# An Enhanced Test Case Selection Approach for Model-Based Testing: An Industrial Case Study

Hadi Hemmati[a,b], Lionel Briand[a,b], Andrea Arcuri[a], Shaukat Ali[a,b]
[a] Simula Research Laboratory
[b] Department of Informatics, University of Oslo
{hemmati, briand, arcuri, shaukat} @simula.no

## ABSTRACT

In recent years, Model-Based Testing (MBT) has attracted an increasingly wide interest from industry and academia. MBT allows automatic generation of a large and comprehensive set of test cases from system models (e.g., state machines), which leads to the systematic testing of the system. However, even when using simple test strategies, applying MBT in large industrial systems often leads to generating large sets of test cases that cannot possibly be executed within time and cost constraints. In this situation, test case selection techniques are employed to select a subset from the entire test suite such that the selected subset conforms to available resources while maximizing fault detection. In this paper, we propose a new similarity-based selection technique for state machine-based test case selection, which includes a new similarity function using triggers and guards on transitions of state machines and a Genetic algorithm-based selection algorithm. Applying this technique on an industrial case study, we show that our proposed approach is more effective in detecting real faults than existing alternatives. We also assess the overall benefits of model-based test case selection in our case study by comparing the fault detection rate of the selected subset with the maximum possible fault detection rate of the original test suite.

## Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]

## General Terms

Experimentation and Verification.

## Keywords

Test case selection, model-based testing, similarity-based selection, genetic algorithms.

## 1. INTRODUCTION

Model-Based Testing (MBT) is getting increasing attention both in industry and academia as a test automation approach [1]. The idea is to generate executable test cases by systematically traversing specification models (e.g. represented as UML state machines [2]) based on a test strategy such as a coverage criterion that aims to cover certain features of the model (e.g., all transitions). There are many academic and commercial MBT tools [3] and some studies report on the applicability and cost-effectiveness of MBT [1]. Unfortunately, in practice, more specifically at the integration and system levels, MBT may lead to very large test suites, even for simple coverage criteria. We have observed cases [3] leading to the generation of thousands of test cases for relatively modest industrial case studies with well-known coverage criteria such as all transition-pairs and all round-trip paths [4]. Therefore, in many situations where deadlines are tight, resources limited, the testing cost is high due to the use of hardware-in-the-loop or access to dedicated test infrastructures (e.g., network), executing the entire test suite is not an option. This is typically the case for many embedded and distributed systems. For example, system level testing of a video conferencing system requires establishing connections with other video conferencing systems over the network and streaming audio and video and communicating control data. To test the software of such system, we have to assign enough resources (actual physical devices dedicated for the test) to the test case, which increases the cost of executing each test case compared to running a test case created for testing a local function on a PC. In addition, such test cases should properly handle acceptable delays in the system execution and the network communication, which means that the execution time of each test case can be quite high in such systems.

The goal of selection techniques, given limited resources leading to an estimated maximum test suite size, is to maximize the fault detection rate of the selected subset. In general, this test case selection problem is NP hard (traditional set cover) [5]. Other than random selection, there have been two main types of test case selection heuristics proposed in the literature. The first class of techniques (coverage-based) tries to directly maximize the coverage of the selected subset [6] and the second type (similarity-based), which has recently been getting more interest among researchers, is about minimizing similarity (where its definition varies on different studies) between selected test cases and each selected subset of test cases contains test cases that are less similar to each other [7].

In this paper, we propose a new similarity-based selection technique which is applied on test suites automatically generated from UML state machines. The approach, which does not require any execution information and is applied before executing any test case, first improves the similarity function for model-based test case selection introduced in [7], by using triggers and guards on transitions of UML state machine as a basis of measuring similarity. Second, it improves the selection algorithm by using Genetic Algorithms (GA) [8] instead of a Greedy search. This work, to the best of authors' knowledge, is the first to address similarity-based test case selection for UML-based testing. The selection technique is integrated with a fully automated test case generation tool (TRUST) [3], where the inputs are UML state machines and outputs are the selected executable test cases. The context and objectives of industrial case study can be briefly characterized as follows: (1) our selection technique is applied to an industrial system where MBT was already used for test case generation, (2) there are no seeded faults and all faults are based on actual mistakes made by developers, (3) the size of the test suite is significantly larger than that of previous, similar studies [7, 9] (more than double of their largest test suite), (4) a comparison is performed not only with other similarity-based techniques (even those which are not specific to MBT but are applicable), but also with all other well-known selection techniques (additional coverage-based [10, 11], GA-based coverage [12, 13], and random selection [4]), (5) we provide a thorough discussion on cost analysis, (6) the improvement, in terms of fault detection rate, by our selection technique is compared to using a stricter coverage criterion, and (7) the practical benefits of using our test case selection technique for MBT is investigated by showing that our approach can select a small (approximately 10%) subset of the automatically generated test suite which can find more than 90% of the faults detectable by the entire test suite.

The rest of the paper is organized as follows. The next section reports on background information about test case selection. Section 3 discusses the basic principles regarding GA which are necessary to understand the paper. Section 4 provides a brief overview of related works covering similarity-based selection techniques. Section 5 introduces our approach for test case selection in state machine-based testing, and Section 6 reports the experimentation results of applying the technique on an industrial case study. Section 7 concludes the paper and outlines our future work plan.

## 2. TEST CASE SELECTION

There are several strategies for reducing the number of automatically generated test cases in MBT. One can try using a stricter coverage criterion (i.e., the criterion that results in fewer number of test cases). For instance, if using all transition-pairs [4] generates a far too large test suite, the all-transitions [4] criterion can be adopted instead to decrease the number of test cases, which still achieves systematic testing but may reduce the fault detection rate. However, often this is not a practical solution as one cannot ensure that the number of test cases will be below a required threshold. Test suite reduction can also be useful when the goal is to minimize the test suite by removing redundant test cases with respect to a criterion (e.g. code coverage). In test case selection, given a maximum number of test cases, the goal is to select a subset of the entire test suite that maximizes fault detection. Prioritization techniques, on the other hand, do not remove any test case but order their execution [4, 5, 14], and therefore do not address our problem. As a result, we focus in this paper only on test case selection.

Test case selection is mostly studied in the context of regression testing, where the goal of test case selection is to find a subset of the original test suite that guarantees the execution of fault-revealing test cases [4, 5, 14]. The main differences between model-based test case selection and selection in the context of regression testing are that, in our context, we are not interested in finding the affected parts of the system and we do not have execution information of the test suite as it is the case in regression testing. Therefore, heuristics such as using component meta data [15], model differences [16], and execution traces (e.g. call stack [17]) are not applicable here. In addition, most studies in test case selection (even those which are general purpose and not specific to regression testing) are based on code-level information and do not directly apply to MBT (e.g. code-based dependency analysis [18, 19] and additional statement coverage [10, 11]). Rather, MBT selection heuristics are based only on the characteristics of the (abstract) test cases.

There are three main classes of selection techniques which are introduced for MBT:

1) Random [5] or semi-random selection [4], where there is no guidance to select test cases.

2) Coverage-based selections, where we hypothesize that "*the test cases which have more coverage (such as model-based and requirement-based coverage) are more likely to detect faults*". The idea is inspired from redundant test case removal in test case reduction, where redundant test cases are those which have the same coverage. Note that assessing the coverage of a test case must not necessarily require its execution. For example, transition coverage in a state machine can be determined if traceability has been preserved between a test case and its source state machine. Most coverage-based techniques are re-expressed into optimization problems where the goal is to select the best combination (or permutation in case of prioritization [11]) of test cases to achieve full coverage [20-23]. For example, in [11] a Greedy search selects, at every step, the test case that covers the most uncovered statements whereas in [12, 22] a GA is used to find the maximum coverage.

3) Similarity-based selections, where we hypothesize that "*the more diverse the test cases the higher their fault revealing capacity* [24]". To use this approach one needs a (dis)similarity function to measure the diversity of a subset by averaging all pair-wise similarity values. Code-based similarity functions have been proposed in the literature. However, to the best of authors' knowledge, there is only one model-based similarity function [7], denoted here as Identical Transitions Similarity (*It*). For any two test paths $tp_i$ and $tp_j$, $It(tp_i, tp_j)$ is defined as:

   > "The number of identical transitions (which in UML state machines means: same source states, triggers, and target states) in $tp_i$ and $tp_j$ divided by the average length (number of transitions in the test path) of $tp_i$ and $tp_j$".

   After defining a similarity function, a selection algorithm is required to choose a sample of test cases with the minimum pair-wise similarity among its members.

## 3. GENETIC ALGORITHMS

For a given similarity measure, several alternative selection techniques can be used, such as optimization techniques, Greedy search, and clustering. In this paper we use GA and compare it with Greedy search (which is the only reported similarity-based test case selection algorithm to date in the context of state-based testing and MBT in general [7]) as a baseline. GA is used in this paper since the nature of our problem, which is a form of optimization, resembles typical problems addressed in search-based software engineering [25] where GA is the most used and successful reported technique [25]. A more comprehensive study of selection algorithms will be part of our future work. Though further details on how we have employed GA in a test case selection context will be discussed in Section 4, we provide below minimum background information on GA.

GA rely on four basic features: population, selection, crossover and mutation. More than one solution is considered at the same time (population). At each generation (i.e., at each step of the algorithm), some good solutions in the current population chosen by the selection mechanism generate offspring using the crossover operator. This operator combines parts of the chromosomes (i.e., the solution representation) of the offspring with a certain probability; otherwise it just produces copies of the parents. These new offspring solutions will fill the population of the next generation.

The mutation operator is applied to make small changes in the chromosomes of the offspring. To avoid the possible loss of good solutions, a number of best solutions can be copied directly to the new generation without any modification. Another option is to use a steady state approach, in which only the offspring that are not worse than their parents are added to the next generations. Fitter individuals should have more chances to survive and reproduce. This is represented by the selection mechanism, and there are several variants for it. Eventually, after a number of generations, an individual that solves the addressed problem will be evolved.

## 4. RELATED WORK

In this section, we only review studies on similarity-based test case selection techniques, since we have already discussed about alternative techniques and their limitations for our context in Section 2. Although there exist studies regarding similarity-based selection, minimization, and prioritization on code-based testing, model-based test case selection using a similarity function has not been a focus of study in the literature. However many ideas from code-based selection can be adapted to MBT.

Not surprisingly, most similarity-based techniques have been performed in the context of code-based regression testing and use code coverage or other types of execution information. In [26] the similarity function is based on all def-use pairs coverage and they use a classification algorithm as a selection technique, where they classify similar test cases in one class and distribute their selection over different classes. Basic block coverage in the code (e.g., statement coverage) is a basis for defining similarity functions in [27], [13], and [24, 28]. Greedy search, adaptive random selection, and clustering are used in these studies for selection. In [29] different heuristics are used based on execution information from the original test suite to support regression testing (e.g., memory operations with values from dynamic execution of a test case is used in a similarity function). Ledru et al. [9] have introduced a similarity-based selection technique which can be applied on both code-based and model-based techniques, since it is based on the test scripts and not the source code or a specification model. The basic idea is to analyze the test script as a string and compare each pair of test cases as two strings using edit-distance functions such as Levenshtein [30]. In the current paper, we refer to this similarity function as String-Based Similarity (*Sb*). Using this function Ledru et al. applied a Greedy search to select test cases.

The only similarity-based test case selection technique in MBT is introduced in [7], where sequences of transitions in a Labeled Transition System (LTS) model of the software under test (SUT) are used for representing test paths. The similarity function is *It*, as defined in Section 2, and the selection technique is a Greedy search. This work and the work of Ledru in [9] can be considered as potential baselines of comparison for our study.

Some empirical studies [13, 29] do not use basic random selection as a baseline of comparison. However, it is very important to at least compare any (meta)heuristic-based technique with random selection to show that the improvement, if any, is worth the extra cost which is incurred when using such heuristics. Furthermore, other studies [7, 9, 26] do not have a comparison with coverage-based techniques [10, 11], which may be considered state-of-practice.

## 5. TEST CASE SELECTION BASED ON SIMILARITIES BETWEEN TEST PATHS USING TRIGGERS AND GUARDS

The problem of test case selection in our context can be formalized as:

> "Given a test suite *TS* that detects a set of faults (*F*) in the system, our goal is to maximize $FD(s_n)$, where $s_n$ is a subset of *TS* of size *n* and $FD(s_n)$ is the percentage of *F* which is detected by $s_n$".

Since there is no information about the fault detection rate of each test case without prior execution, a surrogate measure for $FD(s_n)$ is required. In similarity-based selection techniques the assumption is that the more diverse the selected subset, the larger the number of detected faults. Therefore, the problem is reformulated as minimizing $SimMsr(s_n)$:

$$SimMsr(s_n) = \sum_{tp_i\,,\,tp_j \in s_n \,\wedge\, i>j} SimFunc(tp_i\,, tp_j)$$

Where SimFunc($tp_i$ , $tp_j$) returns the similarity of two test paths (abstract test cases in MBT) in $s_n$ represented by $tp_i$ and $tp_j$. According to this definition, we need to define 1) a representation for a test path (*tp*), 2) a similarity function (SimFunc), and 3) a selection algorithm to select the optimal $s_n$.

In MBT finding the best test case representation depends on the type of input model, which in our case is a UML 2.0 state machine. A path on the model (test path) seems to be the best representation, since it is both abstract enough to be used as a similarity function input and rich enough to contain all relevant state-based testing information. Abstract test cases in this notation (called test path) are sequences of states and transitions, identified by their corresponding trigger. If a transition has *k* triggers it will be considered to be *k* transitions from the same source to the same target but with different triggers. A test path can therefore be formalized as follows:

```
<tp> ::= <init> "," <trans>
<trans> ::= <event> "," <target> | <event> "," <target> "," <trans>
<event> ::= <triggerName>|<guardValue>|<Id>|
            <guardValue>","<triggerName>
```

where <*init*> and <*target*> are taken from the set of states and <*triggerName*> and <*guradName*> from the set of triggers and guards on the transitions of the model and <*Id*> is a unique id assigned to transitions which do not have any trigger or guard. If a transition is guarded <*event*> contains <*guardValue*>.

The similarity function that we use in this study is similar to *It* in [7] with a minor but important difference. Because *It* is based on identical transitions, they do not consider two transitions which have the same trigger (same method call or same signal reception) but different source or target state, to be identical. However, our similarity measure (trigger-based similarity, *Tb*) is based on identical triggers. According to this definition of identical triggers, *Tb* is defined as follows:

"*Tb*(*$tp_i$* , $tp_j$) =Number of identical triggers in *$tp_i$* and *$tp_j$* divided by the average length (number of transitions in the test path) of *$tp_i$* and *$tp_j$*".

Since identical triggers are more likely than identical transitions to be present in two test paths, *Tb* can be considered less strict than *It* in assigning similarity to test paths. As a result, *Tb* might be more effective in cases where there are identical triggers in different transitions in the state machine, which is a common situation. *Tb* tends to distinguish similarity among transitions in a more gradual fashion. For example, let us assume *$tp_1$* = <1,a,2,b,3>, *$tp_2$* = <1,c,4,b,3>, and *$tp_3$* = <1,d,5,e,6>, where numbers are state identifiers and characters are trigger names (and there is no guard). Note that *$tp_3$* is completely different than the two others (no similarities except for the initial state "1"). Though similarity-based test case selection seeks to keep the selected test cases as diverse as possible, *It* cannot detect any similarity between any pair of test paths as there is no identical transitions among *$tp_1$*, *$tp_2$*, and *$tp_3$*. However, *Tb*(*$tp_1$* , *$tp_2$*) = 0.5 since there is one identical trigger "b" and the average length of the two test paths is two. Therefore, if we want to select two test paths out of the three, *It* selects randomly (since all similarity values are zero), but *Tb* will choose one of *$tp_1$* or *$tp_2$* to discard, thus achieving more diversity among remaining test cases.

In this paper, we use a steady state GA as a selection technique. An individual (i.e., a solution to the problem) is *$s_n$* (subset of *TS* with size *n*). Given a similarity function SimFunc(*$tp_i$* , *$tp_j$*), the fitness function *f* to minimize is the sum of SimFunc(*$tp_i$* , *$tp_j$*) for each pair of (*$tp_i$* , *$tp_j$*) in *TS* ($SimMsr(s_n)$). The selection mechanism is the rank selection with bias 1.5 which has been shown to work well [31]. The population size is fixed to 50. We use a single point crossover to combine two different parents $s_n^x$ and $s_n^y$. A random position *r* such that 0<*r*<*n* is chosen. All test paths in $s_n^x$ from position *r* and onward are swapped with the values in the same positions in $s_n^y$. Crossover is applied with probability $P_{xover}$ (0.75 in our experiments) with probability 1-$P_{xover}$, the offspring are just be copies of their parents. For example, if $s_4^1$ and $s_4^2$ are two individuals of the population in iteration *i*, and r = 2, there is a probability of 0.75 that they will be replaced by $\acute{s}_4^1$ and $\acute{s}_4^2$ in iteration *i*+1, where:

$$s_4^1 =< tp_1 , tp_2 , tp_3 , tp_4 > \textit{and } s_4^2 =< tp_a , tp_b , tp_c , tp_d >$$

$$\acute{s}_4^1 =< tp_1 , tp_2 , tp_c , tp_d > \textit{and } \acute{s}_4^2 =< tp_a , tp_b , tp_3 , tp_4 >$$

The selected mutation operator is similar to what is typically used for bit strings. Each test path in *$s_n$* is mutated with probability 1/*n*. A mutated test path is replaced by a test path that is selected at random from the set of all possible test paths. For example if $s_4^x =< tp_1 , tp_2 , tp_c , tp_d >$ and $tp_5 \in TS$ then $\acute{s}_4^y =< tp_1 , tp_2 , tp_5 , tp_d >$ can be a mutated version of $s_4^x$. The pseudo-code of the employed GA is defined as follows:

```
Sample a population G of m test cases uniformly from the search space (i.e., the set of all possible valid sets with a given size n)
Repeat until the specified time is expired
    Choose $s_n^x$ and $s_n^y$ from G
    $\left(\acute{s}_n^x , \acute{s}_n^y\right) := crossover\ (s_n^x , s_n^y , P_{xover})$
    Mutate($\acute{s}_n^x , \acute{s}_n^y$)
    If valid ($\acute{s}_n^x , \acute{s}_n^y$) $\wedge$ min ($f(\acute{s}_n^x), f(\acute{s}_n^y)) \le$ min ( $f(s_n^x), f(s_n^y)$)
    Then $s_n^x := \acute{s}_n^x$ and $s_n^y := \acute{s}_n^y$
```

Notice that we only accept "valid" solutions. A solution *$s_n$* is valid if all the test paths in *$s_n$* are unique. The first randomly generated population is forced to contain only unique test paths. However, search operators such as crossover and mutation can produce new offspring that have repeated test paths in them. There are several ways to handle constraints in evolutionary algorithms. One way is to design search operators that always produce valid individuals. In this paper we simply discard the offspring that are not valid. The smaller the sample (test suite) size, the lower the probability of such occurrences. Since we focus here on small samples of test cases, this seems as a more suitable strategy in our context. For example, in our case study experiments, while sampling less than 2% of the total test suite, the probability of generating an invalid individual in one GA run is less than 3%. Increasing the sample size to 30% of the test suite increases this probability up to 30%. However, even with a 30% chance of invalid individual generation, given the fixed short time for one run of the GA, it is still more effective than its baseline of comparison (Greedy search) in detecting faults. Note that this probability also depends on the stopping criterion, since if we let the GA run longer, the diversity in the GA population decreases, which then results in less invalid individuals generated by the crossover operator.

We have applied three types of stopping criteria for GA in this study: (1) stopping after specific number of iterations, (2) stopping after a fixed period of time (e.g., 1sec), and (3) letting the GA run for some time (e.g., 1sec) and then stop only if there is no improvement over a specified period of time (e.g., 200 milliseconds).

## 6. EMPIRICAL EVALUATION

In this section, we assess the effectiveness of the proposed approach by applying it on an industrial case study. In addition, we evaluate its fault detection rate (referred below as *FDR*) by comparing it to other alternatives already reported in the literature. Information about the case study is sanitized due to confidentiality restrictions.

### 6.1 Case study description

The SUT is a safety monitoring component in a safety-critical control system implemented in C++. We chose this system because it exhibits a complex state-based behavior that is modeled as UML state machines complemented by constraints specifying state invariants and guards, which are useful to derive automated test oracles. This SUT is typical of a broad category of reactive systems interacting with sensors and actuators. The first version of the system (including models and code) was developed and verified by company experts and our research team. The 26 faults used in the study were introduced during maintenance activities of subsequent versions of the SUT by developers and re-introduced for the purpose of the experiment in the latest version of the SUT.

The correct and most up-to-date UML state machine, representing the latest version of the SUT's behavior, consists of one orthogonal state with two regions. Enclosed in the first region are two simple states and two simple-composite states. The simple-composite states contain two and three simple states. The second region encloses one simple state and four simple-composite states that again consist of, respectively, two, two, two, and three simple states. This adds up to one orthogonal state, 17 simple states, six simple-composite states, and a maximum hierarchy level of two. The unflattened state machine contains 61 transitions and the flattened state machine consists of 70 simple states and 349 transitions.

Among the 26 faults, 11 of them were sneak paths (illegal transitions in the modified model) [4]. To detect such faults the model should account for the behavior of the SUT when receiving unexpected triggers. Such robustness behavior is not currently modeled and therefore, these 11 faults could not be caught by any test case generated from the model. The remaining 15 faults (detectable by the test cases generated from the model) are collected and 15 faulty versions of the code (mutant programs) are made by introducing one fault per program. Each of these faults belongs to one of the following categories: wrong guards on transitions, wrong state invariant, missing transition, and wrong OnEntry action of states. The purpose was to study each real fault in isolation in order to avoid masking effects and compute fault detection scores. Since a test case stops executing after detecting the first failure, in a program with multiple faults we should either rerun test cases on the SUT after each bug fix, or isolate faults by seeding one fault per mutant program. We chose the latter case to avoid manual bug fixing after each run. Our approach should not be confused with mutation testing which makes use of mutation operators to create faults and then seed them in the SUT one by one. In our approach, all faults were real faults, as described above.

In the next step, the correct UML state machine is given to our test case generation tool [3] as an input model and executable test cases were automatically generated. Note that our selection technique is based on similarities between test paths (abstract test cases without test data). In general different faults can be detected by the same test path instantiated with different test data. Therefore, it is necessary to run the selected test paths with different input data and compare the *FDR* distribution of the test paths selected by different techniques. However, in our case study if a test path has the ability to detect a fault, it can be detected by any valid test data for that test path. Therefore, in our experiment, we have one test case per test path and the *FDR* of a test path is equal to the *FDR* of the corresponding test case.

## 6.2 Experiment design

To evaluate our selection technique we formulated the following four research questions:

**RQ1.** Which similarity measure is more effective for UML state machine-based test case selection, in terms of *FDR*?

**RQ2.** Is using GA for test case selection significantly more cost-effective (in terms of time spent to find a solution) compared to a Greedy search?

**RQ3.** Are similarity-based selection techniques more effective than coverage-based and random selection techniques?

**RQ4.** In the context of MBT, what is the practical benefit of test case selection, on a representative industrial case study, when applying GA using our similarity measure (*Tb*)?

For the first three research questions, the input test suite is generated by TRUST using *All-Transitions* coverage and in RQ4 we will discuss about the effect of using other coverage criteria. The test suite is made of 281 test cases and can detect all 15 detectable faults. Among 281 test cases 207 cannot detect any faults and 74 catch at least one fault. The average number of detected faults per test case is 0.72 and the maximum is five. Each fault is also detected on average by 13 test cases. There are nine faults which are only detected by three test cases and two faults are detectable by 65 test cases.

To capture the randomness of *FDR* results, which exists for all selection algorithms (even in Greedy search when it needs to select among test cases which have the same similarity measure), we ran each experiment 100 times and report distribution statistics. We report the results of different techniques for sample sizes less than 140 (~50% of the test suite) with intervals of 10, since our focus is, for practical reasons, on smaller size subsets. This is due to the fact that in practice test case selection is mostly used for selecting a relatively small sample of the test suite. Furthermore, for large sample sizes all selection techniques will usually be as good as random selection which typically detects most faults. We have performed parametric (*t*-test) and non-parametric (Mann-Whitney) statistical tests, with a significance level $\alpha = 0.05$, to compare the *FDR* means and medians of the proposed and alternative selection techniques. Due to space constraints, we only report the results of the non-parametric test, since it is more robust than the *t*-test when there are strong departures from normality and since we have a large enough sample of observations (100). In addition, we provide *FDR* means and medians over different runs for six sample sizes.

To compare effectiveness of different techniques, we use three measures based on *FDR*. These measures are complementary and help interpreting the *FDR* from different angles:

(1) $\rho(i)_\Gamma$ is the number of faults detected by $s_i$ (a subset of size *i* selected by technique Γ from the test suite TS with size *n*) divided by the total number of detectable faults in TS (15 in our case). This measure is used in the paper wherever we want to simply report the *FDR* for a given technique and sample size. Since we run each test suite 100 times on faulty programs we report the *FDR* means and variances.

(2) $AFDR_m^\gamma(\Gamma)$. Enables the overall comparison of two selection techniques for a range of sample sizes. $AFDR_m^\gamma(\Gamma)$, which is inspired by the APFD measure [11] for test case prioritization, is adapted to test case selection in our context. It is a measure for comparing curves and measures the sum of all $\rho(i)_\Gamma$ for all sample sizes in the given intervals and range (0 to *m*). More precisely, it is equal to the area under the curve representing $\rho(i)_\Gamma$ (*y*-axis) over different sample sizes (*x*-axis). Since sample size has discrete values, the area under the curve is calculated as:

$$AFDR_m^\gamma(\Gamma) = \frac{\frac{\rho(0)+\rho(m)}{2} + \sum_{i=1}^{\left(\frac{m}{\gamma}\right)-1} \rho(i*\gamma)_\Gamma}{\frac{m}{\gamma}}$$

where $0 \leq AFDR_m^\gamma(\Gamma) \leq 1$

As we discussed, in this paper we report the result of sample sizes less than 140 (~50% of the test suite) with intervals of 10, therefore we always report $AFDR_{140}^{10}(\Gamma)$.

(3) $\min_k(\Gamma)$ is the minimum number of test cases from the given test suite TS that are selected by technique Γ to detect at least $k\%$ of the detectable faults. This measure is more useful, from a practical standpoint, when selection techniques are compared with respect to their reduction in cost while ensuring a given fault detection rate.

To compare the cost of GA and Greedy, the execution time spent by the algorithms to select a subset is used as cost measure. The experiments has been conducted on a PC with Intel Core(TM)2 Duo CPU 2.40 Hz and 4 GB memory running Windows 7.

## 6.3 Experiment results

In the following subsections, we investigate each of the research questions stated above.

### *6.3.1 Which similarity measure is more effective for UML State Machine-based test case selection, in terms of FDR?*

Since there is no reported similarity-based selection measure for UML state machines, we need to tailor results from the most similar studies to obtain a baseline of comparisons. *Sb* by Ledru et al. in [9] and *It* by Cartaxo et al. in [7] are two potential similarity functions that we can adapt and apply on UML state machine-based test paths.

The measure *It* was straightforward to apply in our case, using the representation of our test paths from Section 5. We identify each transition uniquely, by a string composed of its source state, trigger, guard, and target state. States are identified by their name, triggers by the name of the operation or signal reception, and guards by their constraint. This means that transitions can be considered identical only if the entire string is the same. Since *Sb* is a general purpose function (it applies to the text of test scripts), it requires some modifications to be useful for our case. This was necessary since our executable test scripts are long and contain significant platform dependant information. Therefore, comparing such test scripts as strings results in useless similarity measures which are significantly blurred by irrelevant information. But we nevertheless decided to implement our adjusted version of *Sb* for strings using abstract test scripts, which in our case are the test paths defined in Section 5. Therefore, all elements of the test paths (states, triggers, and guards) constitute the alphabet of the strings to be compared. We then applied Levenshtein distance with standard parameters (1 for match and 0 for mismatch and gap) [32] on these strings. We denote this technique as modified *Sb* (*Ms*). The main difference between *Ms* and *It* is the fact that *Ms* accounts for orders of states and triggers (with or without guards) in the paths, whereas *It* only looks at the number of common transitions. We also have introduced yet another measure using only state similarities, Identical State Similarity (*Is*), which is equal to the number of identical states in two test paths divided by their average number of states. This measure is at the same level of detail as *It* but targeting different state-related faults.

We compare *Ms*, *Is*, and *Tb* with *It* as it is the only directly applicable solution from the literature for our models. We use a Greedy search since this is the technique used with *It* in the original study [7]. In short, for all similarity measures, our implementation of similarity-based Greedy search is exactly the same as in [7] and works as follows: In each step, the algorithm finds the most similar pair of test cases and removes the one which has less number of transitions from the test suite. This will continue till the number of remaining test paths in the test suite becomes equal to the required sample size. Removing the shorter test path in the selected pair actually aims to keep transition coverage as high as possible, while diversifying the subset. In cases where there is more than one pair with maximum similarity value, one of them is randomly chosen.

Figure 1and Figure 2 show the FDR means of the Greedy search using Tb, Is, Ms, and It ($\rho(i)_{Tb\mathrm{Gr}}$, $\rho(i)_{Is\mathrm{Gr}}$, $\rho(i)_{Ms\mathrm{Gr}}$, and $\rho(i)_{It\mathrm{Gr}}$) after running the algorithms 100 times for sample sizes less than 140 (~50% of the test suite). In addition, summarizes means and medians of $\rho(i)$ for these techniques and reports the Mann-Whitney test results highlighting cells in gray shade when there is a statistical difference between the selected comparison techniques and our proposed similarity measure *Tb*.

The results show that *Tb* and *Is* have the highest and lowest fault detection rates, respectively. The reason that *Is* is by far worse than the others can be explained by the fact that there are commonly several transitions per state and *Is* simply ignores differences between them as far as they have the same source or target states. The results also show that *It* is more effective than *Sb* for smaller sample sizes. This means that even when string similarity measures (e.g., Levenshtein) use detailed path information (e.g., the order of states and triggers), it may not be effective without careful tuning (e.g., gap and mismatch) and therefore makes such an approach less practical.

Since *It* is more effective than *Ms* and *Is*, we now take it as a baseline of comparison with our proposal *Tb*. As it is shown in Figure 1, Figure 2, and Table 1, for sample sizes less than 90 (~32% of the test suite) $\rho(i)_{Tb\mathrm{Gr}}$ is always higher than $\rho(i)_{It\mathrm{Gr}}$ and after 90 both techniques find all faults. This difference between *Tb* and *It* goes up to 35% (sample size 50) and is also shown to be statistically significant. An overall comparison of the two curves also shows the improvement brought by *Tb* ($AFDR^{10}_{140}(TbGr) \cong 0.88$ vs. $AFDR^{10}_{140}(ItGr) \cong 0.76$). This shows that in practice, our case study suggests it is likely better to use *Tb* than *It*. *TbGr* is also very effective with respect to finding most faults with fewer test cases: $\min_{95}(Tb) \cong 35 \cong$ (12% of the test suite) vs. $\min_{95}(It) \cong 90 \cong$ (~32% of the test suite). Although on average *Tb* is always more effective than *It*, looking at Figure 2 suggests that both techniques show a large variance for smaller sample sizes. In practice, this means that if the tester runs out of luck, selecting a subset of test cases can lead to a very low fault detection. In the next section we show how using GA can help increase our confidence in *Tb* by decreasing its variance.

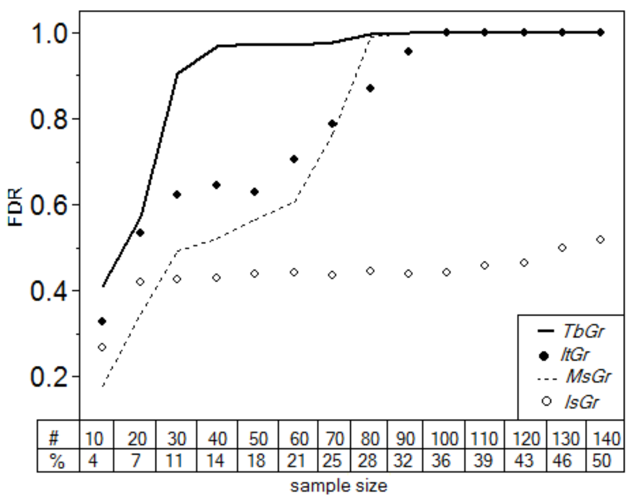

**Figure 1. The average *FDR* of *TbGr, ItGr, MsGr, IsGr* for different sample sizes.**

**Table 1. RQ1: The median and mean *FDR*s per sample size (10 to 100 intervals of 10) over 100 runs and the Mann-Whitney test results (significant differences on medians with *TbGr* highlighted as gray cells) for different measures using Greedy search.**

| Selection technique | | *FDR*s per sample size | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| *TbGr* | median | 0.4 | 0.57 | 0.93 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | mean | 0.41 | 0.57 | 0.90 | 0.97 | 0.97 | 0.97 | 0.98 | 1 | 1 | 1 |
| *ItGr* | median | 0.33 | 0.53 | 0.6 | 0.6 | 0.5 | 0.8 | 0.8 | 0.8 | 1 | 1 |
| | mean | 0.33 | 0.53 | 0.62 | 0.65 | 0.63 | 0.70 | 0.79 | 0.87 | 0.95 | 1 |
| *IsGr* | median | 0.2 | 0.4 | 0.4 | 0.4 | 0.4 | 0.4 | 0.4 | 0.4 | 0.4 | 0.4 |
| | mean | 0.29 | 0.42 | 0.43 | 0.43 | 0.44 | 0.44 | 0.44 | 0.45 | 0.44 | 0.44 |
| *MsGr* | median | 0.13 | 0.4 | 0.4 | 0.6 | 0.6 | 0.6 | 0.8 | 1 | 1 | 1 |
| | mean | 0.18 | 0.35 | 0.5 | 0.53 | 0.57 | 0.6 | 0.77 | 0.99 | 1 | 1 |

**Table 2. RQ2-3: The median and mean *FDR*s per sample size (10 to 100 intervals of 10) over 100 runs and the Mann-Whitney test results (significant differences on medians with *TbGa*(175ms) highlighted as gray cells) for different selection techniques.**

| Selection technique | | *FDR* per sample size | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| *TbGa* 175ms | median | 0.53 | 0.93 | 0.93 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | mean | 0.56 | 0.83 | 0.92 | 0.96 | 0.97 | 0.97 | 0.99 | 0.98 | 0.98 | 0.98 |
| *TbGa 1000ms* | median | 0.53 | 0.9 | 0.93 | 0.93 | 0.93 | 1 | 1 | 1 | 1 | 1 |
| | mean | 0.55 | 0.82 | 0.94 | 0.95 | 0.95 | 0.97 | 0.99 | 1 | 1 | 1 |
| *TbGr* | median | 0.4 | 0.57 | 0.93 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | mean | 0.41 | 0.57 | 0.90 | 0.97 | 0.97 | 0.97 | 0.98 | 1 | 1 | 1 |
| *ItGr* | median | 0.33 | 0.53 | 0.6 | 0.6 | 0.5 | 0.8 | 0.8 | 0.8 | 1 | 1 |
| | mean | 0.33 | 0.53 | 0.62 | 0.65 | 0.63 | 0.70 | 0.79 | 0.87 | 0.95 | 1 |
| *CvGr* | median | 0.4 | 0.53 | 0.6 | 0.6 | 0.6 | 0.8 | 0.8 | 0.8 | 0.8 | 0.8 |
| | mean | 0.37 | 0.52 | 0.63 | 0.67 | 0.69 | 0.77 | 0.77 | 0.81 | 0.85 | 0.86 |
| *Rnd* | median | 0.27 | 0.37 | 0.5 | 0.6 | 0.6 | 0.6 | 0.8 | 0.8 | 0.8 | 0.8 |
| | mean | 0.28 | 0.38 | 0.5 | 0.57 | 0.61 | 0.65 | 0.76 | 0.76 | 0.8 | 0.84 |

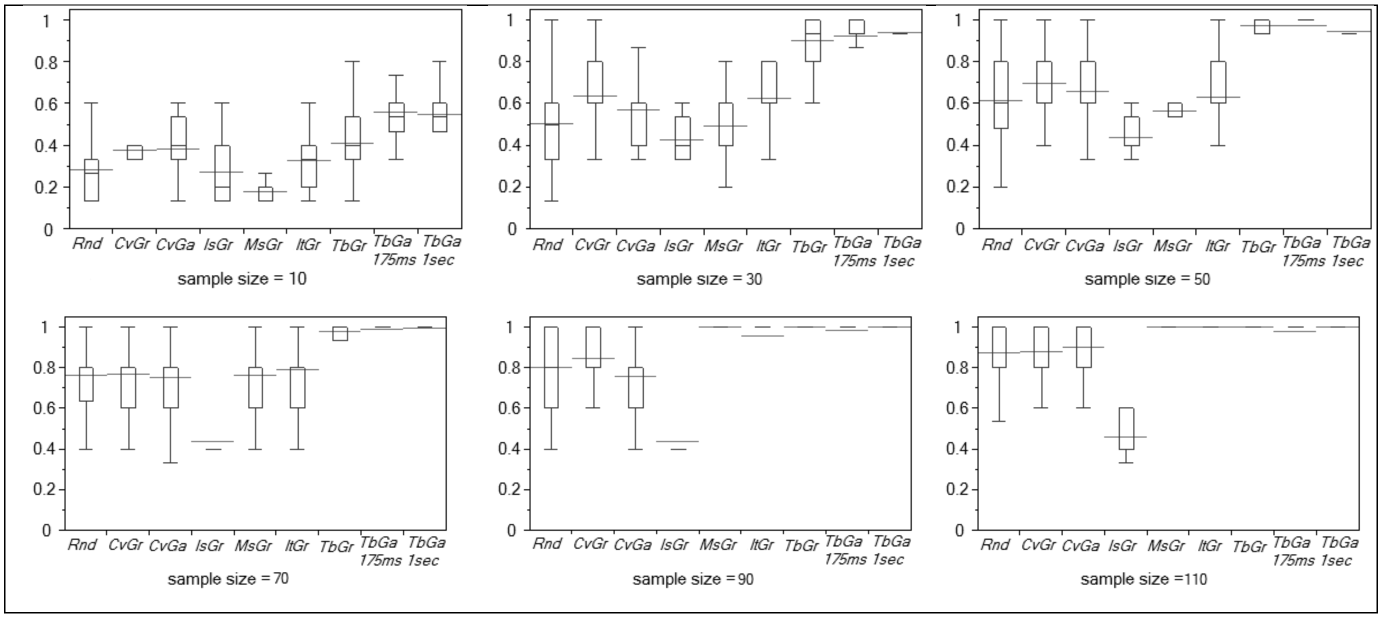

**Figure 2. *FDR* (y-axis) Boxplots for different selection techniques (x-axis) for sample sizes ranging from 10 to 110 by intervals of 20 over 100. The Boxplots show the 10th, 25th, 50th, 75th, and 90th percentiles and means.**

### 6.3.2 *Is using GA for test case selection significantly more cost-effective (in terms of time spent to find a solution) compared to a Greedy search?*

Before discussing about the cost-effectiveness analysis between GA and Greedy search, it is worth mentioning that using an exhaustive search in our case (and for most realistic cases) is not an option, since the search space size for selecting a subset of size *n* is equal to the number of possible n-combinations within a test suite of a given size. In our case, as an example, the search space size for *n*=28 (~10% of the test suite) is $\binom{281}{28}$ ≅2.9*10^38.

Our implementation of Greedy search does not have any parameter settings. In this paper, we do not tune GA, instead we use the parameters based on our previous experience in using GA [33]. The only setting of GA that we will discuss here is the stopping criterion since it has direct effect on GA's cost and there is no obvious decision. In this paper, we only report the result taken from experiments with the fixed execution time stopping criterion since we wanted to keep cost constant when comparing *FDR* with Greedy search.

Cost here will be measured as execution time since this drives the applicability of a test strategy as discussed in Section 1. Running Greedy search 100 times for sample sizes from 10 to 140 showed that it needs 175ms on average for each selection. Therefore, we set the GA stopping criterion to 175ms to compare their *FDR* using constant execution time. Next, we will increase execution time to a significantly larger but yet practical number (1000ms) and investigate how much more effective GA can be. Note that Greedy search cannot be improved even if one can afford running it for a longer period of time as opposed to GA which can potentially be improved within practical bounds.

Using our proposed similarity measure *Tb* we investigate the extent to which GA can improve *FDR*. Using execution times of 175ms (as for Greedy search), Figure 3 and Figure 2 show *FDR* distributions for GA and Greedy search using *Tb* ($\rho(i)_{Tb\mathrm{Ga}}$ and $\rho(i)_{TbGr}$) while running the algorithms 100 times for each sample size from 10 to 140 (~50% of the test suite). Greedy always shows a lower *FDR* (Table 2 shows that the differences are statistically significant) than GA for sample sizes less than 75 (~27% of the test suite), with a maximum difference of ~30% (sample size 25). In practice, for large test suites, this is probably the most important part of the sample size range. For larger sample sizes, an execution tie of 175ms does not seem to be enough for GA to be as effective as Greedy search. The main reason is that for larger sample size GA takes a great deal of time to generate an initial population with unique test paths and does not have enough time to generate many subsequent populations. Still for the overall sample size range GA is more effective: $AFDR_{140}^{10}(TbGr) \cong 0.88$ vs. $AFDR_{140}^{10}(TbGa) \cong 0.90$. To find 95% of the faults both techniques need the same number of test cases: $\min_{95}(TbGa) = \min_{95}(TbGr) \cong 35$. However, the *FDR* variance for Greedy search is significantly higher than that for GA (Figure 2), especially for sample sizes less than 50 (~ 18% of the test suite). This means that although both techniques, on average, can find 95% of the faults with 35 test cases, GA entails less risk. In practice, people need to be confident in the results of a technique to use it. They cannot rely on chance. One selects only one subset, and no one wants to incur the risk (no matter how low the probability) of missing most of the faults.

The increase in execution time for GA's stopping criterion shows that on average there is no practically significant *FDR* improvement ($AFDR_{140}^{10}(TBS_{GA}) \cong 0.90$ for both 175ms and 1sec execution times). In this case, running GA for longer execution times does not seem to produce significantly better results. An explanation could be within 175ms GA finds a (near-)optimal solution in our case. However, increasing execution time helps decrease the *FDR* variance and therefore decreases the risk involved in test selection. Another point is that GA needs less time for smaller sample sizes. Therefore, GA running 175ms starts to perform slightly worse than GA running 1000ms for subsets larger than 70 (~25% of the test suite), as illustrated in Figure 2 (sample sizes > 70).

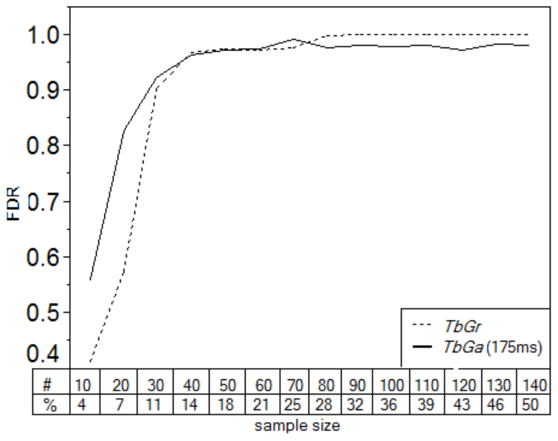

**Figure 3. The average *FDR* of *TbGr* and *TbGa(175ms)* for different sample sizes.**

### 6.3.3 *Are similarity-based selection techniques more effective than coverage-based and random selection techniques?*

In this research question, we are interested in the improvement that similarity-based techniques can provide for model-based test case selection when compared to simpler alternatives. We compare our proposal (*TbGa*) with three different techniques: (1) Random selection (RnD) as a baseline of comparison for any type of (meta)heuristic search, (2) Additional coverage Greedy selection [10, 11] (*CvGr*), and (3) *ItGr* as the state of the art for similarity-based techniques. We also have experimented with using GA for coverage-based selection as it is defined in [12, 22]. The results show that *CvGr* outperforms the GA-coverage-based technique in our case study, as visible in Figure 2. Therefore, we compare with *CvGr* in this section.

All techniques are spending almost the same execution time for selection (on average less than 200 ms). Figure 2 and Figure 4 show *FDR* for the different techniques ($\rho(i)_{Tb\mathrm{Ga}}$, $\rho(i)_{It\mathrm{Gr}}$, $\rho(i)_{CvGr}$, $\rho(i)_{\mathrm{Rnd}}$) when running the algorithms 100 times for each sample size from 10 to 140. Based on Table 2, for all sample sizes, *TbGa*(175ms) is significantly more effective than the others.

As we can see that, on average, the FDR of *TbGa* is significantly higher than that for *Rnd* and *CvGr* for all sample sizes, with maximum differences of 35% (*Rnd*) and 30% (*CvGr*). The comparison over the entire sample size range also confirms this observation: $AFDR_{140}^{10}(Rnd) \cong 0.66$, $AFDR_{140}^{10}(CvGr) \cong 0.72$ and $AFDR_{140}^{10}(TbGa) \cong 0.90$.

The next best technique, both in terms of $\rho(i)$ and $AFDR_m^{10}$, is *ItGr*, which shows that again a similarity-based technique outperforms the coverage-based and random selection. *TbGa* is also very effective in finding more faults with less number of test cases. For example, *TbGa* (175ms) can find 95% of the faults with only 35 test cases ($\min_{95}(TbGa) \cong 35$) where both coverage-based and random selection techniques cannot find 95% of the faults, even when using 140 test cases. Another observation is that coverage-based techniques are not much more effective than random selection.

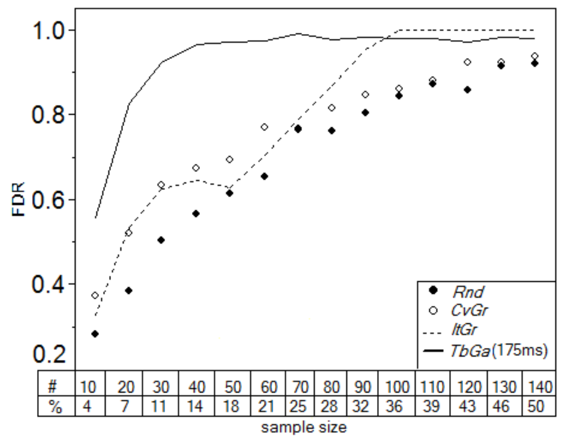

**Figure 4. The average *FDR* of *Rnd, CvGr, ItGr, TbGa(175ms)*, for different sample sizes**

### *6.3.4 In the context of MBT, what is the practical benefit of test case selection, on a representative industrial case study, when applying TbGa?*

In this subsection, we look at a broader question which is about the usefulness of test case selection for reducing the size of the test suite generated by MBT tools, which is the main motivation for this study. We will answer RQ4 by answering two sub-questions:

RQ4.1. Are test selection techniques more effective than using stricter coverage criteria?

RQ4.2. How effective is test selection in reducing the cost of testing in MBT?

As we discussed in Section 2, using a stricter criterion (for example using *all-transitions* instead of *all-transition-pairs*) is an alternative to selection techniques. If after using the least demanding criterion (e.g., *all-transitions*), the test suite is still too large, then using criteria such as *all-length-N*, where *N* is the maximum test path length can be used. Here we compare these alternatives with using a similarity-based selection technique. In our case, *N*=3 results in around 150 test cases and N=2 yields 27 test cases. Since *TbGa* (1sec) shows on average a 100% *FDR* with 75 test cases, then a test suite of 150 is obviously suboptimal. Comparing the result of *length-2* with *TbGa*(175ms) yields $\rho(27)_{TbGa} \cong 0.90$ whereas $\rho(27)_{length\_2} \cong 0.34$. This result confirms our claim that stricter criteria cannot be a replacement for test selection techniques.

With respect to RQ 4.2, we are looking at the reduction of cost that a selection technique like *TbGa* can provide for a MBT testing strategy. In our case the original test suite contains 281 test cases. For a one-second execution time, $\min_{100}(TbGa) \cong 75$ meaning that 75 test cases are as effective (same *FDR*) as the entire test suite (281 test cases), entailing a 73% reduction in cost. As we discussed earlier, in distributed and embedded software systems (as our case study system), where test execution cost can be very significant, this 73% reduction is of practical importance.

### *6.3.5 Discussion on validity threats*

In this subsection we discuss the potential threats to the validity of the study using the framework discussed in [34] about conducting empirical studies for search-based testing.

*Construct validity:* For measuring test execution cost, we used the actual time spent by different algorithms and running all algorithms on the same machine. Our effectiveness measure (*FDR*) is based on a set of real faults, as explained earlier, that we used to create mutant programs.

*Internal validity*: We implemented both Greedy and GA algorithms and strived to achieve the same level of optimization. GA parameter tuning may have positive effect on its performance (which we have not systematically carried out) but Greedy does not have any influential parameter. This means that GA could possibly work better with some fine tuning. Regarding our implementation of *It*, since we had to adapt its definition to our context (UML state machine and the encoding and representation of test paths), it might be a potential threat and one could argue that it is possible to more effectively implement it.

*Conclusion validity*: Hundred independent runs were performed to account for random variation and obtain a sufficient number of observations to report means, medians, and standard deviations. We used the *t-test* and *Mann-Whitney test* for independent samples to check the statistical differences in *FDR* across selection techniques, but only reported the latter here for reasons explained earlier. We also discussed about practical significance by looking at the magnitude of the differences between *FDR* and cost of different techniques.

*External validity*: Our results rely on one industrial case study using a given set of real faults. Though running such studies is very time consuming, it is obviously required to replicate it as many times as possible. However, as discussed earlier, the system used here is typical of a broad category of industrial systems: control systems with state-dependent behavior, controlling sensors and actuators.

## 7. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced a new technique for selecting test cases in the context of Model-Based Testing (MBT), more specifically UML state machine-based testing. Our motivation is to make MBT scalable in situations where executing test cases satisfying a coverage criterion (e.g., all transitions) is too expensive, such as when there is hardware in the loop, interacting external systems, or test case executions are lengthy.

We propose a new similarity-based test case selection technique, which contains a similarity measure based on UML state machines' triggers and guards on the transitions. It uses a Genetic Algorithm (GA) as a selection mechanism in order to minimize similarity among test cases. Our results, based on an industrial case study of a safety controller, showed that our approach yields significantly better results than other alternatives such as random, coverage-based, and other existing similarity-based selection techniques. We also have shown that our technique can significantly reduce the cost of test case execution in MBT by selecting 27% of the test suite to be executed, while retaining a 100% fault detection rate. In the future, we plan to have a more exhaustive investigation of other possible similarity measures and selection techniques. We will also investigate hybrid techniques which use both coverage and similarity measures, for example using a multi-objective GA. We will also conduct additional studies on other industrial systems to replicate the current study.

## 8. REFERENCES

[1] Utting, M. and Legeard, B., *Practical Model-Based Testing: A Tools Approach*, Morgan-Kaufmann, 2006.

[2] Pender, T., *UML Bible*, Wiley, 2003.

[3] Ali, S., Hemmati, H., Holt, N. E., Arisholm, E. and Briand, L., *Model Transformations as a Strategy to Automate Model-Based Testing - A Tool and Industrial Case Studies*, Simula Research Laboratory, Technical Report(2010-01), 2010.

[4] Binder, R. V., *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley Professional, 1999.

[5] Mathur, A. P., *Foundations of Software Testing*, Addison-Wesley Professional, 2008.

[6] Jones, J. A. and Harrold, M. J., *Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage*, *IEEE Transactions on Software Engineering*, 29(3), 2003, 195-209.

[7] Cartaxo, E. G., Machado, P. D. L. and Neto, F. G. O., *On the use of a similarity function for test case selection in the context of model-based testing*, *Software Testing, Verification and Reliability*, Published Online: 22 Jul 2009.

[8] Goldberg, D. E., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Professional, 2001.

[9] Ledru, Y., Petrenko, A. and Boroday, S., *Using String Distances for Test Case Prioritisation*, In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, 2009.

[10] Rothermel, G., Harrold, M. J., Ronne, J. v. and Hong, C., *Empirical studies of test-suite reduction*, *Software Testing, Verification and Reliability*, 12(4), 2002, 219-249.

[11] Elbaum, S. G., Malishevsky, A. G. and Rothermel, G., *Test Case Prioritization: A Family of Empirical Studies*, *IEEE Transactions on Software Engineering*, 28(2), 2002, 159-182.

[12] Li, Z., Harman, M. and Hierons, R. M., *Search Algorithms for Regression Test Case Prioritization*, *IEEE Transactions on Software Engineering*, 33(4), 2007, 225-237.

[13] Yoo, S., Harman, M., Tonella, P. and Susi, A., *Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge*, In *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009.

[14] Fahad, M. and Nadeem, A., *A survey of UML based regression testing.* , In *Proceedings of the Intelligent Information Processing*, 2008.

[15] Orso, A., Do, H., Rothermel, G., Harrold, M. J. and Rosenblum, D. S., *Using component metadata to regression test component-based software*, *Software Testing, Verification and Reliability*, 17(2), 2007, 61-94.

[16] Muccini, H., *Using Model Differencing for Architecture-level Regression Testing*, In *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*, 2007.

[17] McMaster, S. and Memon, A., *Call-Stack Coverage for GUI Test Suite Reduction*, *IEEE Transactions on Software Engineering*, 34(1), 2008, 99-115.

[18] Chen, Y., Probert, R. L. and Ural, H., *Regression test suite reduction based on SDL models of system requirements*, *Journal of Software Maintenance and Evolution: Research and Practice*, 21(6), 2009, 379-405.

[19] Jourdan, G.-V., Ritthiruangdech, P. and Ural, H., *Test Suite Reduction Based on Dependence Analysis*, *Computer and Information Sciences – ISCIS 2006*, Springer Berlin / Heidelberg, 4263/2006, 1021-1030, 2006.

[20] Farooq, U. and Lam, C. P., *A Max-Min Multiobjective Technique to Optimize Model Based Test Suite*, In *Proceedings of the 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, 2009.

[21] Farooq, U. and Lam, C. P., *Evolving the Quality of a Model Based Test Suite*, In *Proceedings of the Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, 2009.

[22] Ma, X. Y., Sheng, B. K. and Ye, C. Q., *Test-Suite Reduction Using Genetic Algorithm*, *Advanced Parallel Processing Technologies*, Springer Berlin / Heidelberg, 3756/2005, 2005.

[23] Chen, T. Y. and Lau, M. F., *A simulation study on some heuristics for test suite reduction*, *Information and Software Technology*, 40(13), 1998, 777-787.

[24] Leon, D. and Podgurski, A., *A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases*, In *Proceedings of the IEEE International Symposium on Software Reliability Engineering*, 2003.

[25] Harman, M., *The Current State and Future of Search Based Software Engineering*, In *Proceedings of the Future of Software Engineering*, 2007, IEEE Computer Society.

[26] Simão, A. d. S., Mello, R. F. d. and Senger, L. J., *A Technique to Reduce the Test Case Suites for Regression Testing Based on a Self-Organizing Neural Network Architecture*, In *Proceedings of the COMPSAC*, 2006.

[27] Jiang, B., Zhang, Z., Chan, W. K. and Tse, T. H., *Adaptive random test case prioritization*, In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, 2009.

[28] Masri, W., Podgurski, A. and Leon, D., *An Empirical Study of Test Case Filtering Techniques Based on Exercising Information Flows*, *IEEE Transactions on Software Engineering*, 33(7), 2007.

[29] Ramanathan, M. K., Koyutürk, M., Grama, A. and Jagannathan, S., *PHALANX: a graph-theoretic framework for test case prioritization.* , In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing*, 2008.

[30] http://www.levenshtein.net/

[31] Whitley, D., *The genitor algorithm and selective pressure: Why rank-based allocation of reproductive trials is best*, In *Proceedings of the Third International Conference on Genetic Algorithms (ICGA-89)*, 1989.

[32] Gusfield, D., *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.

[33] Arcuri, A., *Insight Knowledge in Search Based Software Testing*, In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2009.

[34] Ali, S., Briand, L. C., Hemmati, H. and Panesar-Walawege, R. K., *A Systematic Review of the Application and Empirical Investigation of Search-based Test-Case Generation*, *Accepted for publication in IEEE Transactions on Software Engineering, Special issue on Search-Based Software Engineering (SBSE)*, 2009.