An Industrial Investigation of Similarity Measures for Model-Based Test Case Selection

Hadi Hemmati and Lionel Briand
Simula Research Laboratory and
Department of Informatics, University of Oslo
{hemmati, briand}@simula.no

***Abstract*—Applying model-based testing (MBT) in practice requires practical solutions for scaling up to large industrial systems. One challenge that we have faced while applying MBT was the generation of test suites that were too large to be practical, even for simple coverage criteria. The goal of test case selection techniques is to select a subset of the generated test suite that satisfies resource constraints while yielding a maximum fault detection rate. One interesting heuristic is to choose the most diverse test cases based on a pre-defined similarity measure. In this paper, we investigate and compare possible similarity functions to support similarity-based test selection in the context of state machine testing, which is the most common form of MBT. We apply the proposed similarity measures and a selection strategy based on genetic algorithms to an industrial software system. We compare their fault detection rate based on actual faults. The results show that applying Jaccard Index on test cases represented as a set of trigger-guards is the most cost-effective similarity measure. We also discuss the overall benefits of our test selection approach in terms of test execution savings.**

Keywords—Test case selection; model-based testing; UML state machine; similarity measure; and genetic algorithms

I. INTRODUCTION

In recent years the software industry has shown increasing interest in automating the process of test case generation using models of the system under test. The main idea behind model-based testing (MBT) is to generate executable test cases (including oracles) by systematically traversing system models (e.g., represented as UML state machines) based on test strategies usually involving some form of coverage criterion that aims to cover certain features of the model (e.g., all transitions in state machine-based testing (SMBT)) [1]. MBT tools are becoming increasingly sophisticated and robust and MBT is becoming the best test automation solution for many practitioners. However, there are still many unsolved issues regarding how to scale up MBT to large industrial software systems. Our experience has shown that in many practical contexts even simple coverage criteria yield far too many test cases to be usable.

In general, system test case execution can be very costly in most embedded and distributed systems when there is hardware in the loop or test execution requires access to dedicated test infrastructures or no automated oracle is available. Testing such systems requires, for example, assigning enough resources (e.g., actual physical devices) to the test case, properly handling acceptable delays in the system execution and the network communication, and manually analyzing the results when there is no automated oracles. This can be a major hindrance for making MBT practical, especially in the context of system testing when release deadlines are close and the project is already often behind schedule.

Test case selection is used to reduce test suite sizes to what can be handled in a specific context while retaining the largest possible fault revealing power. In general, regardless of the heuristic used, this test case selection problem is NP hard (traditional set cover) [2]. Other than random selection, where there is no guidance to select test cases, there have been two main types of test case selection heuristics proposed in the literature. In coverage-based selection [3], the underlying hypothesis is that “the test suites which achieve more coverage (of model or code) are more likely to detect faults”. In similarity-based test case selections (STCS) [4], the underlying hypothesis is that “the more diverse the test suites the higher their fault revealing power”. To use this latter approach one needs a (dis)similarity measure to calculate the diversity of a subset by averaging all pair-wise similarity values. After defining a similarity measure, a selection algorithm is required to choose a set of test cases with the minimum pair-wise similarity among its members. In [5], we introduced a new STCS technique for SMBT, which includes a new similarity measure using triggers and guards on transitions of state machines and a genetic algorithm (GA)-based selection algorithm. Applying this technique on an industrial case study, we showed that STCS in general and more specifically our proposed approach is by far more effective at detecting real faults than coverage-based and random selection.

In this paper, we take a deeper look into the effect of similarity measures in test case selection by distinguishing the test case representation (encoding) from the similarity function as two distinct parameters of a similarity measure. A comprehensive investigation of different similarity functions is performed through an industrial empirical study where the software under test (SUT) is a safety controller system which is modeled using UML state machines and test cases are generated using our MBT tool (TRUST) [1]. The case study, although modest compared to other industrial systems, is much larger both in terms of models and number of generated test cases, than what is reported in related works. Moreover, the faults we use are real (no seeded faults) thus significantly increasing the level of realism. The results show that choosing a proper similarity measure has a very significant effect on fault detection. The best similarity measure results in increasing the fault detection rate (FDR) by 50% when compared to the best alternative, coverage-based selection in this case, for small sample sizes (~10% of the test suite). In addition, our approach for test case selection reduces significantly the cost of MBT by reducing the number of test executions. For example, to achieve a FDR higher than 90%, we only need to execute 20 test cases selected with our approach, whereas other alternatives select at least 85 test cases to achieve the same FDR. Our approach therefore entails a 77% saving in execution cost.

The rest of the paper is organized as follows. Section II reports on background information about test case selection. Section III discusses on different similarity functions which are used in this study. Section IV provides a brief overview of related works covering STCS techniques. Section V reports the experimentation results of applying different STCS techniques on an industrial case study. Section VI concludes the paper and outlines our future work plan.

II. Test case selection

In general, there are two options for decreasing the number of test case executions. The first is generating fewer test cases which in the context of MBT means using a less demanding coverage criterion. For instance, if using all transition-pairs [6] generates a too large test suite, the all-transitions [6] criterion can be adopted instead to decrease the number of test cases. This still results in systematic testing but may reduce the FDR. The second approach is to select a subset of test cases from the test suite for execution. This can be done either by test suite reduction where the goal is to minimize the test suite by removing redundant test cases with respect to a criterion (e.g., code coverage) or by test case selection where the goal is to select a subset of the entire test suite that maximizes fault detection based on a heuristic, given a maximum number of test cases. Using a less demanding coverage criterion or test suite reduction is often impractical as one cannot precisely select a maximum number of test cases. Furthermore, we have shown in [5] that even when the scale of reduction achieved by using less demanding criteria is acceptable, it is still much less cost-effective than a STCS. Test case prioritization, which does not remove any test case but order their execution [7], could also be considered but does not directly address our problem, though some of the underlying ideas could be adapted. For example, as we will see in the related work section, most similarity measures that are used in similarity-based test case prioritization can be used in test case selection as well. In this study, the focus is on test case selection.

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". We can classify test case selection techniques as follows: (1) those which make use of test execution information as it is usually the case in regression testing and (2) those which select test cases solely based upon the characteristics of the (abstract) test cases. The latter category is the one of interest in our context where the test suite cannot be executed before selection. Therefore, execution-based heuristics such as execution traces (e.g. call stack [8]) are not applicable here.

A. Coverage-based Test Case Selection

Maximizing coverage has been a common practice in selection and prioritization for years. Most studies in test case selection (even those which are general purpose and not specific to regression testing) are based on code-level information (e.g., additional statement coverage[7]) and cannot directly be applied to MBT. However, it is possible to extract additional information from test cases to help the selection even without executing it. 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 selection techniques are re-expressed into optimization problems where the goal is to select the best subset of test cases to achieve full coverage. For example, a technique in [7] uses a Greedy search to select, at every step, the test case that covers the most uncovered statements (additional coverage-based technique) whereas in [9] a GA is used to achieve maximum coverage.

B. Similarity-based Test Case Selection

In STCS techniques, a (dis)similarity measure is used for comparing similarity (diversity) between a pair of test cases. A similarity measure is the value that a similarity function assigns to two inputs which are being compared. Inputs are usually encoded as a set or sequence of elements. In the context of MBT, the inputs are abstract test cases instead of concrete test cases. We do not need the execution information of the test case and abstract test cases are naturally generated as a first step by MBT. Therefore, we reduce the cost of test case generation by only generating executable test cases for the selected abstract test cases and also by hiding the unnecessary information for similarity comparisons. For example, in SMBT an abstract test case representation can be a path in the state machine specifying the SUT. In general, different faults can be detected by the same test path instantiated with different test data. Therefore, to compare different techniques, it is necessary to run the selected test paths with different input data and use their FDR distribution.

Representation (encoding) of the test cases has an important effect on the similarity measure. Though in state-based testing a test path represents an encoded abstract test case, the test path can be described at different levels of details. We consider three possible encodings for a test path in UML state machine: state-based, transition-based, and trigger-guard-based:

1. state-based: <tp> ::= *state* | *state* "," <tp>
2. transition-based: <tp> ::= *trans* | *trans* "," <tp>
3. trigger-guard-based: <tp> ::= <TrGu> | <TrGu> "," <tp>
   < TrGu > ::= *trig* |*guard* | *id* | *guard* "+" *trig*

where *state* is the id of a state, *trans* the id of a transition, *trig* the id of a trigger, and *guard* the id of a guard in the state machine. In the case of trigger-guard-based encoding, a transition is identified by its trigger and guard. It can be only a trigger, or a guard or both together. If there is a transition with no guard and trigger, we use the transition id as its identifier. Note that the difference between trigger-guard-based and transition-based encoding is that in trigger-guard-based encoding transitions with the same trigger-guard but different source or target state are identical.

Given an encoding, one may use different similarity functions to calculate the similarity value. Similarity is usually defined on either two sets or two sequences of elements. The main difference is that set-based similarity measures as opposed to sequence-based ones do not take the order of elements into account. For example, if test case 1 includes method calls A and B and test case 2 includes method calls B and A, respectively, and method calls are the only encoded elements in the test path, set-based similarity functions assume these two test cases as identical. In the next section, the functions which are used in our study are introduced. In this paper, we take the best encoding from our previous study [5] and investigate the effect of different similarity functions on the FDR of the selected test cases.

Given a set of encoded test cases (s_n) and a similarity function ($SimFunc$), the test case selection 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 (or other encoded abstract test cases in MBT) in s_n represented by tp_i and tp_j. The last step in STCS is applying a selection algorithm which selects a subset of test cases with minimum average pair-wise similarity ($SimMsr$). Our experience in [5] showed that using a GA is more cost-effective than a Greedy search which is common in the STCS literature [4]. Therefore, in this study we use a GA as our selection mechanism. GAs 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 are chosen by the selection mechanism to 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 be part of the population of the next generation. The mutation operator is applied to make small changes in the chromosomes of the offspring. Eventually, after a number of generations, an individual that solves the addressed problem will be evolved. We use a steady state GA where an individual (i.e., a solution to the problem) is s_n (subset of *TS* with size *n*). SimFunc($tp_i\,, tp_j$) is the fitness function to be minimized. A mutated test path is replaced by a test path that is selected at random from the set of all possible test paths. We do not tune our GA parameters and use what is suggested in the literature (e.g. [10])—a high crossover probability (0.75) and low mutation probability (inversely proportional to the population size) and a reasonable sample size (50). The stopping criterion used in this study is stopping after a fixed period of time (175ms), which is 10 times more than the amount of time that a basic Greedy search would take on average in our case study. Though the GA is more costly than the Greedy, the GA is still a better option since 175ms is negligible compared to the execution time of a test case and no improvement can be obtained with Greedy even if we let the algorithm search for longer periods of time (e.g., 175ms).

III. SIMILARITY FUNCTION

As we mentioned in Section B, common similarity functions are either set-based or sequence-based. In this study, we compare measures which have been used in the similarity-based selection or prioritization literature (Counting, Hamming, Jaccard, and Levenshtein functions) and measures (Global and Local alignments) which have not been used in software testing but are commonly used in other disciplines (such as bioinformatics) for similarity comparisons.

A. *Set-based Similarity Functions*

The main two measures in this category are the Jaccard Index [11] and the Hamming Distance function [12]. However, we also compare another measure (we call it Counting function) which is used in the only other reported study about STCS in MBT [4].

1) *Counting Function*

The Counting function (Cnt) is the simplest way of comparing two sets which we have reused from the measure used in [4] for comparing two sets of transitions. Given two sets *S1* and *S2*, Cnt(*S1*, *S2*) = number of identical members in *S1* and *S2* divided by the average number of members in *S1* and *S2*.

2) *Hamming Distance*

Hamming Distance is one of the most used distance functions in the literature which is a basic edit-distance. The edit-distance between two sequences is defined as the minimum number of edit operations –insertions, deletions, and substitutions– needed to transform the first sequence into the second [12-14]. Hamming is only applicable on identical length inputs and is equal to the number of substitutions required in one input to become the second one [12]. If all inputs are originally of identical length, the function can be used as a sequence-based measure. However, in most applications, inputs have different lengths. Therefore, to force them to have an identical length, a binary vector is made per input that indicates which elements from the set of all possible elements of the encoding exist in the input. As a result, the function does not preserve the original order of elements in the input anymore and it becomes a set-based similarity function. In our case, to use Hamming Distance, each encoded test case is represented as a binary vector of length *n*, where *n* is the number of all possible elements for that encoding (e.g., *n* is the number of all states, if state-based encoding is used). A bit in the vector is *true* only if the encoded test case contains the corresponding element (e.g., the state in the above example). We also need to change distance into similarity in our study. Therefore, our version of the Hamming function (denoted Ham) counts identical bits in the two input vectors, as opposed to the standard Hamming Distance where differences are counted.

3) Jaccard Index

Jaccard Index or Jaccard similarity coefficient (denoted Jac) is defined to compare similarity of sample sets [11]. It is defined as the size of the intersection divided by the size of the union of the sample sets:

$$Jac(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

B. Sequence-based Similarity Functions

Sequence similarity is usually applied on string matching in text mining [14] and homologous pattern recognition in bioinformatics [13]. Here we are using basic edit distance (Levenshtein) from text mining and global and local alignment from bioinformatics.

1) Levenshtein

One of the the most well-known algorithms implementing edit-distance which is not limited to identical length sequences is Levenshtein [14] where each mismatch (substitutions) or gap (insertion/deletion) increases the distance by one unit. To change distances into similarities, we need to reward each match and penalize each mismatch and gap. The dynamic programming [15] implementation of the algorithms in addition to examples can be found in [14]. The relative scores assigned to matches, mismatches, and gaps can be different (operation weight). Moreover, in some versions of the algorithm there are different match scores based on the type of matches (alphabet weight). Here we use a basic setting for the function (denoted Lev) where matches are rewarded by one point and mismatch and gap are treated the same by giving no reward.

2) Global and local sequence alignments

An alignment of two sequences is a mapping between positions in them [13]. In local alignment the goal is finding the best alignment for sub-sequences of given sequences while in global alignment the entire sequences must be aligned. The most basic global and local alignment algorithms are respectively Needleman-Wunsch (NW) [13] and Smith-Waterman (SW) [13]. The dynamic programming implementation of the algorithms, along with examples, can be found in [13]. The scoring matrix F for Needleman–Wunsch alignment is defined as:

$$\mathrm{F}[0][j] = -j * d,\ \mathrm{F}[i][0] = -i * d$$

$$\mathrm{F}[i][j] = \max\begin{cases} \mathrm{F}[i-1][j-1] + \mathrm{sim}(x_i, y_j), \\ \mathrm{F}[i-1][j] - d, \\ \mathrm{F}[i][j-1] - d. \end{cases}$$

Where the $\mathrm{sim}(x_i, y_j)$ returns the match/mismatch scores between the *i*th member of *x* and the *j*th member of *y*, and *d* is the gap penalty. The similarity between the two sequences is F[*n*][*m*] where *n* and *m* are the lengths of the input sequences. The scoring matrix F for SW alignment is defined in a similar way as in the NW scoring matrix but with a small change:

$$\mathrm{F}[0][j] = -j * d,\ \mathrm{F}[i][0] = -i * d$$

$$\mathrm{F}[i][j] = \max\begin{cases} \mathrm{F}[i-1][j-1] + \mathrm{sim}(x_i, y_j), \\ \mathrm{F}[i-1][j] - d, \\ \mathrm{F}[i][j-1] - d, \\ 0 \end{cases}$$

Having zero as one option in the max function results in having only positive values. In this approach, the similarity value is the highest F[*i*][*j*] which identifies the longest most similar subsequence between input sequences as well. Note that each alignment technique uses a similarity function to align the input sequences. The NW alignment algorithm actually uses the Levenshtein similarity function but with different weightings for match, mismatch and gap. In this study, we use Levenshtein with match score +3, mismatch -2, and a gap penalty of 1 as the similarity function for global alignment (denoted Glb). The same settings are used for local alignment as well (denoted Loc). These parameters were selected based on the result of a small tuning experiment that we have applied for different parameter settings of Glb and Loc but not reported here due to space restrictions. The fact that we only tune the parameters of Glb and Loc does not introduce any bias in the results since Cnt, Ham, Jac, and Lev do not have parameters to be set. However, the need for tuning is an impediment since it might be time consuming and not easy in practice. Note that in the case of Lev, we assume the basic Levenshtein definition (with fix parameters as +1 for match and zero for mismatch and gap). Levenshtein algorithms with other weights than what is used in Lev are actually called Global alignment similarity functions in this paper and Glb is one of them, which is tuned for our case.

IV. Related Work

As we discussed in Section II, there have been many studies on code-based test case selection and selection for regression testing which are not applicable in our context. There exist studies regarding similarity-based selection, minimization, and prioritization for code-based testing. However, model-based test case selection using a similarity function has not been a focus of many studies in the literature though many ideas from code-based selection can be adapted to MBT. For example the authors in [16] use a bit vector encoding for some code features (e.g. statement coverage) and Hamming Distance to measure diversity. In [17] test cases are encoded again as bit vectors for some basic block coverage in source code (e.g., statement coverage) but this time the Euclidian distance is used to measure diversity. In [18] the authors use Jaccard Index on a set of covered statement and the work in [19] applies Levenshtein on a sequence of memory operations. In [20] authors use the whole test script as their encoded test case and apply Hamming, Euclidian, Manhattan, and Levenshtein distance on it. However, this encoding is not very effective when the test script contains a great deal of irrelevant platform dependant information, which is usually the case in industrial systems.

STCS techniques for MBT are proposed in [4] and our initial work [5]. Both studies use Cnt as their similarity function but the work in [4] uses transition-based encoding whereas we employ the trigger-guard-based encoding. In [5] we implemented the three encodings explained in Section II.B (state, transition, and trigger-guard-based) and compared their effectiveness in terms of average FDR. The results showed that trigger-guard is the best encoding among them. Using it with the Cnt similarity function and a GA as a selection algorithm, we significantly increased the effectiveness of the current selection techniques such as random, coverage-based, and the transition-based approach (the only reported STCS in MBT [4]). In this paper, we further improve our approach in [5] by using the same encoding (trigger-guard-based) and selection algorithm (the GA) but a better similarity function than Cnt. We compare different similarity functions introduced in Section III in terms of their *FDR* on an industrial case study and also discuss the cost of each function. The practical benefits of our proposed approach compared to other alternatives are also reported.

V. Empirical study

In this section, we investigate the effect of similarity functions on the fault detection ability of STCS techniques by applying them on an industrial case study. We also compare the results of the best STCS approach with random and coverage-based selection techniques.

A. *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) [6]. 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. Since the focus of this paper is on improving test cases selection rather than generation, faults which cannot be caught by the original test suite is not of interest. 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 TRUST [1] as an input model and executable test cases were automatically generated. Note that 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 do not need to run the test path several times with the different input data and we have only one test case per test path and the FDR of a test path is equal to the FDR of the corresponding test case.

The original test suites which selections are applied on is generated by TRUST using All-Transitions coverage. 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.

B. *Experiment design*

In [5] we showed that trigger-guard-based encoding is by far more effective than the other alternatives for SMBT (transition-based and state-based). Also, we showed that the improvement yielded by GA compared to Greedy search was significant. Therefore, to evaluate different similarity functions we use the best encoding and selection technique based on our previous study. Our research questions in the current paper can be summarized as follows:

RQ1. What is the most cost-effective similarity function for similarity-based test case selection in SMBT?

RQ1.1 Which similarity function (among set and sequence based functions) is more effective in terms of FDR?

RQ1.2 Which similarity functions (set or sequence based functions) are less expensive in terms of execution cost?

RQ2. In practice, how much test case execution resources do we save by using the best STCS compared to random selection and coverage-based selections?

To account for the randomness of FDR results, which exists for all selection algorithms, we run 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 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 large test suites. Furthermore, for large sample sizes, all selection techniques will usually be as good as random selection which typically detects most faults. We have performed non-parametric (Mann-Whitney) statistical tests, using a significance level $\alpha = 0.05$, to compare the FDR medians of the proposed and alternative selection techniques. Non-parametric tests are more robust than a parametric test (e.g., the *t*-test) when there are strong departures from normality and they do have enough statistical power for the sample size we deal with in this study (100 observations). In addition, we provide FDR means, standard deviations, and distributions as Boxplots over different runs for the six smaller sample sizes (10 to 60), where differences among techniques are more visible.

The measures that we use for comparing the effectiveness of different techniques are defined in [5] as follows:

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 [7] 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 $0 \le AFDR_m^\gamma(\Gamma) \le 1$. 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, and therefore 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 when selection techniques are compared with respect to their reduction in cost while ensuring a given fault detection rate.

The three measures above are complementary and help interpreting the FDR from different angles. The experiments have been conducted on a PC with Intel Core(TM)2 Duo CPU 2.40 Hz and 4 GB memory running Windows 7.

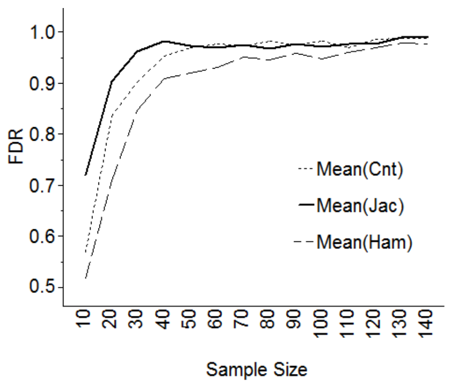
C. Experiment results

In this section we answer research questions RQ1 and RQ2 based on our case study.

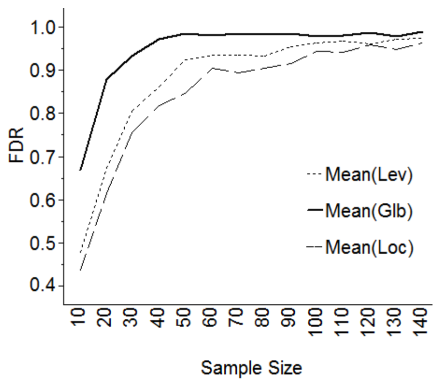
1) Experiment Results for RQ1

We start with RQ1.1 and first compare the effectiveness of set-based and sequence-based techniques separately and identify the best function in each class. We then compare the best set-based similarity versus the best sequence-based function. Figure 1.a shows the average FDR of the three set-based functions introduced in Section III.A ($\rho(i)_{Cnt}$, $\rho(i)_{Jac}$, $\rho(i)_{Ham}$). The results show that Jac has the largest average FDR and Ham the smallest one for almost every sample size and especially so for smaller sample sizes. An overall comparison of the curves also suggests that Jac fares better than Cnt and Ham. ($AFDR_{140}^{10}(Jac) \cong 0.95$, $AFDR_{140}^{10}(Cnt) \cong 0.93$, $AFDR_{140}^{10}(Ham) \cong 0.89$). Using Jac is also better in finding faults with fewer test cases as for example $\min_{90}(\mathrm{Jac}) \cong 20$ (~7% of the test suite) whereas $\min_{90}(\mathrm{Cnt}) \cong 30$ (~11% of the test suite) and $\min_{90}(\mathrm{Ham}) \cong 40$ (~14% of the test suite). Table 1 contains the FDR means and standard deviations of the three functions over 100 runs for various sample sizes. Mann-whitney U-tests shows that Jac median FDR is significantly higher than those of Cnt and Ham, for sample sizes less than 50. For sample sizes between 50 and 140, Jac and Cnt show similar FDR results, which are significantly higher than the FDR results for Ham. Looking at Boxplots in Figure 2 and the standard deviations in Table 1 however suggests that Jac is a better option since it shows less variance for sample sizes above or equal to 30. For sample sizes higher than 140 (50%), all techniques' FDR quickly converges to 1.0.

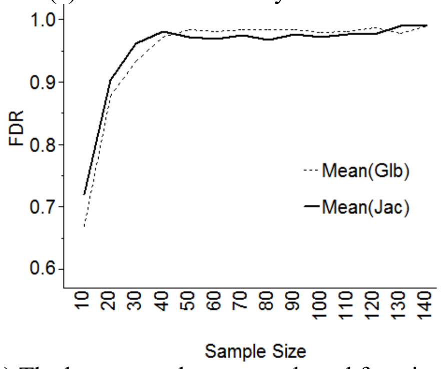
The most plausible reason explaining the above results is that although all three algorithms consider the number of identical elements in the inputs, Ham only reports this value without any normalization. Jac and Cnt, however, normalize the number of identical elements with respect to the total elements in both inputs, which makes the similarity value more precise. For example, let A, B, and C be three input sets. A and B are identical both containing one member *x.* On the other hand C contains three members *x, y,* and *z.* Therefore, a good similarity function should assign higher similarity value to (A,B) than (A,C). Since the number of identical elements in both pairs (A,B) and (A,C) is one, Ham(A,B)=Ham(A,C)=1 whereas Cnt(A,B)=Jac(A,B)=1 but Cnt(A,C)=0.5 and Jac(A,C)=0.34. Therefore, Jac and Cnt are more precise than Ham. Comparing Jac and Cnt, we notice that both use the same information (number of identical and different elements in the input sets). Assume



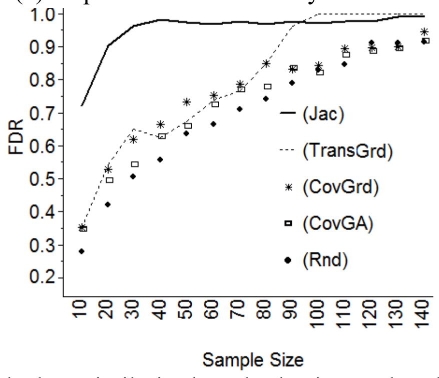
(a) Set-based similarity functions



(b) Sequence-based similarity functions



(c) The best set and sequence-based functions



(d) The best similarity-based selection vs. baselines

Figure 1 The average FDR of different selection techniques for sample sizes 10 to 140

the number of identical elements in two inputs A and B is S and the number of different elements is D. Then Cnt(A,B)=S/(S+D/2) and Jac(A,B)=S/(S+D). Theoretically, none is preferable to the other but our case study is showing that Jac, which normalizes the similarity value by treating S and D the same, is more effective in finding faults than Cnt which gives more weight to identical elements (S) than different ones (D).

Figure 1.b shows the average FDR of three sequence-based selection techniques, introduced in Section III.B ($\rho(i)_{Lev}$, $\rho(i)_{Glb}$, $\rho(i)_{Loc}$). Not surprisingly Glb performs better than Lev. With sample sizes less than 130, Glb is always significantly more effective in terms of FDR (based on Mann- Whitney U-test) since it is basically a tuned version of Lev. However, Loc with the same settings as Glb is much less effective. A plausible reason is that this algorithm is designed for long sequences in bioinformatics, where aligning the whole sequence results in very bad scores. Therefore, they align the sequences locally, which is not as precise as globally aligning them. However, in our case where the average and maximum length of test paths is 5 and 7, respectively, Glb performs better. Comparing the overall curves shows clear differences ($AFDR_{140}^{10}(Lev) \cong 0.88$, $AFDR_{140}^{10}(Glb) \cong 0.92$, $AFDR_{140}^{10}(Loc) \cong 0.85$). In terms of finding more faults with fewer test cases, Glb is significantly better than other sequence-based similarity functions. For example, $\min_{90}(\text{Lev}) \cong 50$, $\min_{90}(\text{Glb}) \cong 25$, $\min_{90}(\text{Loc}) \cong 60$. Furthermore, Lev and Loc show high variance (Table 1 and Figure 2), which makes them very risky to use. For example, even with a large sample size like 110, 10% of the 100 selections using Loc result in an FDR below 0.6 whereas Glb, even with sample size 20, ensures that FDR > 0.6 with a confidence over 90%.

In Figure 1.c the best sequence-based (Glb) is compared with the best set-based (Jac) similarity function. From average FDR's point of view, for sample sizes less than 50, Jac performs better than Glb. In addition, an overall comparison of the curves shows a similar performance ($AFDR_{140}^{10}(Glb) \cong 0.92$ vs. $\text{AFDR}_{140}^{10}(\text{Jac}) \cong 0.95$) and a similar results for variance comparisons (Table 1 and Figure 2). However, the differences are not practically significant in most cases. On the other hand, Jac is from a practical standpoint easier to use since it does not require any parameter settings, whereas weights and penalties in Glb require tuning. Therefore, based on these results, we suggest using Jaccard Index as similarity function in STCS.

Answering RQ1.2 we compare the cost of different similarity functions both in terms of computational complexity and the actual time required for the similarity calculation. We notice that set-based measures are less expensive (O(n+m)) than sequence-based measures (O(n*m)), where n and m are the size of two test cases being compared represented as sets of trigger-guards. In terms of the actual time spent for the calculation, set-based measures required around 0.5 seconds in average for building the similarity matrix (filled with 39340 similarity values between all pairs of test cases in the test suite), whereas sequence-based measures require more than 3 seconds to build such matrix. These results also suggest that set-based measures are less expensive. Therefore, we suggest Jaccard Index, given its low cost, high effectiveness, low variation, and ease of use.

2) Experiment Results for RQ2

We compare our suggested selection technique (Jac) with random selection (Rnd), coverage-based Greedy selection (CovGr), coverage-based GA selection (CovGA), and the state of the art in STCS [4] (TransGr). TransGr uses a transition-based encoding, a Counting similarity function, and a Greedy search for selection. Note that Jac refers to a STCS which uses trigger-guard-based encoding, Jaccard Index as similarity function, and a GA for selection. Figure 1.d shows all average FDRs for different sample sizes for all the techniques. The improvement we get using our technique is clearly visible from the graph and is confirmed by Mann-Whitney U-tests, for sample sizes less than 90. For example, for sample size 30 (~10% of the test suite), we get a 50% improvement from the best alternative technique (CovGrd). The results get even more interesting when we see that the best improvements are on the smaller sample sizes (less than 30% of the test suite), which are more likely to be used in practice. The overall comparison of curves also show large differences ($AFDR_{140}^{10}(Jac) \cong 0.95$, $AFDR_{140}^{10}(TransGr) \cong 0.80$, $AFDR_{140}^{10}(CovGr) \cong 0.76$, $AFDR_{140}^{10}(CovGA) \cong 0.7$, and $AFDR_{140}^{10}(Rnd) \cong 0.69$). As we have mentioned, the minimum number of test cases required for Jac to yield an average FDR above 0.9 is 20 ($\min_{90}(\text{Jac}) \cong 20$ (or ~7% of TS) whereas the best alternatives require at least 85 test cases ($\min_{90}(\text{TransGr}) \cong 85$ or ~30% of TS), thus implying a near 77% reduction in cost. Note that, for sample sizes larger than 100, the mean FDR of TransGr is 1.0 whereas the mean FDR of Jac is below 1.0. The most plausible reason is that Jac uses the GA with a 175ms stopping criterion, which is a very short time for exploring the solution space for large sample sizes. Therefore, among these techniques, the best for yielding 100% FDR with minimum number of test cases is a GA with longer stopping time (e.g., using 1 sec instead of 175ms, Jac can find all faults for sample sizes less than 30). Given the small execution times involved, this has no practical consequences on the applicability of the GA.

The other interesting observation from Figure 2 and Table 1 is the confidence that we gain by using our approach rather than coverage-based selection, random selection, or even the best existing STCS approaches. For example, looking at results for sample size 40 in Figure 2, we see that 90% of the 100 runs of our approach resulted in a median FDR equal to 1.0, while 75% of all runs, for all the alternative approaches (Rnd, CovGrd, CovGA, and TransGrd), yield a median FDR below 0.80. These results strengthen further our confidence in recommending Jac to support SMBT (and in general MBT) in practice.

Analyzing the cost of STCS compared to alternatives, we consider the actual selection time spent by each technique, since no better measure is applicable in our context. For example, the number of fitness evaluations in GAs, a better alternative in some cases, is not applicable to CovGr and Rnd. We use 175ms as stopping criterion for the GA, which seems unfair given that CovGr only requires on average one tenth of this time and Rnd less than 1 ms. However, CovGr and Rnd could not be improved even with increased execution time. Moreover, stopping the GA exactly at the execution time used by CovGr, still improves the FDR though the improvement is not practically significant. From a practical standpoint, all these differences are anyway negligible as 175ms, even when considering the overhead of the similarity matrix creation (in average 500ms for Jac), is very small compared to the actual test case execution time (which is in the range of minutes). In cases where the number of test cases is much larger than in our case study, our conclusions would still hold as both the time of executing test cases and computing similarities would increase, the latter still being negligible. Overall, in order to minimize the overall testing effort, we recommend the use of Jac over existing alternatives.

D. Discussion on validity threats

This study was conducted according to recently proposed guidelines for conducting empirical studies in search-based testing [21]. In terms of the construct validity of our measures, effectiveness (FDR) is based on a set of real faults, as explained earlier, that we used to create mutant programs. Comparing the cost of different similarity functions we considered the computational complexity of their implementations along with their actual time consumptions to gain a more precise understanding of their relative cost. The cost discussion on different selection techniques was not practically interesting in our case because the difference between the execution time of different techniques is negligible compared to even one test case execution time (less than a second compared to minutes). However, for very large test suites with faster test case executions, the differences among selection techniques may no longer be negligible compared to test execution time. However, in most cases, we expect the selection time to be negligible compared to the total reduction in test execution time (time required for executing all excluded test case). The exact threshold above which a selection technique will no longer be cost-effective depends on the test suite size, the percentage of selection, and the average test case execution time. Note that, in our implementation of STCS algorithms, the similarity matrix is created beforehand and kept in memory. This creates an initial overhead and will generate a memory problem for large test suites. The other option which may be even quicker (depending on the number of distinct similarity evaluations that GA requires during its execution and the matrix size) is the on-demand calculation of similarities. In addition, the most used similarities may be cached. Except for sequence-based similarity functions (which implementation is taken from [13]) we implemented the other similarity functions and search techniques and strived to achieve the same level of optimization. Our proposed similarity function (Jac) does not require any tuning but the parameter tuning for Glb and Loc, which is done with a small experiment on a small sample set might not be optimal This means that it is in theory possible to

TABLE 1 THE MEAN FDRS (HIGHEST VALUES PER SAMPLE SIZE ARE IN BOLD) AND THEIR STANDARD DEVIATIONS PER SAMPLE SIZE OVER 100 RUNS.

Selection technique		FDRs per sample size											
		10		20		30		40		50		60	
		Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std
Set-based	Jac	**0.72**	0.14	**0.90**	0.11	**0.96**	0.07	**0.98**	0.05	0.97	0.06	0.97	0.07
	Cnt	0.57	0.18	0.84	0.14	0.90	0.12	0.95	0.08	0.96	0.07	0.97	0.06
	Ham	0.52	0.14	0.71	0.14	0.85	0.14	0.91	0.12	0.92	0.11	0.93	0.10
Sequence-based	Glb	0.67	0.14	0.88	0.12	0.93	0.08	0.97	0.05	**0.98**	0.05	**0.98**	0.05
	Lev	0.48	0.16	0.67	0.14	0.80	0.14	0.86	0.12	0.92	0.10	0.93	0.09
	Loc	0.44	0.13	0.61	0.14	0.76	0.13	0.82	0.13	0.85	0.12	0.90	0.12
Baselines	TranGr	0.35	0.13	0.54	0.13	0.65	0.14	0.62	0.15	0.67	0.14	0.74	0.13
	CovGr	0.35	0.14	0.53	0.13	0.62	0.13	0.67	0.13	0.73	0.14	0.75	0.15
	CovGA	0.35	0.14	0.50	0.15	0.545	0.17	0.63	0.16	0.66	0.19	0.72	0.15
	Rnd	0.28	0.16	0.42	0.18	0.50	0.16	0.56	0.16	0.63	0.19	0.66	0.18

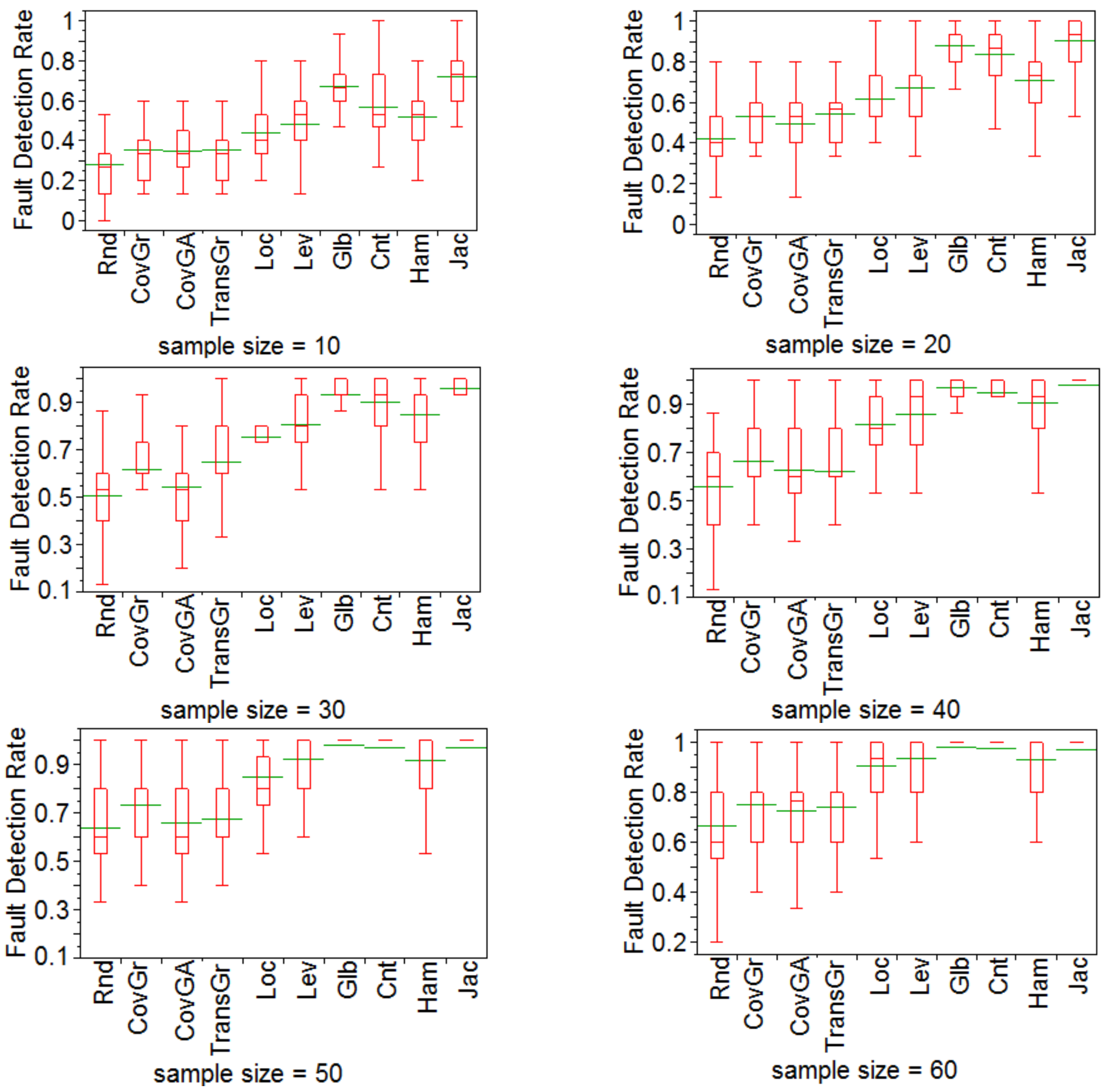


Figure 2 FDR (y-axis) Boxplots for different selection techniques (x-axis) for sample sizes ranging from 10 to 60 by intervals of 10 over 100 runs. The Boxplots show the 10th, 25th, 50th, 75th, and 90th percentiles and means.

obtain a better FDR than Jac using an optimal Glb or Loc. However, this tuning, in general, is not easy to apply in practice and entails extra cost.

One hundred independent runs were performed for each selection technique to account for random variation and obtain a sufficient number of observations to report means, medians, and standard deviations. We used the non-parametric Mann-Whitney U-test for independent samples to check the statistical differences in FDR across selection techniques and we are thus not relying on any assumption. We also discussed about practical significance by looking at the magnitude of the differences between FDR (percentage of improvement) and cost (actual time) of different techniques. Our results rely on one industrial case study using a set of real faults. Though such realistic studies are rare in the research literature and very time consuming, it must be replicated on other systems and sets of faults. However, as discussed earlier, the system used here is typical of a broad category of industrial systems: control systems with state-dependent behavior.

VI. Conclusion and future work

In the context of embedded and telecom software, among many other examples, system testing must often occur on realistic infrastructure and test networks involving limited access time and entailing significant costs. Though Model-Based Testing (MBT) has been found to be an interesting solution in practice, on typical industrial models, the number of test cases generated is still very large. In addition, for many systems, automatically generating oracles from models is very difficult or impossible. In such cases, test cases should be evaluated manually, greatly increasing the cost of test execution and analysis. In this paper, we investigate ways to select an affordable subset with maximum fault detection rate by maximizing diversity among test cases with respect to a similarity measure. In the context of state machine-based testing, a common but specific type of MBT, we used a trigger-guard-based encoding for test case representation and proposed six different similarity measures. A Genetic algorithm was used for optimizing the selected subsets for each measure and their fault detection rates were compared. Applying the techniques on an industrial case study, we showed that using Jaccard Index to measure the trigger-guards similarity of the respective test paths yields a subset of the test suite with the best fault detection rate. Comparing the results of our best proposal with currently existing approaches such as coverage-based and random selection, and other similarity-based selection techniques, we also showed that we are far more effective than other alternatives for smaller sample sizes (which are more interesting in practice) and can save up to 77% of the test execution cost of state machine-based testing. In the future, we plan to look at the effect of other search techniques and other combination of encodings and similarity functions on similarity-based selections. In addition we will replicate the study on another industrial system to analyze the generalizability of the approach.

VII. References

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

[2] A. P. Mathur, *Foundations of Software Testing*, 1 ed.: Addison-Wesley Professional, 2008.

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

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

[5] H. Hemmati, L. Briand, A. Arcuri, and S. Ali, "An Enhanced Test Case Selection Approach for Model-Based Testing: An Industrial Case Study," in *18th ACM International Symposium on Foundations of Software Engineering (FSE)*, 2010.

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

[7] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test Case Prioritization: A Family of Empirical Studies," *IEEE Transactions on Software Engineering,* vol. 28, pp. 159-182, 2002.

[8] S. McMaster and A. Memon, "Call-Stack Coverage for GUI Test Suite Reduction," *IEEE Transactions on Software Engineering,* vol. 34, pp. 99-115, 2008.

[9] Z. Li, M. Harman, and R. M. Hierons, "Search Algorithms for Regression Test Case Prioritization," *IEEE Transactions on Software Engineering,* vol. 33, pp. 225-237, 2007.

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

[11] P. N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*: Addison Wesley, 2006.

[12] G. Dong and J. Pei, *Sequence Data Mining*: springer, 2007.

[13] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*: Cambridge University Press, 1999.

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

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2 ed.: The MIT Press, 2001.

[16] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge," in *18th ACM International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

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

[18] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse, "Adaptive random test case prioritization," in *25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2009.

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

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

[21] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A Systematic Review of the Application and Empirical Investigation of Search-based Test-Case Generation," *IEEE Transactions on Software Engineering, Special issue on Search-Based Software Engineering (SBSE), in press,* 2010.