical TS coverage criteria for FTSs. A coverage criterion is an adequacy measure to qualify if the test objective defined by the test engineer is reached when executing a test suite on a SUT. In classical behavioural MBT approaches most commonly used selection criteria are structural criteria: state, transition, transition-pair and path coverage [23, 33]. The state (respectively transition) coverage criterion specify that when executing a test suite on the SUT, all the states (respectively transitions) of the test model are visited (respectively fired) at least once. The transition-pair coverage specifies that for each state, all the incoming-outgoing transitions pairs are fired at least once. The path coverage criterion specifies that each path in the test model has to be executed at least once. In order to adapt the test selection problem to FTSs, we define a coverage criterion as a function that associates an FTS and an abstract test suite over this FTS to a real value between 0 and 1 giving the adequacy of the test suite.

In the remainder of this paper, we present an algorithm generating executable abstract test suites satisfying the all-states coverage criterion and compare this suite to randomly generated abstract test suite using fault seeding [23]. The all-states coverage criterion relates to the ratio between the states visited by the test cases pertaining to the abstract test suite and all the states of the FTS. When the value of the function equals to 1, the abstract test suite satisfies **all-states coverage**. On the soda vending machine in Fig. 1b, an all-states covering abstract test suite might be:

$\{(pay, change, soda, serveSoda, open, take, close)$
$(free, tea, serveTea, take); (free, cancel, return)\}$

```
Data: FTS (in) ; A (out)
Result: Accessibility matrix A
∀s_i, s_j ∈ S : A[i,j] ← ⋁_{t=(s_i,α,s_j)∈trans} γt ;
for k ← 1 to #S do
    for i ← 1 to #S do
        for j ← 1 to #S do
            A[i,j] ← A[i,j] ∨ (A[i,k] ∧ A[k,j]);
        end
    end
end
return A;
```

Algorithm 1: Modified Warshall algorithm [30]

```
Data: k (in) ; A (in) ; e (in) ; score (out)
Result: The score for the state on line k in A
score ← 0;
for j ← 1 to #S do
    if s_j ∈ toVisit and SAT(A[k,j] ∧ d ∧ e) then
        score ← score + 1;
    end
end
return score;
```

Algorithm 2: Score computation

```
Data: FTS (in) ; A (in) ; testsuite (out)
Result: Test suite satisfying the all-states coverage
        criterion
toVisit ← S;
candidates ← ⋃_{t=(i,α,s_k)∈trans}((α), γt, score(k, γt));
testsuite ← ∅;
while toVisit ≠ ∅ do
    c ← (atc, e, score) ∈ candidates such as score is
      maximal in candidates;
    candidates ← candidates \ {c};
    if last(c.atc) = i then
        if c.ts contains states from toVisit then
            testsuite ← testset ∪ {c.atc};
            toVisit ← toVisit \ {visited(c.atc)};
        end
    else
        forall t = (last(c.atc), α, s_k) ∈ trans do
            if SAT(d ∧ c.e ∧ γt) then
                candidates ← candidates ∪ {(c.atc +
                  +(α), (e ∧ γt), score(k, (e ∧ γt)))};
            end
        end
    end
end
return testset;
```

Algorithm 3: All-states test suite generation algorithm

3. FTS TEST-CASES GENERATION

The following sections presents our all-states covering and random generation algorithms.

3.1 All-States Generation Algorithm

To compute a test suite satisfying the all-states coverage criterion, we define a branch and bound algorithm with a heuristic based on the accessibility matrix for the FTS. The idea is, at each iteration of the algorithm, to branch out the current partial abstract test case into multiple partial abstract test cases by adding to each one the possible next transitions to visit. And bound up to the partial abstract test case with the highest score, i.e., where the last state may lead to the highest number of states that has not yet been visited by a previously computed abstract test case. We present hereafter three algorithms involved in the computation of a all-states covering test suite: the computation of the *accessibility matrix* for a FTS using a variant of the Warshall algorithm [30], the *heuristic* which computes for a given state its score and the *branch and bound algorithm* which computes the test suite satisfying the all-states coverage criterion.

Accessibility matrix computation. An accessibility matrix A gives for two states (s_1, s_2) the products able to execute a paths from s_1 to s_2. This matrix corresponds to the transitive closure of the FTS and is computed using the Warshall algorithm [30]. Contrary to an accessibility matrix computed for a classical TS, the entry for s_1 and s_2 (noted $A[1,2]$) will not be $true$ or $false$ (i.e., there exists a path from s_1 to s_2 or not), but rather the products for which there exists a path from s_1 to s_2. In our implementation of the algorithm (as for in ProVeLines [9]), the products able to execute a transition t (noted $\gamma\ t$) are represented using feature expressions (i.e., boolean expressions over features). An entry of the accessibility matrix A will be a feature expression. E.g., for the soda vending machine in figure 1b, the entry $A[1,4] = (\neg f \vee f) \wedge c$, states that there exists a path in the FTS from s_1 to s_4 for all the products of the SPL having the *cancel* feature and having or not $free$ drinks. Algo. 1 presents the adaptation of the Warshall algorithm for FTSs. The output of the algorithm is the accessibility matrix A for the given FTS. First A is initialised with the feature expressions conditioning the transition from one state to another. In the next steps, the matrix is updated by computing all the possible paths for each pair of states.

Score computation. Once we have the accessibility matrix, we use a branch and bound algorithm which will explore the FTS according to our heuristic. We choose a simple heuristic: it computes a score equal for a given state to the number of states not yet covered by actual test suite and accessible from this state. Algo. 2 presents the score computation for a given accessibility matrix A and a state s_k. This score is computed dynamically during the generation of an abstract test case by iterating over the k-th line of the accessibility matrix A. The score is incremented by 1 for every cell corresponding to a not yet reached state. Before the increment, we verify that the feature expression is compatible with the FD d and the actual partial abstract test case feature expression e (i.e., there exist one product able to execute the partial abstract test case).

All-states covering abstract test suite computation. The branch and bound algorithm is described in Algo. 3. This algorithm produces an abstract test suite that satisfy the all-states coverage criterion. First the states to visit ($toVisit$) is initialised to S, all the states of the FTS. The candidates abstract test cases to consider ($candidates$) are the test cases with one transition going out from the initial state i. Each candidate is a triplet with an abstract test case (atc), the abstract test case feature expression (e) and a score corresponding to the number of states reachable by the last state of the partial abstract test case ($score$). At this

Data: FTS (in) ; *tescase* (out)
Result: A random executable abstract test case in FTS
$atc \leftarrow random(\alpha)$ such as $t = (i, \alpha, s_k) \in trans$;
$e \leftarrow \gamma t$;
while $last(atc) \neq i$ **do**
  $atc \leftarrow atc + +random(\alpha)$ such as $t = (last(atc), \alpha, s_k) \in trans$;
  $e \leftarrow e \wedge \gamma t$;
  if $last(atc) = i \wedge \neg SAT(d \wedge e)$ **then**
    $atc \leftarrow ();;$
    $atc \leftarrow random(\alpha)$ such as $t = (i, \alpha, s_k) \in trans$;
    $e \leftarrow \gamma t$;
  end
end
return *atc*;

Algorithm 4: Random test case generation algorithm

stage, the test suite (*testsuite*) is empty. The main loop of the algorithm will compute the abstract test cases and continue as long as there remains states to visit in the FTS. We made the assumption here that the graph formed by the states and the transitions of the FTS is connected. In this loop, the best candidate c (with the highest score) is picked and removed from the list of candidates to consider. If the last state of this candidate ($last(c.atc)$) is the initial sate i, the abstract test case is considered as complete and added to the test suite (if it contains states not yet visited). The states reached by the abstract test case ($visited(c.atc)$) are removed from the states to visit and the algorithm will pick the next candidate at the next iteration. If the last state of the abstract test case is not the initial state, the exploration continues and new candidates are computed. The outgoing transitions of the final states are added to the partial abstract test case (if there exists a product able to execute the partial abstract test case) and for each one, a new score is computed.

Simplification for large models. In order to scale to our largest model, a simplification has been implemented in the algorithm: we ignore the feature expressions and check the validity of the abstract test case only before adding it to the test suite. Before adding the abstract test case to the test suite and removing the visited states from the $toVisit$ set, we perform a satisfiability call (SAT) on the conjunction of the FD (d) and the feature expression of the abstract test case ($atc.e$). This simplification reduces the number of SAT calls which are very costly. This allows to reduce the time from more than two days (after which we killed the algorithm without any results) to around 48 seconds.

3.2 Random Generation Algorithm

The random generation algorithm generates random executable abstract test cases (atc). Meaning that test cases may be executed by at least one valid product of the product line. Algo. 4 produces for a given FTS an executable abstract test case. It differs from a pure random algorithm by only returning test cases that may be executed by at least one product of the SPL. This verification is done once the last state of the test case $last(atc)$ is equal to the initial state i. If the conjunction of the FD d and the feature expression of the test case is not satisfiable ($\neg SAT(d \wedge e)$), the test case is cleared and the algorithm loops again. To build a random test suite, one will call the Algo. 4 multiple times.

Table 1: Models characteristics

Model	States	Transitions	Actions	Features
Soda V. M.	9	13	12	9
Minepump	25	41	23	9
Claroline	106	2055	106	44

4. CASE STUDIES

To assess the all-states generation algorithm described in section 3.1, we use fault seeding and compare the number faults detected by the generated test suite to the number of faults detected by randomly generated test suites. Our evaluation has been performed on three case studies. For each of them, we generated random test suites and a test suite satisfying the all-states coverage criterion. The first case is the soda vending machine presented in Fig. 1. The second case is the minepump system [5]. It models the behaviour of a SPL of pumps to be used in a mine that has to be kept safe from flooding and avoid explosions. The third system is the Claroline system [11]. Claroline is an on line course management system dedicated to information sharing between students, professors and teaching assistants. The FTS of the Claroline system has been reverse engineered using statistical web application testing technique [31] from a 5.26 Go Apache web log (which represents 12.689.030 HTTP requests) of the local instance of Claroline used at the University of Namur[1]. Each state corresponds to a web page (without parameters) and each transition corresponds to a link from one page to another. Since we are in a web environment, every page is directly accessible from the initial state and the initial state is directly accessible from every state. It represents the fact that we may directly access a page (using its URL) and leave the website at any moment.

Characteristics of the three considered models are presented in table 1. The algorithms presented in sections 3.1 and 3.2 have been implemented in Java and run on a Windows 7 machine with an Intel Core i3 (3.10GHz) processor and 4GB of memory. We generate random test suites with the same number of test cases as the ones generated by all-states algorithm to enable direct comparison of coverage. We also randomly generate larger test suites to figure out coverage gains. For the soda vending machine, the average generation time of a test set of 20 random test cases was $0,485$ sec. and $0,383$ sec. for the generation of a test suite with 5 test cases satisfying the all-states coverage criterion using the algorithm presented in section 3.1. For the minepump model, the average generation time of 20 random test cases was $0,374$ sec. and $0,432$ sec. for 12 test cases satisfying the all-states criterion. As expected, the average generation time for Claroline was longer: $47,972$ sec. for a test set with 105 test cases satisfying the all-states criterion. Claroline's random test case generation took on average $1,361$ sec. for 200 test cases.

4.1 Results and Discussion

Fault seeding is a classical technique to assess and compare test suites coverage [1, 2]. The idea is to inject faults in

[1] `http://webcampus.unamur.be`

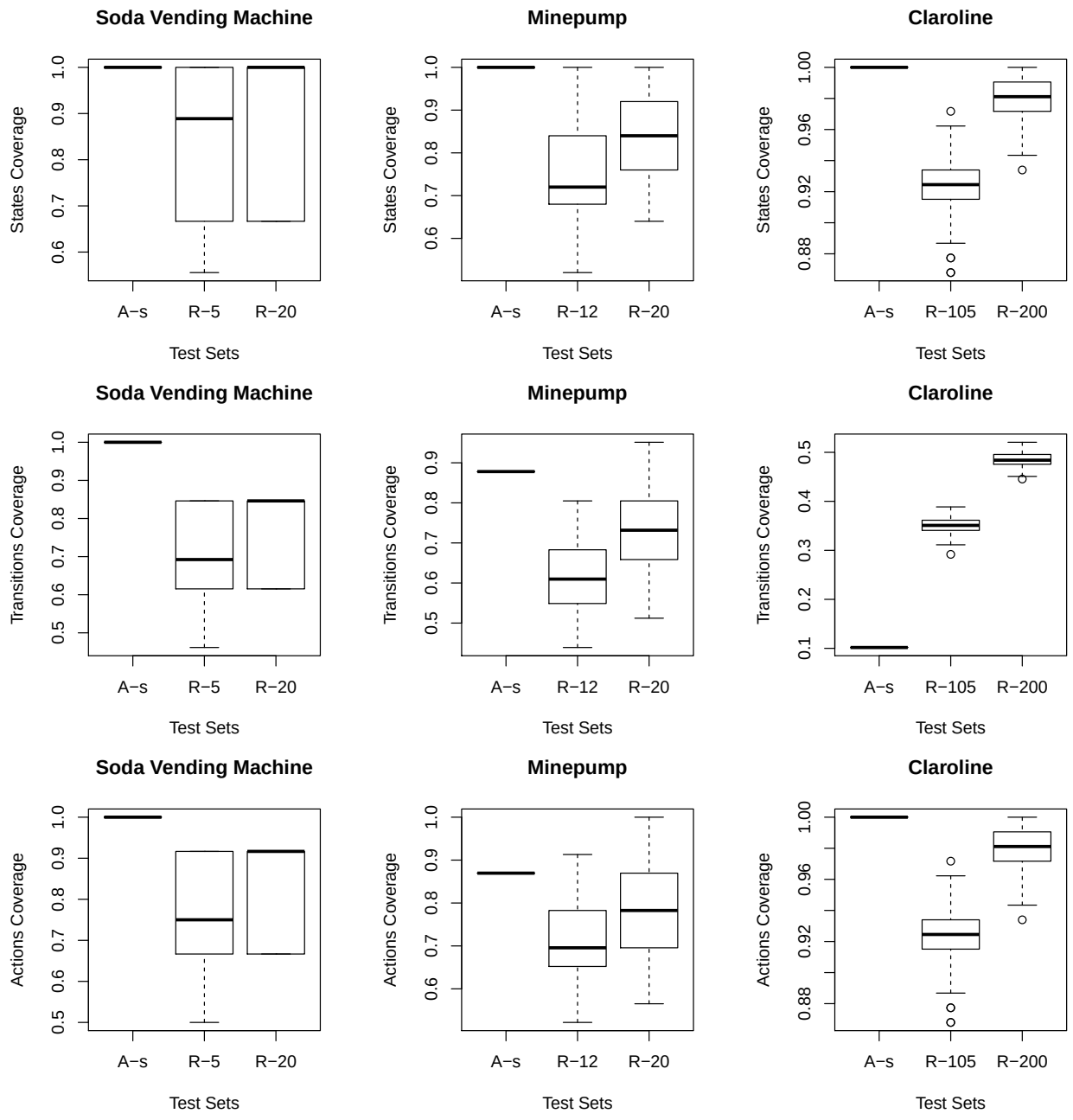


Figure 2: Fault coverage of the soda vending machine, minepump and Claroline systems

Table 2: Average number of faults

Model	States	Transitions	Actions
Soda V. M.	6,05	7,91	7,02
Minepump	16,39	24,41	11,8
Claroline	67,05	1151,06	39,5

SUT and measure the number of faults detected by the test suite. In a SPL context, fault injection has been applied to feature diagrams by Henard et al. [15]. We did not consider fault injection in the FD, in place we choose to artificially inject faults into the FTS by tagging state, transitions and actions as faulty. The fault seeding has been applied 100 times for each FTS model and each test set has been run on those 100 faulty FTSs. The average number of faults for state, transitions and actions for the different models are presented in table 2. The fault coverages presented in figure 2 is the ratio between the number of detected faults and the total number of injected faults. A fault is considered

Table 3: Faults Coverage

Model	Min.	Median	Mean	Max.
Faulty states coverage				
Soda V. M. (a-s)	1.0000	1.0000	1.0000	1.0000
Soda V. M. (r-5)	0.5556	0.8889	0.8167	1.0000
Soda V. M. (r-20)	0.6667	1.0000	0.8533	1.0000
Minepump (a-s)	1.0000	1.0000	1.0000	1.0000
Minepump (r-12)	0.5200	0.7200	0.8533	1.0000
Minepump (r-20)	0.6400	0.8400	0.8440	1.0000
Claroline (a-s)	1.0000	1.0000	1.0000	1.0000
Claroline (r-105)	0.8679	0.9245	0.9224	0.9717
Claroline (r-200)	0.9340	0.9811	0.9792	1.0000
Faulty transitions coverage				
Soda V. M. (a-s)	1.0000	1.0000	1.0000	1.0000
Soda V. M. (r-5)	0.4615	0.6923	0.6938	0.8462
Soda V. M. (r-20)	0.6154	0.8462	0.7446	0.8462
Minepump (a-s)	1.0000	1.0000	1.0000	1.0000
Minepump (r-12)	0.5200	0.7200	0.8533	1 .0000
Minepump (r-20)	0.5122	0.7317	0.7334	0.9512
Claroline (a-s)	0.1017	0.1017	0.1017	0.1017
Claroline (r-105)	0.2918	0.3509	0.3502	0.3886
Claroline (r-200)	0.4455	0.4839	0.4845	0.5205
Faulty actions coverage				
Soda V. M. (a-s)	1.0000	1.0000	1.0000	1.0000
Soda V. M. (r-5)	0.5000	0.7500	0.7517	0.9167
Soda V. M. (r-20)	0.6667	0.9167	0.8067	0.9167
Minepump (a-s)	0.8695	0.8695	0.8695	0.8695
Minepump (r-12)	0.5217	0.6957	0.7096	0.9130
Minepump (r-20)	0.5652	0.7826	0.7883	1.0000
Claroline (a-s)	1.0000	1.0000	1.0000	1.0000
Claroline (r-105)	0.8679	0.9245	0.9224	0.9717
Claroline (r-200)	0.9340	0.9811	0.9792	1.0000

as detected as soon as there is one test case that covers the faulty transition, action or state.

By construction, abstract test suites generated using our all-states algorithm (*a-s* in Fig. 2 and Tab. 3) find all the faulty states for the three FTSs. On average, the random algorithm does not perform as well to cover all the states of the different models. On the contrary, the random algorithm performs better at detecting faulty transition on the largest model (Claroline): an average of 0.4845 for the 200 randomly generated test suites (*r-200* in Fig. 2 and Tab. 3) against 0.1017 for the all-states generation algorithm. We investigated this difference and found that the all-states generated test suite contains only short test cases (2 actions). This is due to our heuristic. Since it prefers states that have a path to uncovered states, the initial state has the highest score in the Claroline model (because all the states in the models have transitions coming from and going to the initial state, due to the web nature of the application). Changing the heuristic to avoid direct return to the initial state may improve the results for this kind of models (where each state is strongly connected to the initial state) but may increase the complexity of the algorithm and generate inadequate abstract test cases for other kinds of models. These results are in line with the fact that all-states coverage criterion is poor to cover transitions. Of courses an all-transitions algorithm would have given much better results (and all-states coverage). However our preliminary evaluation of such an algorithm resulted in huge scalability problems for the Claroline case study; after more than 3 days of generation and a text file describing the test suite of more 250 GB, our systems ran out of memory (and also of hard drive space!). Exhaustive computation of all-transitions for moderate size FTS is therefore not an option. Finally, we observe that the test suite generate to cover all-states in the Claroline FTS also covers all the actions. This is due to the nature of the model, since each state represents a page and each action represents a link followed from a page (i.e., a state in the FTS) to another page (i.e., another state), there is as many actions as there is states. Each faulty action is thus detected. This is not always the case, e.g., the soda vending machine in Fig. 1b.

4.2 Threats to validity

Models size and construct validity. Two of the models we used in the case study are home-made small models and may not reflect a real case. To mitigate this risk, we also used the Claroline model [11]. This model has been generated from the Apache log of an existing website used by the students and professors at the university of Namur and reflects a real system.

All-states generation algorithm. The all-states generation algorithm has been simplified to reduce the number of SAT calls which are very costly. This simplification gives good results on our largest model (Claroline) due to the few constraints on the FTS. On other models with more constraints on the different transitions, this simplification may give poor results. We intend to compare the actual implementation which uses a SAT solver with binary decision diagrams (BDDs) which have performed better when processing FTSs [6].

Random generation algorithm. To avoid too many SAT calls, we verify that an abstract test case is executable *a posteriori* by calling the SAT solver once with the conjunction of the FD (represented as a boolean formula) and the feature expression of the transitions of the test case. We repeat the building of an abstract test case while it is not executable. As shown in section 4, the execution time on a large model is good. Since the largest FTS model we considered does not have a lot of behaviours exclusive to subsets of the product line, this implementation of the random algorithm works fast. This may be not the case for other models with a lot of constrained behaviour.

Duplicate random abstract test cases. The random generation of a test suite does not check whether there are duplicates abstract test cases or not. Since the size of the test suites considered in section 4 is larger than the size of the all-states covering test suite, this thread is limited. To avoid this, one may implement a filter to check that newly generated test suites are not duplicated.

Few constraints for Claroline FTS. The Claroline FTS and its FD contain few constraints ending in a SPL with lots of products. We believe this is a typical characteristic of web applications which are a particular class of system. As explained in section 3.2, this has influenced the implementation of the random algorithm in order to minimize the number of SAT calls (which are costly in CPU time). It has also influenced the heuristic during the generation of the test suite covering all-states (see section 4) and gives very short test cases in regard to the size of the system. We plan to apply our algorithms on large industrial systems with more constrained FD and FTSs in order to validate our

conclusions.

Coverage vs effectiveness. As discussed by Inozemtseva et al. [17], a test suite with a good coverage does not guarantee the effectiveness of this test suite. However, in a first attempt to compare our generation algorithms, injected faults coverage seems to be a reasonable approach. We plan to develop our fault injection strategies in order to adapt classical mutation testing [23] to SPLs.

5. RELATED WORK

Coverage testing for SPL targets both *variability* and on *behavioural* models. Many approaches targeting variability models (mostly feature models) exploit ideas of *Combinatorial interaction testing* to sample configurations [28, 8, 18], yielding configuration sets even for very large feature models. The central coverage criterion here is named "t-wise": all t-combinations of features must appear at least once in the generated configurations. It based on the empirical observation that most bugs are related to undesired interactions between features. Such configurations can be further prioritized according to assigned feature weights [16, 18], or similarity [14]. This actually helps testers to scope more finely and flexibly relevant products to test than a covering criteria alone. However the benefits of such techniques in terms of *behavioural coverage* have to be assessed.

At the behavioural level, several techniques have also been proposed. One of those considers incremental testing in the SPL context [34, 26, 21]. For example, Lochau et al. [21] proposed a model-based approach that shifts from one product to another by applying"deltas" to statemachine models. These deltas enable automatic reuse/adaptation of test model and derivation of retest obligations. Oster et al. [26] extend combinatorial interaction testing with the possibility to specify a predefined set of products in the configuration suite to be tested. There are also approaches focused on the SPL code by building variability-aware interpreters for various languages [20]. Based on symbolic execution techniques such interpreters are able to run a very large set of products with respect to one given test case [25]. Cichos et al. [4] use the notion of 150% test model (i.e., a test model of the behaviour of a product line) and test goal to derive test cases for a product line but do not redefine coverage criteria at the SPL level. Finally, Beohar et al. [3] propose to adapt the *ioco* framework proposed by Tretmans [32] to FTSs. Contrary to this approach, we do not seek exhaustive testing of an implementation but rather to select relevant abstract test cases based on the criteria provided by the test engineer.

6. CONCLUSION

In this paper we present a all-states coverage driven executable abstract test suite generation algorithm and evaluate it against randomly generated executable abstract test suites using fault seeding. This generates an executable abstract test suites satisfying the all-states coverage criteria, meaning that when executing all the tests of this test suite, all the states of the FTS are visited at least once. We compare the results of the generated abstract test suites on three case models, two small models of hardware systems and one larger model of a web application (Claroline), using fault seeding. The all-states generation algorithm produces test cases able to detect all faulty states (as expected) but also able to detect a large amount of faulty actions. However, due to the particular web nature of the application in the Claroline case study, the test cases do not cover faulty transitions very well. The random generation algorithm performed much better in this case producing longer test cases thus covering more faulty transitions. Random generation is also faster (35 times speed-up on average). Future works involve improvements on our all-states generation algorithm and the of a scalable transitions coverage approach. As all-transitions coverage is tricky to compute for moderate size systems, we may employ heuristics to "mimic" this coverage without explicitly computing it, adopting a similar strategy to the on we proposed for t-wise testing over feature models [14]. Our long term goal is to provide product line of practical algorithms for FTS coverage test case generation.

7. REFERENCES

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? [software testing]. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 402–411. IEEe, 2005.

[2] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *Software Engineering, IEEE Transactions on*, 32(8):608–624, 2006.

[3] H. Beohar and M. R. Mousavi. Spinal Test Suites for Software Product Lines. *ArXiv e-prints*, 2014.

[4] H. Cichos, S. Oster, M. Lochau, and A. Schürr. Model-based Coverage-driven Test Suite Generation for Software Product Lines. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS '11, pages 425–439. Springer, 2011.

[5] A. Classen. Modelling with fts: a collection of illustrative examples. Technical report, University of Namur (FUNDP), 2010.

[6] A. Classen. *Modelling and Model Checking Variability-Intensive Systems*. PhD thesis, University of Namur (FUNDP), 2011.

[7] 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 Transactions on Software Engineering*, 39(8):1069–1089, Aug. 2013.

[8] M. Cohen, M. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA '07*, pages 129–139, 2007.

[9] M. Cordy, A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. ProVeLines: A Product Line of Verifiers for Software Product Lines. In *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*, SPLC '13 Workshops, pages 141–146, New York, NY, USA, 2013. ACM.

[10] X. Devroey, M. Cordy, G. Perrouin, E.-Y. Kang, P.-Y. Schobbens, P. Heymans, A. Legay, and B. Baudry. A Vision for Behavioural Model-Driven Validation of Software Product Lines. In Margaria T., Steffen B., and Merten M., editors, *ISoLA 2012, Part I*, LNCS 7609, pages 208–222, Crete, 2012. Springer-Verlag Berlin Heidelberg.

[11] X. Devroey, G. Perrouin, M. Cordy, P.-y. Schobbens, A. Legay, and P. Heymans. Towards Statistical Prioritization for Software Product Lines Testing. In A. Wasowski and T. Weyer, editors, *Proceedings of the 8th International Workshop on Variability Modelling of Software-intensive Systems*, Nice, France, 2014. ACM.

[12] X. Devroey, G. Perrouin, A. Legay, M. Cordy, P.-y. Schobbens, and P. Heymans. Coverage Criteria for Behavioural Testing of Software Product Lines. In T. Margaria and B. Steffen, editors, *Proceedings of the 6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (to appear)*, ISoLa'14. Springer, 2014.

[13] E. Engström and P. Runeson. Software product line testing-a systematic mapping study. *Information and Software Technology*, 2010.

[14] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Transactions on Software Engineering*, To Appear, 2014.

[15] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon. Assessing Software Product Line Testing Via Model-Based Mutation: An Application to Similarity Testing. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 0:188–197, 2013.

[16] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon. Multi-objective Test Generation for Software Product Lines. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 62–71, New York, NY, USA, 2013. ACM.

[17] L. Inozemtseva and R. Holmes. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, 2014. ACM.

[18] M. F. Johansen, Ø. Haugen, F. Fleurey, A. G. Eldegard, and T. Syversen. Generating better partial covering arrays by modeling weights on sub-product lines. In *MoDELS '12*, pages 269–284, 2012.

[19] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, 1990.

[20] C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward variability-aware testing. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*, FOSD '12, pages 1–8. ACM, 2012.

[21] M. Lochau, I. Schaefer, J. Kamischke, and S. Lity. Incremental model-based testing of delta-oriented software product lines. In A. Brucker and J. Julliand, editors, *Tests and Proofs*, volume 7305 of *LNCS*, pages 67–82. Springer, 2012.

[22] I. d. C. Machado, J. D. McGregor, Y. a. C. Cavalcanti, and E. S. de Almeida. On strategies for testing software product lines: A systematic literature review. *Information and Software Technology*, Apr. 2014.

[23] A. P. Mathur. *Foundations of software testing*. Pearson Education, 2008.

[24] J. D. McGregor. Testing a Software Product Line. Technical report, Carnegie-Mellon University, Software Engineering Institute, 2001.

[25] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *36th International Conference on Software Engineering*, ICSE '14. IEEE, 2014.

[26] S. Oster, F. Markert, and P. Ritter. Automated incremental pairwise testing of software product lines. In *Software Product Lines: Going Beyond*, pages 196–210. Springer, 2010.

[27] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. le Traon. Pairwise testing for software product lines: Comparison of two approaches. *Software Quality Journal*, 2011.

[28] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. L. Traon. Pairwise testing for software product lines: comparison of two approaches. *Software Quality Journal*, 20(3-4):605–643, 2012.

[29] K. Pohl, G. Böckle, and F. van der Linden. *Software product line engineering - foundations, principles, and techniques*. Springer, 2005.

[30] K. Rosen. *Discrete Mathematics and Its Applications 7th edition*. McGraw-Hill Science, 2011.

[31] S. E. Sprenkle, L. L. Pollock, and L. M. Simko. Configuring effective navigation models and abstract test cases for web applications by analysing user behaviour. *Software Testing, Verification and Reliability*, 23(6):439–464, 2013.

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

[33] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2007.

[34] E. Uzuncaova, S. Khurshid, and D. Batory. Incremental test generation for software product lines. *Software Engineering, IEEE Transactions on*, 36(3):309–322, 2010.