Similarity-Based Test Case Selection: Toward Scalable and Practical Model-Based Testing

Hadi Hemmati

Thesis submitted for the degree of Ph.D.

Department of Informatics
Faculty of Mathematics and Natural Sciences
University of Oslo
February 2011

Abstract

The growing complexity and size of software systems, along with the increasing role of software in everyday life, makes verification and validation, and testing in particular, essential in software engineering. High fault revealing power, with minimum cost, is the ultimate goal in software testing. Model-based testing (MBT) targets this goal by automating test generation in a systematic way from abstract models of the software under test. Automation reduces the test generation cost dramatically, but the total testing cost includes the cost of test execution and evaluation as well. In practice, there are limitations in testing budget both in terms of time and testing resources. A testing approach cannot be scalable and practical in large industrial systems, unless it addresses all dimensions of the testing cost. But the systematic test generation nature of MBT potentially results in large test suites with execution costs that far exceed the testing budget. Therefore, a mechanism for adjusting the size of the output test suite in MBT is an absolute necessity to ensure success in industry.

This thesis proposes techniques that minimize the test suite size while preserving (to the maximum extent) its fault detection rate. The proposed family of techniques, called *similarity-based test case selection*, hypothesizes that the more diverse the test cases, the higher the fault detection rate. The thesis initiates with a systematic review on search-based techniques for test case generation, which is a starting point for identifying the potential approaches for search-based test case selection being used in similarity-based test case selection. Finding the most effective ways of defining similarity measures and selection algorithms constitutes the core of the thesis. The best selection techniques among different variants of the proposed similarity-based techniques are identified through rigorous empirical analyses on two industrial case studies. The cost-effectiveness of the approach is also compared to the existing selection techniques in the literature. Then, different influential factors on the effectiveness of the technique are examined through controlled experiments in order to gain insight on the analyzed problem, and to gain confidence in the reliability of the results.

The main contribution of this thesis includes the proposal and evaluation of highly effective *similarity-based test case selection* techniques, which turns out to be extremely beneficial in two industrial contexts (up to 50% reduction in the number of test cases required for detecting the same number of faults as to the current, best alternatives).

Furthermore, the technique showed to be scalable with test suite size and also robust to variations in the fault detection rate of the test suite.

Another contribution is a complementary study on estimating the best size for a test suite based on similarity comparisons among test cases. From a practical standpoint, this is a significant contribution to increase the usability of the proposed techniques, since testers are no longer required to select an arbitrary test suite size.

In conclusion, *similarity-based test case selection* showed promising results on two industrial case studies with respect to minimizing the testing cost in MBT. The proposed technique is far more effective than existing techniques. It helps make MBT applicable on larger systems by adjusting the output test suite according to test budget. This research has therefore the potential to significantly impact how MBT is performed today.

Acknowledgement

First and foremost I would like to thank my supervisors Lionel Briand and Andrea Arcuri. Lionel, with his invaluable knowledge and expertise in software engineering research, not only helped me in all steps of my PhD but also taught me how to be a good researcher. He was an excellent supervisor, supportive boss, and at the same time a good friend. I learned a lot from him and I owe him. Though Andrea became my supervisor only in the last stages of my PhD, but if it was not his expertise in search-based software testing, I could not finish my thesis by now. I learned a lot about randomized algorithms and statistics from him. There was not any detail that I could discuss with him and he could not constructively comment on it. I wish to thank Erik Arisholm who supervised me in the first stages of PhD. Apart from priceless empirical software engineering knowledge that I learnt from him, I especially enjoyed his patience during discussions.

I would like to thank my officemate for nearly four years, Rajwinder Panesar-Walawege, for both interesting everyday academic or non-academic discussions and for helping me adjust myself to the new culture and living style by her great multi-cultural background. I also wish to thank a lot my other best friends in Simula, Shaukat Ali and Aiko Fallas Yamashita. We spent a great time together at work and out. I should also thank Razieh Behjati, Mehrdad Sabetzadeh, and Shiva Nejati from Approve group at Simula who helped me by smart comments on different steps of my work.

Special thanks to Marius Liaaen from Tandberg AS (now part of Cisco) for his helps on conducting studies at Tandberg. I truly appreciate the time and resources that he and others in his team spent on providing the domain knowledge and helping me on experimenting with their systems. I am thankful to Simula Research Laboratory and the Simula School of Research and Innovation for accepting me as a PhD-student, and for the excellent environment they offered for research. I thank all the people in the Approve and Prepare groups for making such an inspiring and enjoyable environment.

Last but not least, I am grateful to my family and friends for their supports. My dear wife, Leili, and my parents who are the most motivating and supportive family in world deserve my special thank.

List of papers

The following papers are included in this thesis:

Paper 1. A Systematic Review of the Application and Empirical Investigation of Search-based Test-Case Generation

S. Ali, L. Briand, H. Hemmati, and R. K. Panesar-Walawege

Published in the IEEE Transactions on Software Engineering (TSE), vol 36, no 6, pp. 742-762, 2010

Paper 2. An Enhanced Test Case Selection Approach for Model-Based Testing: An Industrial Case Study

H. Hemmati, L. Briand, A. Arcuri, and S. Ali

Published in the proceedings of the 18th ACM International Symposium on Foundations of Software Engineering (FSE), pp. 267-276, 2010

Paper 3. An Industrial Investigation of Similarity Measures for Model-Based Test Case Selection

H. Hemmati and L. Briand

Published in the proceedings of the 21st IEEE International Symposium on Software Reliability Engineering (ISSRE), pp. 141-150, 2010

Paper 4. Reducing the Cost of Model-Based Testing through Test Case Diversity

H. Hemmati, A. Arcuri, and L. Briand

Published in the proceedings of the 22nd IFIP International Conference on Testing Software and Systems (ICTSS), formerly TestCom/FATES, pp. 63-78, 2010

Paper 5. Empirical Investigation of the Effects of Test Suite Properties on Similarity-Based Test Case Selection

H. Hemmati, A. Arcuri, and L. Briand

To appear in the proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation (ICST), 2011

Paper 6. Achieving Scalable Model-Based Testing Through Test Case Diversity

H. Hemmati, A. Arcuri, and L. Briand

Submitted to ACM Transactions on Software Engineering and Methodology (TOSEM), 2010

The six papers are self-contained. Therefore, some information is repeated. There are also some differences in the abbreviations used in the papers.

My contributions

For all papers except the first paper, I was responsible for the idea, the study design, implementation, data collection, analysis, and writing. My supervisors contributed in all phases of the work. In paper 1, all authors equally contributed in the study. However, I was the responsible person for the paper.

During my PhD study, I also contributed in two other articles which are not included in this thesis.

A. Model Transformations as a Strategy to Automate Model-Based Testing - A Tool and Industrial Case Studies.

S. Ali, H. Hemmati, N. E. Holt, E. Arisholm, and L. Briand

Technical Report 2010-01, Simula Research Laboratory

During my PhD, I contributed in developing a model-based software testing tool and writing its technical report. The tool employs model transformation to automate test case generation. The first three authors were responsible for the work and equally contributed on the development of the tool and writing the report.

B. Modeling Robustness Behavior Using Aspect-Oriented Modeling to Support Robustness Testing of Industrial Systems, Submitted to Software and Systems Modeling,

S. Ali, L. Briand, and H. Hemmati.

Submitted to Journal of Software and Systems Modeling (SOSYM), 2010.

I also have contributed in a work on modeling non-functional requirements using aspect modeling, which is not included in this thesis. The study introduces a UML profile for aspect state machine, a weaver tool, and a methodology for non-functional requirement modeling using aspect state machines. I was the last contributor in this work and my contribution is on developing the idea and modeling the case study.

Contents

Summary

1 Introduction

Software is being incorporated into an ever-increasing number of systems including embedded and safety critical systems, and hence it is becoming increasingly important to thoroughly test these systems. One challenge in software testing is the effort involved in creating a test suite that will systematically test the system and reveal latent faults in an effective manner [1]. In recent years, systematically deriving test cases from a behavioral model of a system, Model-Based Testing (MBT), has attracted an increasingly wide interest from industry and academia. This interest can be seen from the many academic studies [2-8] and industrial projects [9-12] on MBT. This suggests that there is an increasing awareness of the benefits offered by MBT compared to other testing methodologies. However to make MBT a practical solution, it should be scalable to large industrial systems and assume realistic test budgets.

One of the important problems regarding scalability is the relationship between the cost of the test technique and the complexity of realistic systems. The MBT cost can be divided into two categories: (1) test case generation and (2) test case execution. Test case generation cost is referred to the modeling cost plus the time and resources required by an MBT tool for generating executable test cases from the model. On the other hand, the test case execution cost is the time and resources required for execution and evaluation of the generated test cases. This thesis concentrates only on the latter category since test suite execution is an important factor for the applicability of any test case generation technique, though it is far less investigated in the literature than test case generation cost [13-15].

Although test case execution cost is directly related to the test suite size (the number of generated test cases which are going to be executed), MBT does not provide any special support for managing the size of its output test suites. However, such support seems necessary because test suites generated by MBT approaches tend to be very large and they get exponentially larger with increasing model size (due to large software under test, or SUT). Furthermore, the matter gets even worse when testing is semi-automated (e.g., automatically generating oracles may be very difficult or impossible, such as in a subjective quality assessment of a video stream) and human effort is necessary for the evaluation of the test results. In addition, what usually happens at system level testing of industrial SUTs is that test cases are executed on real hardware platform and network. This has a huge effect on the total testing cost because (a) test case execution time is much longer than for desktop software, and (b) test case execution often requires physical testing resources (e.g., specific assigned machines and restricted-access network) whose availability is limited.

During this project, applying MBT on two SUTs of different sizes and from different application domains showed that the cost of executing test suites generated using MBT (given standard coverage criteria) can be by far higher than the resources (i.e., the so called testing budget) available for testing the SUT. In some cases, it is not even feasible to run all test cases by the project deadlines. Therefore, lowering the cost of test suite execution, both in terms of time and resource, is a crucial success factor for MBT.

Solving this problem, in this thesis, a novel approach for test case selection in MBT is investigated. The approach, referred to as similarity-based test case selection (STCS), selects the least similar test cases from the test suite for a given test selection size. The underling idea is that diversifying selected test cases will help to detect more distinct faults. In addition, the thesis explores how to define similarity measures and how to minimize the overall similarity among selected test cases. STCS is also empirically evaluated on two industrial cases studies coming from two different companies located in Norway. The results show that it reduces the size of the test suite while nearly preserving the original test suite's fault detection rate (FDR). This results in test suites with almost the same quality (in terms of FDR) but much less overall execution cost (because less test cases need to be executed), which consequently results in a more scalable MBT approach in practice.

The thesis is structured in two parts***:***

Summary: This part of the thesis consists of seven sections. After this Introduction, in Section 2, a short background (required for understanding the included papers) on model-based and search-based testing, plus test case selection techniques, is provided. In Section 3, the core idea of the thesis, STCS, is introduced. Section 4 explains the research methodology and Section 5 provides an overview of the included papers. In Section 6, suggestions for future work are presented. Finally, Section 7 concludes the thesis.

Papers: The rest of the thesis consists of six published, accepted for publication, and submitted papers in international journals and peer-reviewed conferences.

2 Background

In this section, background information is provided as a context of the papers that are included in the thesis. Section 2.1 introduces MBT, the approach that is used in this thesis for generating test cases from UML state machines, and the MBT tool that has been developed during this project. Section 2.2 presents a brief introduction to search-based software testing (SBST), which is used for test case selection in STCS. A more comprehensive review of SBST techniques is reported in Paper1. Finally, in Section 2.3, an overview of different test case selection techniques is presented.

2.1 Model-based testing

Model-based testing (MBT) is defined as "the generation of executable test cases from behavior model of the system under test" [2]. A test case specifies the present state of the SUT and its environment, the test inputs and conditions, and oracle information [16]. An example of a test input is a sequence of functions or method calls and their input parameters. Oracle information identifies properties that should be true after the execution of the test case. Several strategies can be considered to implement efficient oracles [16].

Since many systems, such as embedded real-time systems [17], telecommunication systems [18, 19], and multimedia systems [20], exhibit state-driven behavior, traditional *Finite State Machine* and its extensions are commonly used to model such behavior. *Finite State Machine* is a graph-based representation of the SUT's behavior, where a finite number of graph *nodes* represent the states of the system and a finite number of graph *arcs* represent transitions of the system from one state to another [2]. Traditional *Finite State Machines* have some limitations. For example, they cannot model software systems with concurrent behavior. Therefore, the Unified Modeling Language (UML) standard introduced concurrency in UML state machines by defining composite states with two or

more regions [21]. When modeling complex software systems with *Finite State Machines*, the number of states and transitions can grow exponentially with system size. This can be handled by UML state machine features for modeling sub-machines. Many commercial and academic tools (e.g., [20, 22]) support the modeling of UML state machines.

Several well-known MBT tools have been developed in recent years, such as *TDE/UML* (*Siemens*) [11], *SpecExplorer* (Microsoft) [12], *IBM Rational Functional Tester* [10], and *Qtronic* [23]. In this thesis, an MBT tool called *TRansformation-based tool for Uml-baSed Testing* (TRUST) is used, which has been developed (by the author and some other students) as a part of the *Automated Model-based Testing of State-driven Systems* (AMOS) project [24]. The main motivation for developing TRUST was having an easily extensible tool on which to base our research. Basically, TRUST is used in this thesis as an infrastructure that the proposed similarity-based test case selector component is deployed on. However, the test case selector component can be potentially integrated with any UML state machine-based MBT tool. TRUST accepts UML state machines containing concurrency and hierarchy as the input model and generates executable test cases along with oracles. It is integrated with *IBM Rationale Software Architect* (RSA) [22] as modeling tool and applies a series of model-to-model (in *Kermeta* [25]) and model-to-text (in MOFScript [26]) transformation rules on the input model to generate the final test scripts. Figure 1 illustrates this transformation-based approach for MBT.

The general process of MBT that is used in this thesis starts with modeling the SUT and making it ready for test generation, e.g., by enriching a UML state machine with state invariants, which are used for test oracle generation. TRUST, specifically dedicates one step of model-transformation for preparing test ready models. Since in this project, the input models (UML 2.0 state machines) had hierarchy and concurrency, they were first flattened, to be able to apply classic, graph-based coverage criteria [27]. The flattening is automatically done by TRUST. The next step is deriving abstract test cases (ATCs) from the test ready model (flattened state machines in TRUST) according to a test strategy, which is typically defined based on a test model and coverage criteria (e.g., all states) to guide its traversal [1]. ATCs, like concrete test cases, contain the present state of the SUT and its environment, the test inputs and conditions, and the expected results, but expressed at a higher level of abstraction. Finally, executable test cases are generated by adding all platform dependent information to ATCs and mapping abstract information (e.g., triggers and state variables) of the ATC to the actual executable information (e.g. method names and system variables) in the test script.

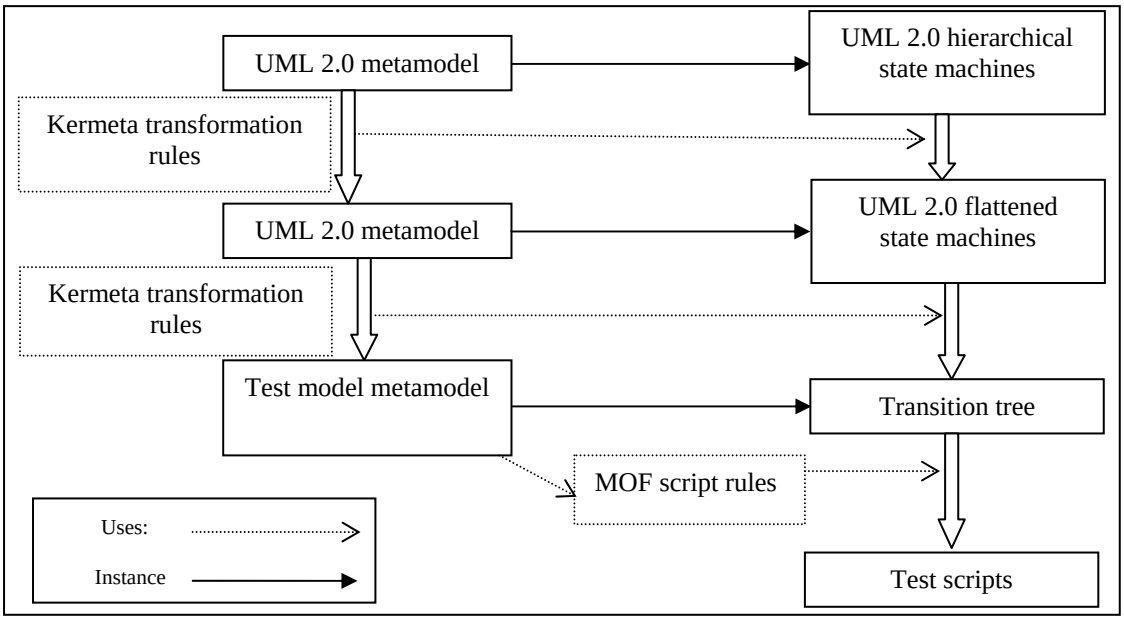


Figure 1 Model transformation-based approach for test case generation

Each component of TRUST implements one set of transformation rules (e.g., transformation from the test tree to the test cases is done by *testScriptGenerator* in Figure 2). Each component has well-defined interfaces with other components. More specifically, each interface provides output to, or requires inputs from, other components by means of intermediate models. Separation of concerns among components has made each transformation responsible for providing one artifact such as the test model, test data, and test scripts. Therefore, adding a new feature (for example, outputting test scripts in a new language) can be achieved by writing a new set of transformation rules in the component that provides the corresponding artifact, without affecting the other components. Model transformation languages provide the developer direct support for navigating, creating, and manipulating a model, based on its metamodel. Generally, the transformation rules are relatively compact and easy to read, write, and change.

Figure 2 depicts the architecture of TRUST, which consists of five components. The first component, s*tateMachineFlattener*, takes the UML 2.0 state machine metamodel and one instance of it, which may contain hierarchy and concurrency, as inputs. Using a set of model-to-model *Kermeta* transformations, it provides the *flattenedStateMachine* only containing simple states and transitions. The *stateMachine2TransitionTreeTransformer* component takes metamodel of transition tree and a coverage criterion and generates a transition tree conforming to the metamodel by applying the criterion on the *flattenedStateMachine* using another set of *Kermeta* transformations. Finally, the

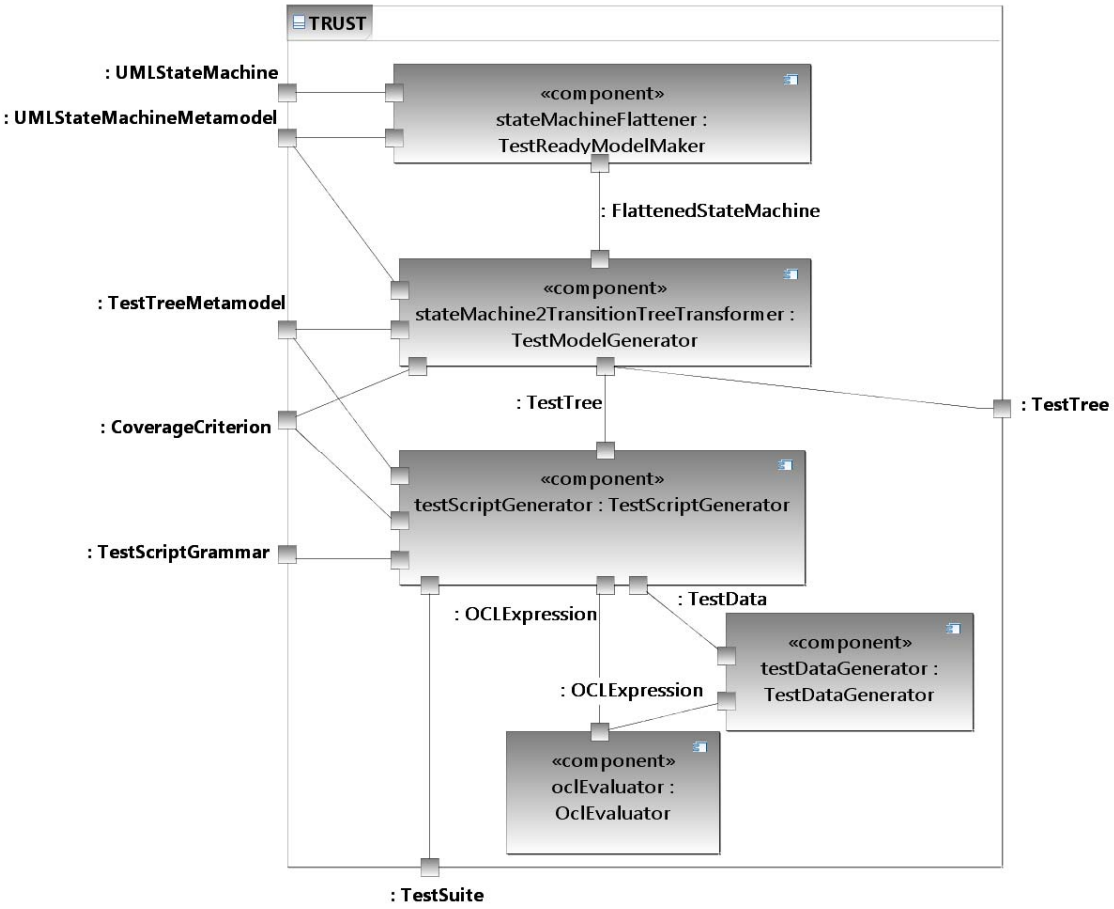


Figure 2 Architecture diagram of TRUST

testScript, which is a test suite of executable test cases (currently supporting Python and C++), is generated by the *testScriptGenerator* component by traversing the tree in MOFScript according to the given coverage criterion. The test paths, which are generated by traversing the tree, are concretized using the input data for the parameters of the transitions operations, generated by the *testDataGenerator* component. Input data are randomly generated based on the parameters type, unless there is a guard on the transition to be covered. In such cases, the constraint is solved by a search-based OCL solver implemented in the *testDataGenerator* (developed by Ali *et al* [28]). EyeOCL [29] is the embedded OCL evaluator in TRUST which is used for search-based data generation and oracle evaluation at run-time.

In TRUST, oracles are automatically generated as part of the *testScript*. Executing test cases, OCL expressions in the state invariants associated with the states must be evaluated. EyeOCL is called during the execution of a test case to evaluate the state invariant. The object model of the SUT at runtime, representing the current state of the system along with

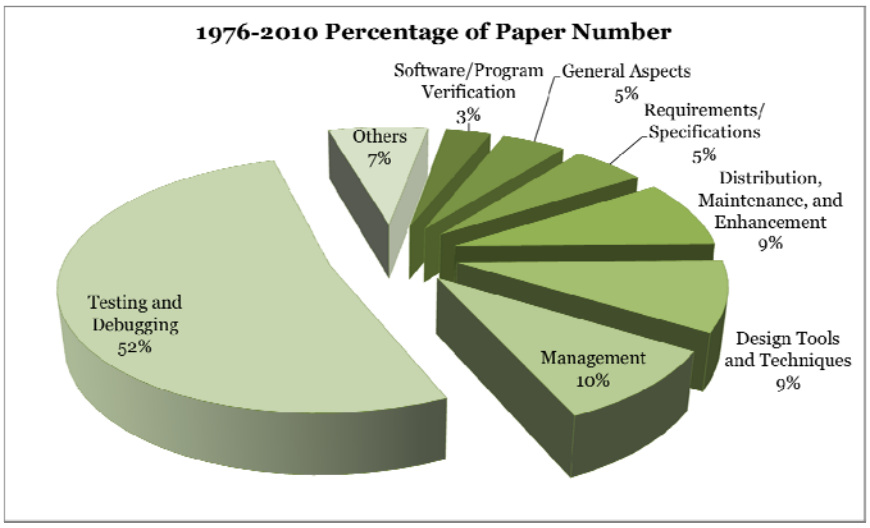


Figure 3 Spread of 763 SBSE papers from 1976 to Nov 30th 2010 based on their application areas activities [34]

the constraint to be evaluated is passed to the OCL evaluator, which in turn returns the *Boolean* result of the evaluation. Querying the current state of the system depends on the implementation of the SUT and the test script language's facility to access the state of the SUT. More detail on the implementation of TRUST can be found in [30].

2.2 Search-based software testing

The application of search and optimization techniques in software engineering, i.e., search based software engineering (SBSE), is a research area that has attracted a lot of attention from academia in the last few years [31]. In fact, search techniques have been very effective in solving many types of engineering problems, and they can be applied in software engineering as well. Many reviews and surveys recently published in this domain [1, 31-33] show many successful results. Based on [34], 763 articles were published from 1976 to November 30th 2010 on SBSE and 52% of the papers (Figure 3) are concerned with the application of optimization techniques to software testing, i.e., search based software testing (SBST). Though SBST targets various problems in verification, and validation, most of the SBST activities till now are focused on test case generation [1, 33]. The main idea of search-based test case generation is that "the set of test cases forms a search space and the test adequacy criterion is formulated as a fitness function for the search technique" [33].

Search techniques are general strategies that need to be adapted to the problem at hand. In the rest of this section, a typical SBST process is explained with an example of

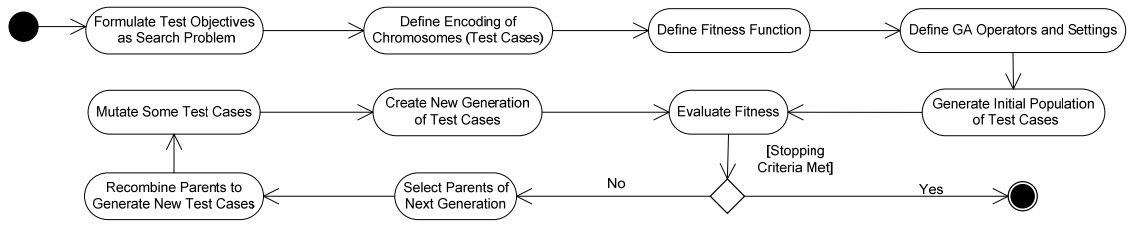


Figure 4 Test case generation using Genetic Algorithm

test case generation using Genetic Algorithm (GA) [35], which is the most used search technique in SBST [1]. Figure 4 depicts the process.

2.2.1 Formulating test objectives and encoding of chromosomes

Representation of a testing problem as a search problem includes the definition of a mapping from the solution space into the search space. This in GA involves defining an encoding for the genes and the chromosomes. The genes are a constituent part of chromosomes. The chromosome encoding is dependent on the kind of problem being addressed and for test case generation it can be, for example, the data for a specific test case.

2.2.2 Fitness (cost) function formulation

Fitness functions are one of the main parts of a search algorithm because they are responsible for guiding it in the search for (near-)optimal solutions. A fitness function is used to evaluate how "good" a candidate solution is. GAs tend to select and reproduce "fitter" solutions, and so discard solutions that are not particularly fit.

2.2.3 Initial population and selection strategies

GA settings can have a huge impact on the efficiency of a GA. The first two choices are on the strategy for making the initial population (e.g. random or a specific strategy [35, 36]) and selecting parents for recombination (e.g. random pairing [36], roulette wheel [37], tournament selection [37], rank selection [38]). The following items explain other operators and settings.

2.2.4 Search operators

Search operators (or recombination operators) are the means by which a search algorithm explores the search space. Though there are guidelines for choosing the operators, but there is not a single best solution for all problems (*No Free Lunch Theorem* [39]). Usually these

operators require tuning to adjust their parameters with respect to a specific problem. The choice of the operators and their tuning make the exploration of the search space in balance with the exploitation of the solution towards the target. Sometimes, based on the encoding of the individuals and the problem, the operators also have to be chosen carefully so that invalid chromosomes are impossible to generate or a rare occurrence [40]. The most commonly used search operators are crossover (for exploration) and mutation (for exploitation) operators. Mutation operators (e.g., bit-flip operator [40]) randomly change genes in chromosomes. Crossover allows individuals (chromosomes) to exchange information (genes). Different types of crossover operators (e.g., single point and two-point crossover operators [41]) are defined based on the number, the locus, and the probability (i.e., rate) of choosing the chromosome for crossover.

After applying operators such as crossover and mutation, the next generation of chromosomes is generated by replacing some of the parents with the generated offspring. Two points should be considered while defining a replacement strategy (e.g., steady-state [41]): How many individuals are retained from the previous population (i.e., not replaced by offspring) and what is the selection strategy for deciding which individuals should be retained.

2.2.5 Elitism

Elitism is a mechanism to ensure that the traits of the fittest individuals are transferred to the next generation. This is achieved by selecting some of the fittest individuals and keeping them as part of the next generation. Note that, elitism is different than a selection strategy since without elitism even the fittest chromosome may be replaced by some probabilities.

2.2.6 Stopping criteria

Selecting stopping criteria for GAs is one of the challenging issues. In practice, if the search target (optimal solution) is known, then one stopping criterion is “stopping after finding the solution”. However, in many problems, the optimum solution is unknown. In addition, the searching resource is limited. Therefore, a maximum searching time is usually set for stopping the search. When comparing different algorithms, other stopping criteria such as “a maximum number of generations” and “stopping when there is no improvement in fitness values” are more preferred, since they help providing more fair and valid comparisons.

In Paper 1, the systematic review on search-based test case generation with a focus on its application and empirical investigation is reported. Since the body of knowledge and empirical results are limited regarding search-based test case selection, this review helped us identify candidate STCS techniques based on a careful analysis of results from search-based test case generation.

2.3 Test case selection

There are two main solutions to overcome the problem of large test suites in MBT. The first approach is using a less demanding coverage criterion to generate fewer test cases. For example, using all-transitions [16] instead of all-transitions pairs [16] can significantly decrease the number of generated test cases. However, this is often not a practical solution as it is difficult to control the number of test cases that way and one cannot ensure that the number of test cases will be below a required threshold [42]. The second solution is to execute only a portion of the test suite. There are three types of techniques introduced in the literature in this regard [43]:

- Test suite minimization that tries to minimize the test suite by removing redundant test cases with respect to a criterion (e.g., code coverage).
- Test case selection that, given a test selection size, tries to select a subset of the entire test suite that maximizes its FDR.
- Prioritization techniques that do not remove any test case but order their execution.

By this definition, the context of the problem falls into the second category (test case selection). However, most of the ideas used in minimization and prioritization can also be applied in selection. Therefore, this thesis focuses on the test case selection problem, but all applicable ideas including also minimization and prioritization are reviewed.

All the above three categories are mostly studied in the context of regression testing, where the goal is to find an optimal subset of the original test suite that guarantees the execution of fault revealing test cases [43]. Most of the regression-based minimization, selection, and prioritization techniques are based on identifying what parts of the system are affected by changes and use information from previous execution of the test cases [43]. In the context of this thesis (i.e., reducing MBT execution cost), there is not any execution information available and, therefore, most regression-based selection techniques cannot be applied.

From another point of view, test case selection techniques use either code-level (e.g., code-based dependency analysis [44] and statement coverage [45-47]) or model-level information [48-50]. However, some of the code-based selection heuristics can also be adapted to MBT, usually by replacing the code-level inputs to analogous model-level information. For example, additional statement coverage [47] can be converted easily to additional transition coverage for state machine-based testing [50].

Regardless of the level of abstraction (code or model based) and the purpose of selection (regression testing or general minimization of the test suite), the existing techniques (minimization, selection, and prioritization) can be categorized into three main classes:

1) Random [51] or semi-random [16] selection and prioritization, where there is no guidance to select/prioritize test cases. Usually it is used as a baseline of comparison, since it is simple approach that can be applied on any test suite, no matter how it has been generated and what information is available.
2) Coverage-based techniques, where we hypothesize that "*the test cases which have more coverage (such as code or model based coverage) are more likely to detect faults*" [43]. Maximizing coverage has been a common practice over the years in minimization, selection, and prioritization [43, 47, 52]. In MBT, coverage is defined at the model level, which can be extracted from ATCs without execution. For example, transition coverage in a state machine [50] can be determined if traceability has been preserved between an ATC and its source state machine. Most coverage-based techniques are re-expressed into optimization problems where the goal is to select the best subset/permutation of test cases to achieve maximum coverage with minimum cost [43].

   For example, a technique presented in [47] uses a Greedy search to select, at every step, the test case that covers the most uncovered statements (additional coverage technique). Similarly, in [53-56] a GA is used to achieve maximum coverage. A more complete survey of coverage-based minimization, selection, and prioritization techniques can be found in [33] and [55].
3) Similarity-based techniques, where we hypothesize that "*the more diverse the test cases the higher their fault revealing capacity*" [52]. This technique is a relatively recent category [48, 57] of selection/prioritization techniques which can be applied in both code and model-based testing. A similarity-based technique maximizes

diversity among test cases with respect to a similarity measure, which requires assigning a similarity value to each pair of test cases and minimizing the average pair-wise similarities between the selected test cases. The next section summarizes the approach of this thesis to STCS as one contribution of the thesis.

3 Similarity-based Test Case Selection in Nutshell

The main contribution of this thesis is to address the problem of scalability in test suite execution in MBT. This thesis proposes and investigates a model-based test case selection technique that makes MBT more practical by allowing testers to adjust the size of output test suites based on time and resource constraints, while preserving the FDR of the original test suite to the maximum extent. The technique, similarity-based test case selection (STCS), is based on selecting the most diverse ATCs generated by applying a coverage criterion on a test model. In other words, the choice of the final concrete test cases to execute is optimized with respect to their pair-wise similarity, given an affordable testing budget. The underlying assumption here is the existence of a correlation between the similarity of test cases (measured by similarity of corresponding ATCs) and their fault detection, which is also studied in the thesis (Paper 4 and Paper 5). An STCS technique is composed of three phases:

- ***Encoding of ATCs***: In this phase, the ATCs are encoded using a sequence of model elements (e.g., states and/or transitions). The choice of elements determines the expected level of precision of the encoding. For example, an ATC represented by a sequence of triggers and guards is encoded at a more precise level than the ATC represented by a sequence of transitions. Recall that ATCs, like concrete test cases, contain information about test input sequences and test oracles. For example, states invariants in ATCs derived from UML state machine can be used to derive test oracles.
- ***Similarity matrix generation***: In this phase, all pair-wise ATC similarities are calculated and saved into a similarity matrix. The similarity between two ATCs is defined based on a similarity function. Eight similarity functions (e.g. Hamming distance [58], Jaccard Index [59], Gower-Legendre [59], Levenshtein [58], and Smith-Waterman [60]) were implemented and are available to use. In addition, the

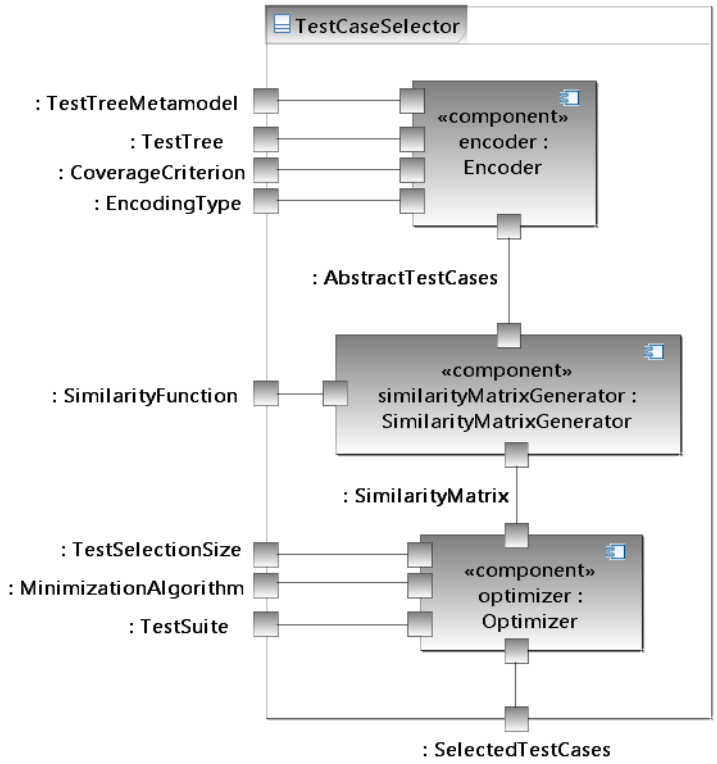


Figure 5 Architecture diagram for the test case selector component in TRUST

effectiveness of different similarity functions has been studied and reported in Paper 3 and Paper 6.

- ***Minimizing similarities***: The last phase of STCS is minimizing the sum of all ATC pair-wise similarities in the selected subset. There are many applicable alternatives, entailing different cost and effectiveness, that were implemented such as Greedy search [61], Agglomerative Hierarchical Clustering [59], Adaptive Random Search [62], and GAs, etc (Paper 5 and Paper 6).

Since there are many possible techniques applicable for each phase, STCS can be seen as a family of selection techniques. Therefore, an STCS variant is made by defining a specific combination of techniques for the three phases. This family of selection techniques have been implemented and integrated with TRUST. The similarity-based test case selector sub-system of TRUST consists of three components corresponding to the three phases of STCS (Figure 5). The *Encoder* component takes the test model (test tree), its metamodel (test tree metamodel), the coverage criterion, and the encoding type as inputs and applies the coverage criterion to the test tree and generates the ATCs corresponding to the encoding type. The ATCs are given to the *SimilarityMatrixGenerator* component which

also takes the similarity function as input. Using the function, a similarity matrix is generated for the ATCs and is given to the *Optimizer* component. Applying a minimization algorithm for a selected size (test selection size), where both the algorithm and size are specified as inputs of this component, the selected ATCs are identified and the corresponding concrete test cases, taken from the test suite, are provided as output.

4 Research Methodology

This thesis relies on the combination of a systematic review, industrial case studies, and simulation-based controlled experiments, which were applied at different stages of the thesis.

Systematic reviews are a means of synthesizing existing research results, in a systematic and unbiased way, regarding a specific research question [63]. They are usually performed to summarize the existing evidence for a particular topic and aid in the identification of gaps in the current research, and thus can form the basis of new research activity [1]. Since a main portion of this study is on the use of search-based techniques on testing (specifically on test case selection), a comprehensive systematic review of the current literature was conducted to collect, classify, and assess existing SBST empirical studies in order to assess the current body of evidence regarding the cost and effectiveness of SBST. Test case selection is not the focus of this review, since there were not enough publications on the use of search techniques for test case selection at the time the systematic review was initiated. Therefore, the focus of the review is on test case generation, since it is the most studied and well-established sub-domain in SBST [31]. Analysis of successful/unsuccessful search techniques in this sub-domain helped us to design better STCSs. In addition, existing surveys and reviews [33, 43] were studied to cover existing works related to test case selection.

Industrial case studies are one of the main means of investigation in this thesis. A case study is “an empirical investigation of a phenomenon in a real-life context, particularly suited when the phenomenon and the context are difficult to separate” [64]. Case studies are necessary for the industrial evaluation of software engineering methods and tools as they provide results obtained in realistic conditions and address scaling problems [65]. In this study, since the cost-effectiveness, scalability, and practicality of the proposed techniques in industry were going to be investigated, the phenomenon under study (STCS)

could not be separated from the context (real faults, realistic models), as the context had a direct impact on the results.

A controlled experiment was also conducted using simulation in Paper 5 to investigate some hypotheses on the effect of different test suite properties on the effectiveness of the selection techniques. Controlled experiments allow us to determine in precise terms how the variables are related and whether cause-effect relationship exists between them. The choice of simulation was due to the fact that conducting industrial case studies is expensive and it was therefore not practical for us to find real software systems satisfying the various combinations of conditions under investigation.

In summary, the thesis started with a systematic review of related works (using search techniques for testing) in Paper 1. Then STCS was introduced and implemented. The next step was conducting case studies on two industrial software systems from two different domains (Papers 2 to 4 and Paper 6). The main research goals were finding whether the proposed technique performed better than other existing selection techniques and if so, how much can be gained in practice in terms of time and other testing resources. In addition, different test suite properties were analyzed using a simulation study in order to analyze how the proposed STCS technique performs under various conditions (Paper 5).

5 Summary of Results

In this section, the included papers are summarized and the key results are presented.

5.1 Paper 1

"A Systematic Review of the Application and Empirical Investigation of Search-based Test-Case Generation", S. Ali, L. Briand, H. Hemmati, and R. K. Panesar-Walawege, in the IEEE Transactions on Software Engineering (TSE), vol 36, no 6, pp. 742-762, 2010.

Through the systematic review we answered the following research questions:

RQ1: What is the research space of search-based software testing?

RQ2: How are the empirical studies in search-based software testing designed and reported?

RQ3: How convincing are the reported results regarding the cost, effectiveness, and scalability of search-based software testing techniques?

Among 450 papers published by the end of 2007 in SBST, 68 papers were included as relevant for the review. The results showed that in 75% of the papers, SBST techniques

have been applied on unit testing. Most papers (78%) focus on structural coverage and the most commonly used algorithm is the GAs and their extensions (73%).

We found out that there is a general lack of rigor in the statistical analysis and reporting of results in most empirical studies reported in the included papers (77% of the studies lack accounting for random variation of the search algorithms or they report incomplete descriptive statistics without any statistical test). Furthermore, most of the papers did not demonstrate the benefits of SBST by comparing it with simpler techniques such as random search. So we concluded that most empirical studies in the context of test case generation using SBST techniques are still not properly conducted and reported for the reader to be able to draw reliable conclusions.

The review also reports that there is a limited body of credible evidence that demonstrates the usefulness of SBST techniques for test case generation. However, the evidence consistently shows that GAs outperform random search in terms for structural coverage. Since the evidence was based on only a few papers and could not be easily generalized, we concluded that more empirical studies must be conducted in the area of SBST, to gain a minimum level of confidence about whether it is a promising option.

To help researchers conduct and report better empirical studies in this domain the paper also provided guidelines in the form of a framework on how to conduct empirical studies in SBST.

5.2 Paper 2

"An Enhanced Test Case Selection Approach for Model-Based Testing: An Industrial Case Study", H. Hemmati, L. Briand, A. Arcuri, and S. Ali, in the proceedings of the 18th ACM International Symposium on Foundations of Software Engineering (FSE), pp. 267-276, 2010.

Our approach to STCS is first introduced by this paper, where three similarity measures and two minimization algorithms are investigated. However, the concept of similarity measure had not yet been divided into encoding and similarity function. The measures were using the same similarity function but with three different encodings. The similarity function used in this paper was adapted from the only existing work on applying STCS in MBT [48] and the minimization algorithms were Greedy search and GA. We had the following research questions in this paper:

RQ1. Which similarity measure is more effective for UML state machine-based test case selection, in terms of FDR?
RQ2. To which extent is using a GA for test case selection more cost-effective (in terms of time spent to find a solution) compared to a Greedy search?
RQ3. To which extent 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 STCS, on a representative industrial case study?

The paper reports that the measure that encodes test cases by their triggers and guards and uses a simple counting function (defined in the paper) has the best performance among all variants of STCS investigated in the paper. The performance is measured by the FDR of the selected test cases for a given test selection size. We also found GA by far more effective than Greedy search. Comparing random and traditional coverage-based selection techniques with STCS, we showed great improvement in terms of FDR. We also reported the practical benefit of our approach by showing the savings that one could get in terms of number of test cases to execute for detecting a certain percentage of faults. Results from our industrial case study showed that the best STCS technique can select only 27% of the test cases generated by MBT while retaining a 100% FDR.

5.3 Paper 3

"An Industrial Investigation of Similarity Measures for Model-Based Test Case Selection", H. Hemmati and L. Briand, in the proceedings of the 21st IEEE International Symposium on Software Reliability Engineering (ISSRE), pp. 141-150, 2010.

In this paper, we differentiated between encoding and similarity function and focused on the effect of similarity function on the performance of STCS. We introduced six similarity functions (three set-based and three sequence-based) and used trigger-guard and GA as encoding and minimization algorithm respectively, based on the results from the previous work (Paper 2). The research questions of this paper were:

RQ1. What is the most cost-effective similarity function for STCS?
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?

The results based on an industrial case study (on the same SUT as Paper 2) showed that Jaccard Index among set-based functions and Needleman-Wunsch [60] algorithm among sequence-based similarity functions are comparable and the most effective functions in terms of FDR of the selected test cases. However Jaccard was a priori more interesting since it was simpler to use (no tuning needed) and faster.

We later compared this improved STCS version (using trigger-guard encoding, Jaccard similarity function, and GA as minimization algorithm) with our baselines (random and coverage-based selection) and showed up to 77% reduction in cost (number of test cases required for achieving 90% FDR in the selected test suite).

5.4 Paper 4

"Reducing the Cost of Model-Based Testing through Test Case Diversity", H. Hemmati, A. Arcuri, and L. Briand, in the proceedings of the 22nd IFIP International Conference on Testing Software and Systems (ICTSS), formerly TestCom/FATES, pp. 63-78, 2010.

After the promising results obtained in the previous studies (Paper 2 and Paper 3), the next step was to look deeper into the basic ideas and assumptions underlying STCS. Therefore, in this paper, we first investigated the fundamental idea of diversifying test cases. Furthermore, we went beyond search algorithms to find the best technique for minimization. The two main research questions of this paper were:

RQ1. Why does diversifying test cases improve fault detection?

RQ2. What is the most cost-effective way to diversify (given our similarity measure) a set of test cases?

To answer RQ1, we needed to investigate whether test cases are distinctly clustered with respect to different faults or not. If they are, then we can conclude that diversifying test cases with respect to their pair-wise similarity increases chances of finding more faults. We have carried out an exhaustive analysis based on our industrial case study and the results showed that test cases which detect distinct faults are dissimilar and test cases that detect a common fault are similar with respect to our similarity measure. This finding suggests that rewarding diversity leads to finding more faults.

In the second half of the paper, we focused on the last step of STCS techniques, the minimization algorithm. In the previous papers (Paper 2 and 3) we used GA and Greedy algorithms, which are both search techniques. In this paper, we compared the best search technique among those two, which was GA, with two different approaches for our

minimization problem. The first technique was clustering and the rationale for choosing it was that test cases detecting distinct faults were distinctly clustered (based on RQ1). The second alternative was Adaptive Random Search, which is a common approach in test case generation for diversifying input data (Adaptive Random Testing). The results of the industrial case study (the same SUT as Paper 2 and Paper 3) showed that the GA is more effective and less expensive than both clustering and Adaptive Random Testing.

5.5 Paper 5

"Empirical Investigation of the Effects of Test Suite Properties on Similarity-Based Test Case Selection", H. Hemmati, A. Arcuri, and L. Briand, to appear in the proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation (ICST), 2011.

The three previous papers (Paper 2 to Paper 4), proposed STCS techniques, investigated the best approaches for encoding, similarity function, and minimization algorithm, by conducting case studies on one industrial software system, and compared the proposed technique with existing techniques. In addition, Paper 4 studied the fundamental idea and assumption behind the approach by an exhaustive analysis of the impact of test cases pair-wise similarities on the test suite FDR. In Paper 5, we conducted a controlled experiment based on two case studies (one new) to find out in which situations, with respect to test suite properties, STCS performs best. Therefore, the first research question of the paper was:

RQ1. Under which conditions, with respect to the similarity of fault revealing test cases in a test suite, STCS performs best?

The results showed that the most ideal situation for an STCS technique is when, in a test suite, (1) test cases that detect a common fault are similar and (2) test cases which detect distinct faults are dissimilar. More importantly, our empirical study shows that property (2) is much more important than property (1). This result will help researchers devise improved similarity functions in the future, which in turn will result into more effective selection techniques.

The paper also investigated (in RQ2 and RQ3) the problem of outliers (a small clustered set of test cases that is far away from all the others) in a test suite—which are not unlikely to happen in MBT—that could compromise the performance of STCS techniques. Results confirmed the significant impact of outliers and an approach, based on using rank scaling

measurement instead of raw similarity values, was proposed and was shown to partially address the problem. More investigation in this area is required.

5.6 Paper 6

"Achieving Scalable Model-Based Testing Through Test Case Diversity", H. Hemmati, A. Arcuri, and L. Briand, submitted to ACM Transactions on Software Engineering and Methodology (TOSEM), 2010.

Paper 6 is an extension of Papers 2 to 4 according to two dimensions:

A. New research questions are studied and reported.
B. New findings are presented for previous research questions, by conducting larger experiments.

(A) Three new research questions:

We investigated five research questions (RQ1 to RQ5) in this paper among which only RQ2 and RQ3 were similar to what was studied in Papers 2 to 4. The investigation of "How influential are STCS parameters on its effectiveness?" (RQ1), "What is the effect of the failure rate on the effectiveness of STCS?" (RQ4), and "How can one estimate the minimum number of test cases required for achieving (near) maximum FDR?" (RQ5) were completely new.

(B) Larger experiments:

This paper extends previous experiments by investigating a much larger number of STCS variants and an additional industrial case study. Regarding RQ2 ("What is the most cost-effective STCS variant?"), in Paper 2 we focused on the encodings and compared four STCS variants. In Paper 3, we focused on similarity functions and investigated six STCS variants. Minimization algorithms were the focus in Paper 5, where three STCS variants were compared. In total, in Papers 2 to 4, we introduced and evaluated 11 STCS variants on one case study.

In this paper, we introduced one new encoding, two new similarity functions, and six new minimization algorithms. This time we applied new analyses (including rank analysis and effect size comparisons) and investigated all possible combinations of the algorithms, which resulted in 4*8*10=320 STCS variants. We employed another extra case study compared to Papers 2 to 4 and evaluated the 320 variants on both case studies and identified the best technique on average on the two industrial case studies.

The results for RQ1 showed that all STCS parameters (encodings, similarity function, and minimization algorithm) are potentially influential on its FDR. The most cost-effective technique, on average across the two industrial case studies, is identified in RQ2 as the selection technique with state-trigger-guard-based encoding, Gower-Legendre similarity function, and (1+1) Evolutionary Algorithm [66].

Regarding RQ3 ("What is the practical benefit of using STCS?") we evaluated the best STCS technique (identified in RQ2) by comparing it with nine baselines on both case studies. In addition, more statistical analyses such as effect size analysis were applied. The results of the comparisons showed that nearly always (with a few exceptions that are discussed in the paper), the best STCS technique results in equal or higher FDR with fewer test cases. In addition, in most cases the FDR improvement is very significant (e.g., for some test selection sizes and case studies, 40% and 110% improvements are achieved when compared to the best coverage-based and random testing techniques, respectively).

To investigate RQ4 we conducted a controlled experiment and simulated test suites with different failure rates. Results showed that the proposed STCS technique is more effective than other baseline selection techniques, regardless of the test selection size and the failure rate. In RQ5, we introduced and successfully evaluated a method that monitors the trend of test cases average similarity increments when increasing the test selection size. The method helps test managers in deciding about the best tradeoff regarding the test suite size. In addition, the scalability of the three steps of the STCS was assessed, by investigating the effect of larger test suites and test cases on the performance of the selection technique.

6 Suggestions for Future Research and Extensions

Future research in this domain can be performed in at least four directions: First there is a need to conduct more and larger scale case studies on STCS, along with more theoretical and empirical analysis in order to assess the promising results in other contexts, with other SUTs and application domains. All this work should then converge towards a more comprehensive theory for STCS. For example, the analysis of the search space properties in this field of application can be helpful for gaining a more detailed understanding of how the approach works and may be improved.

A second direction is to improve the technique. A possible approach, for instance, can be combining STCS and coverage-based selection techniques by applying multi-objective search techniques [55, 67] that minimizes similarities while maximizing coverage of the

selected test cases. Another possibility is assigning weights to test cases based on estimates of their execution cost and modifying the selection technique to minimize the total execution cost. In addition, an improvement to STCS can be obtained by improving the current solution for the outlier problem.

The third direction is about trying to generalize the STCS approach for unsystematically generated test suites, which may have very low FDR. An interesting research question is whether the fundamental idea of STCS is only applicable and effective when the test suite is systematically generated or if it can be effective even on a randomly generated test suite. If the results show the effectiveness of the technique on a randomly generated test suite then its applicability widely increases.

The last direction is applying STCS on other domains in software testing than test case selection. For example, detecting infeasible test cases might be possible by identifying test cases which are similar to the already identified infeasible test cases in the test suite. Therefore, the idea of similarity-based infeasible test case detection is also worth investigating.

7 Conclusion

Transferring ideas from academic prototypes and projects to complex industrial software systems requires investigating their feasibility in practice. Addressing issues regarding problem size and solution cost dramatically increases the chance of a successful technology transfer. Model-based testing (MBT) is a good example of a field that has generated excellent ideas that, however, still need to be improved to be fully functional on real world systems.

This thesis targets one of the drawbacks of applying MBT, as a systematic and automatic technique, for the system level testing of embedded systems. The problem is that system testing in real hardware platforms or test networks may be a highly expensive task and is usually very constrained. Therefore, an ideal automated testing approach should be adjustable to the time and resource constraints of the project. However, MBT, which typically results in very large test suites, does not provide such flexibility.

The proposed solution in this thesis, similarity-based test case selection, allows MBT users to select a small enough portion of the original test suite that fits to their testing budget while preserving to the maximum extent the original fault detection rate of the test suite. The idea behind the proposed similarity-based test case selection technique is diversifying the selected test cases. The process contains three steps: (1) encoding of test cases into abstract test paths, (2) assigning similarity values to each test path pair, and (3) minimizing the sum of similarity values among all the test path pairs corresponding to the selected test cases.

First 320 variants of the technique were identified using different combinations of possible algorithms for each step, and then several case studies (in industrial contexts) and controlled experiments (in the form of simulations) were conducted in this thesis. The results showed that the combination with state-trigger-guard-based encoding, Gower-Legendre similarity function, and (1+1) Evolutionary Algorithm to be the most cost-effective technique on average across two industrial case studies. Using the proposed technique, a much higher fault detection rate was achieved for the same number of test cases compared to the baselines (coverage-based and random selection). This led to very large savings in terms of the number of test cases that do not need to be executed (up to 80% reduction in number of test cases required for detecting the same number of faults). The proposed technique is also more effective than other baseline selection techniques regardless of test selection size and failure rate. The technique showed a high degree of

scalability to larger test suites and longer test cases. In addition, a method was proposed that monitors the trend of test cases' average similarity increments when increasing the test selection size, to help test managers in deciding about the best test selection size within their constraints. In summary, the similarity-based test case selection technique helps practitioners to reduce the cost of test case execution to fit their needs and consequently make it possible to apply MBT to larger systems.

References for the summary

[1] 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*, vol. 36, pp. 742-762, 2010.

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

[3] L. C. Briand and Y. Labiche, "A UML-Based Approach to System Testing," in *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, 2001.

[4] I. K. El-Far and J. A. Whittaker, "Model-Based Software Testing," *Encyclopedia of Software Engineering (edited by J. J. Marciniak), Wiley*, 2001.

[5] S. Ali, L. C. Briand, M. J. Rehman, H. Asghar, M. Z. Z. Iqbal, and A. Nadeem, "A State-Based Approach to Integration Testing Based on UML Models," *Information and Software Technology*, vol. 49, pp. 1087-1106, 2007.

[6] A. C. D. Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A Survey on Model-based Testing Approaches: A Systematic Review," in *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, Atlanta, Georgia, 2007.

[7] D. Drusinsky, *Modeling and Verification using UML Statecharts: A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking*, 1st ed.: Newnes, 2006.

[8] M. Shafique and Y. Labiche, "A Systematic Review of Model Based Testing Tool Support," Carleton University, Technical Report (SCE-10-04), 2010.

[9] "D-MINT, Deployment of Model-Based Technologies to Industrial Testing," http://www.d-mint.org/ (Visited in Januray 2011).

[10] J. Feldstein, "Model-based Testing using IBM Rational Functional Tester," developerWorks, IBM, 2005.

[11] M. Vieira, X. Song, G. Matos, S. Storck, R. Tanikella, and B. Hasling, "Applying Model-Based Testing to Healthcare Products: Preliminary Experiences," in *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, 2008.

[12] Y. Gurevich, W. Schulte, N. Tillmann, and M. Veanes, "Model-based Testing with SpecExplorer," Microsoft research, 2009.

[13] S. Dalal, A. Jain, and J. Poore, "Workshop on Advances in Model-Based Software Testing," in *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, 2005, p. 680.

[14] F. W. Vaandrager, "Does it Pay Off? Model-Based Verification and Validation of Embedded Systems!," Radboud University Nijmegen2006.

[15] W. Grieskamp, R. M. Hierons, and A. Pretschner, "10421 Summary -- Model-Based Testing in Practice," D. S. Proceedings, Ed.: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2011.

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

[17] T. Weigert and R. Reed, "Specifying Telecommunications Systems with UML," in *UML for Real: Design of Embedded Real-time Systems*: Kluwer Academic Publishers, 2003, pp. 301-322.

[18] T. S. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Transactions on Software Engineering*, vol. 4, pp. 178-187, 1978.

[19] S. Sauer and G. Engels, "UML-based Behavior Specification of Interactive Multimedia Applications," in *Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*, 2001.

[20] "Papyrus, www.papyrusuml.org (Visited in January 2011)."

[21] T. Pender, *UML Bible*: Wiley, 2003.

[22] "IBM Rational Software Architect, http://www-01.ibm.com/software (Visited in Januray 2011)."

[23] "QTRONIC," Conformiq, http://www.conformiq.com/qtronic.php (Visited in January 2011).

[24] "AMOS Project home page," http://simula.no/research/approve/projects/amos, (Visited in January 2011).

[25] "Kermeta - Breathe Life into Your Metamodels," Rennes and Britanny IRISA and INRIA.

[26] "MOFScript Home page," http://www.eclipse.org/gmt/mofscript/ (Visited in January 2011).

[27] P. Ammann and J. Offutt, *Introduction to Software Testing*: Cambridge University Press, 2008.

[28] S. Ali, M. Z. Iqbal, A. Arcuri, and L. Briand, "A Search-based OCL Constraint Solver for Model-based Test Data Generation," Simula Research Laboratory, Technical Report(2010-16), 2010.

[29] M. Egea, "EyeOCL Software," http://www.bm1software.com/eos/ (Visited in January 2011).

[30] 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.
[31] M. Harman, "The Current State and Future of Search Based Software Engineering," in *Future of Software Engineering*: IEEE Computer Society, 2007, pp. 342-357.
[32] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology*, vol. 51, pp. 957-976, 2009.
[33] M. Harman, A. Mansouri, and Y. Zhang., "Search based software engineering: A comprehensive analysis and review of trends techniques and applications," Technical Report TR-09-03, Department of Computer Science, King's College London, 2009.
[34] "Software Engineering by Automated Search Home Page," http://www.sebase.org/sbse/publications/ Visited at 20th Jan2011.
[35] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*: Addison-Wesley Professional, 2001.
[36] R. L. Haupt and S. E. Haupt, *Practical Genetic Algorithms*: Wiley-Interscience, 1998.
[37] D. A. Coley, *An Introduction to Genetic Algorithms for Scientists and Engineers*: World Scientific Publishing Company, 1997.
[38] D. Whitley, "The GENITOR algorithm and selection pressure: why rank-based allocation of reproductive trials is best," in *the third International Conference on Genetic Algorithms* 1989, pp. 116-121.
[39] D. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Transactions Evolutionary Computation* vol. 1, pp. 67-82, 1997.
[40] R. Nilsson and D. Henriksson, "Test case generation for flexible real-time control systems," in *Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference on*, 2005, p. 8 pp.
[41] E. K. Burke and G. Kendall, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*: Springer 2006.
[42] 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, pp. 267-276.
[43] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification, and Reliability*, Published online in Wiley InterScience 2010.
[44] Y. Chen, R. L. Probert, and H. Ural, "Regression test suite reduction based on SDL models of system requirements," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 21, pp. 379-405, 2009.
[45] G. Rothermel, M. J. Harrold, J. v. Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Software Testing, Verification and Reliability*, vol. 12, pp. 219-249, 2002.
[46] 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.
[47] 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.
[48] 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, in press*, 2009.
[49] L. Briand, Y. Labiche, and S. He, "Automating regression test selection based on UML designs," *Journal of Information and Software Technology*, vol. 51, pp. 16-30, 2009.
[50] B. Korel, G. Koutsogiannakis, and L. H. Tahat, "Model-Based Test Prioritization Heuristic Methods and Their Evaluation," in *3rd Workshop on Advances in Model Based Testing, A-MOST*, 2007, pp. 34-43.
[51] A. P. Mathur, *Foundations of Software Testing*, 1 ed.: Addison-Wesley Professional, 2008.
[52] D. Leon and A. Podgurski, "A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases," in *14th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2003, pp. 442-456.
[53] X. Y. Ma, B. K. Sheng, and C. Q. Ye, "Test-Suite Reduction Using Genetic Algorithm," in *Advanced Parallel Processing Technologies*. vol. 3756: Springer Berlin / Heidelberg, 2005, pp. 253-262.
[54] 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.

[55] S. Yoo and M. Harman, "Pareto Efficient Multi-Objective Test Case Selection," in *International Symposium on Software Testing and Analysis (ISSTA '07)*, ACM, Ed., 2007, pp. 140-150.
[56] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S.Roos, "Time-Aware Test Suite Prioritization," in *International Symposium on Software Testing and Analysis (ISSTA '06)*, ACM, Ed., 2006, pp. 1-12.
[57] 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, pp. 510-514.
[58] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*: Cambridge University Press, 1997.
[59] R. Xu and D. Wunsch, "Survey of Clustering Algorithms," *IEEE Transactions on Neural Netwoks*, vol. 16, pp. 645-678, 2005.
[60] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*: Cambridge University Press, 1999.
[61] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2 ed.: The MIT Press, 2001.
[62] T. Y. Chen, F.-C. Kuoa, R. G. Merkela, and T. H. Tseb, "Adaptive Random Testing: The ART of test case diversity," *Journal of Systems and Software*, vol. 83, pp. 60-66, 2010.
[63] K. S. Khan, R. Kunz, J. Kleijnen, and G. Antes, *Systematic Review to Support Evidence-Based Medicine: How to Review and Apply Findings of Healthcare Research*, 2003.
[64] R. K. Yin, *Case Study Research: Design and Methods*: Sage Publications Inc, 2003.
[65] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering: An Introduction*: Kluwer Academic Publishers, 2000.
[66] S. Droste, T. Jansen, and I. Wegener, "On the analysis of the (1+1) evolutionary algorithm," *Theoretical Computer Science*, vol. 276, pp. 51-81, 2002.
[67] K. Lakhotia, M. Harman, and P. McMinn, "A multi-objective approach to search-based test data generation," in *The Genetic and Evolutionary Computation Conference (GECCO)*, 2007, pp. 1098-1105.

A Systematic Review of the Application and Empirical Investigation of Search-based Test-Case Generation

Shaukat Ali, Lionel Briand, Hadi Hemmati, and Rajwinder K. Panesar-Walawege

Published in the IEEE Transactions on Software Engineering (TSE), vol 36, no 6, pp. 742-762, 2010

Abstract— Metaheuristic search techniques have been extensively used to automate the process of generating test cases and thus providing solutions for a more cost-effective testing process. This approach to test automation, often coined as "Search-based Software Testing" (SBST), has been used for a wide variety of test case generation purposes. Since SBST techniques are heuristic by nature, they must be empirically investigated in terms of how costly and effective they are at reaching their test objectives and whether they scale up to realistic development artifacts. However, approaches to empirically study SBST techniques have shown wide variation in the literature. This paper presents the results of a systematic, comprehensive review that aims at characterizing how empirical studies have been designed to investigate SBST cost-effectiveness and what empirical evidence is available in the literature regarding SBST cost-effectiveness and scalability. We also provide a framework that drives the data collection process of this systematic review and can be the starting point of guidelines on how SBST techniques can be empirically assessed. The intent is to aid future researchers doing empirical studies in SBST by providing an unbiased view of the body of empirical evidence and by guiding them in performing well designed and executed empirical studies references.

1 Introduction

Software is being incorporated into an ever increasing number of systems and hence it is becoming increasingly important to thoroughly test these systems. One challenge to testing software systems is the effort involved in creating test cases that will systematically test the system and reveal faults in an effective manner. The overall testing cost has been estimated at being almost fifty percent of the entire development cost [6], if not more. Thus, a logical response is to automate the testing process as much as possible, and test case generation is naturally a key part of this automation. A possible strategy which has drawn great interest

in the automation of test case generation is the application and tailoring of metaheuristic search (MHS) algorithms [41]. The main reason for such an interest is that test case generation problems can often be re-expressed as optimization or search problems.

There has been a tremendous amount of research in applying MHS algorithms to test case generation and a large body of research exists: a search of the most relevant databases (as detailed in Section 4.2.1) found 450 articles which after reading abstracts resulted in 122 relevant articles published over the years 1996-2007 on this specific topic, often referred to as search-based software testing (SBST) [4].

Seeing the amount of research activity in this field, it is at this point in time, highly important to characterize what type of research has been performed and how it has been conducted. Among other things, it is crucial to appraise how much empirical evidence there is regarding the cost-effectiveness of SBST and to determine whether there is room for improvement in the way studies are performed and reported. The ultimate goal is to improve the quality of future research in this important, emerging field of research. In order to assess the current state of the art in SBST, we decided to conduct a comprehensive systematic review of the current literature, as this is commonly done in other scientific fields of research such as medicine [25] and social science [29]. The purpose of this systematic review is to collect, classify, and assess the empirical studies on SBST in order to assess the current body of evidence regarding the cost and effectiveness of SBST. By identifying the strengths and weaknesses of the current literature we hope to suggest improved research practices and relevant future research directions.

This paper is organized as follows: In Section 2, we provide the background relevant to the material presented in this paper. Section 3 suggests a framework used to assess the empirical studies in SBST and Section 4 presents the method used to conduct this systematic review. In Section 5, we present the results of our review whilst Section 6 outlines its validity threats. The final conclusions that we can draw from this systematic review are presented in Section 7.

2 Background

Detailed In this systematic review, we are analyzing which MHS algorithms have been used to address test case generation and what body of evidence exists regarding their cost-effectiveness. As a preliminary to the review itself, we introduce here the three main

components involved in this paper: search-based software testing, systematic reviews, and empirical studies.

2.1 Search-based software testing

The main aim of software testing is to detect as many faults as possible, especially the most critical ones, in the system under test (SUT). To gain sufficient confidence that most faults are detected, testing should ideally be exhaustive. Since in practice this is not possible, testers resort to test models and coverage/adequacy criteria to define systematic and effective test strategies that are fault revealing. A test case normally consists of test data and the expected output [36]. The test data can take various forms such as values for input parameters of a function, values of input parameters for a sequence of method calls, or seeding times to trigger task executions. In the context of this review, we are not dealing with the expected outputs, but focus exclusively on the generation of test data as this has been the objective of papers making use of SBST. In order to perform test case generation, systematically and efficiently, automated test case generation strategies are employed. Bertolino [7] addresses the need for 100% automatic testing as a means to improve the quality of complex software systems that are becoming the norm of modern society. A comprehensive testing strategy must address many activities that should ideally be automated: the generation of test requirements, test case generation, test oracle generation, test case selection, or test case prioritization. In our current review, we are only dealing with test case generation. A promising strategy for tackling this challenge comes from the field of search-based software engineering [23].

Search-based software engineering attempts to solve a variety of software engineering problems by reformulating them as search problems [15]. A major research area in this domain is the application of MHS algorithms to test case generation. MHS algorithms are a set of generic algorithms that are used to find optimal or near optimal solutions to problems that have large complex search spaces [15]. There is a natural match between MHS algorithms and software test case generation. The process of generating test cases can be seen as a search or optimization process: there are possibly hundreds of thousands of test cases that could be generated for a particular SUT and from this pool we need to select, systematically and at a reasonable cost, those that comply to certain coverage criteria and are expected to be fault revealing, at least for certain types of faults. Hence, we can reformulate the generation of test cases as a search that aims at finding the required or optimal set of test cases from the space of all possible test cases. When software testing

problems are reformulated into search problems, the resulting search spaces are usually very complex, especially for realistic or real-world SUTs. For example, in the case of white-box testing, this is due to the non-linear nature of software resulting from control structures such as if-statements and loops [17]. In such cases, simple search strategies may not be sufficient and global MHS algorithms1 may, as a result, become a necessity as they implement global search and are less likely to be trapped into local optima [16]. The use of MHS algorithms for test case generation is referred to as search-based software testing [4]. Mantere and Alander [35] discuss the use of MHS algorithms for software testing in general and McMinn [37] provides a survey of some of the MHS algorithms that have been used for test data generation. The most common MHS algorithms that have been employed for search-based software testing are evolutionary algorithms, simulated annealing, hill climbing, ant colony optimization, and particle swarm optimization [12]. Among these algorithms, hill climbing (HC) [12] is a simpler, local search algorithm. The SBST techniques using more complex, global MHS algorithms are often compared with test case generation based on HC and random search to determine whether their complexity is warranted to address a specific test case generation problem. The use of the more complex MHS algorithm may only be justified if it performs significantly better than HC.

2.2 Systematic reviews

Systematic reviews are a means of synthesizing existing research regarding a specific research question [29]. They are usually performed to summarize the existing evidence for a particular topic and aid in the identification of gaps in the current research and thus can form the basis of new research activity. A review protocol is created at the beginning of the review, which lays out the research questions being answered and the methodology that will be used to answer these questions. The protocol specifies a specific search strategy that is used to select as much of the relevant literature as possible and provides justification for why studies are included or excluded from the systematic review. The data to be collected to answer the research questions is also presented in the protocol. All this information is published so that readers can judge the completeness of the systematic review, and if necessary replicate it. These features distinguish the systematic review from the usual literature review or survey that is usually conducted at the beginning of a

[1] Global MHS algorithms are often contrasted with local MHS algorithms. The former are based on strategies for the search to avoid being stuck in local minima, thus being more effective in situations with complex search landscapes [12].

research activity. A systematic review synthesizes the existing work in a systematic, comprehensive, and unbiased manner.

2.3 Empirical studies for search-based software testing

Kitchenham et al. [19, 31] make the case for evidence-based software engineering that seeks to help practitioners make informed decisions related to software development and maintenance by integrating current best evidence from research with practical experience. Thus, to determine if SBST techniques can be applied in practice, we need to conduct empirical studies to assess their cost-effectiveness and scalability. The cost-effectiveness of a SBST technique is normally measured in terms of the ability of the technique to generate test cases that achieve a certain testing objective at a reasonable cost. The testing objective, as is the case with any test case generation technique, is to detect faults of a type that is explicitly defined or implicitly determined by the test model (e.g., state transition faults for a state machine model). In this review, we have focused on empirical studies of SBST techniques in order to assess whether convincing evidence exists to show their cost-effectiveness and scalability. For this purpose, it was necessary to define what we mean by an empirical study in this context and what constitutes a well designed and reported empirical study. Empirical studies are usually divided into three different types: surveys, case studies or experiments [52]. For this review, we have used a broad definition of empirical study, to include any kind of empirical evaluation that has been done in the field of SBST in order to be comprehensive in our investigation.

In order to determine what constitutes a proper empirical study in SBST, we looked at existing guidelines [27, 32, 52] for conducting empirical studies in software engineering, and those for evaluating SBST techniques in other fields. Wohlin et al. [52] and Kitchenham [32] present guidelines on how to conduct experimentation and empirical research in the specific context of software engineering whereas Johnson [27] presents a general guide for experimental analysis of algorithms. We have tailored and augmented some of these guidelines to create a specific framework for conducting and reporting empirical studies in the domain of SBST. This was necessary as SBST studies involve the analysis of automation techniques in which no human subjects are involved and presents many specific challenges. In addition, the fact that SBST techniques are based on MHS algorithms makes it important to account for the inherent random variation that exists in their results. Furthermore, there should also be some means to show that a SBST technique is really necessary for the context that it is being applied in. This can be done, for example,

by showing that other simpler search techniques do not perform as well. The reason for doing this is that we want to ensure that the problems being tackled by the SBST techniques do warrant their use.

The framework was created for a dual purpose. First, it was used in this systematic review to direct the collection of data that was used to assess the current state of empirical research in SBST. Second, it can also be used as a set of guidelines for conducting and reporting future research in the field or at least as a starting point in the development of such guidelines. The next section will present the framework.

3 Framework

As presented here, this framework is not intended to provide complete operational guidelines, but rather to justify the data collection that took place to perform the systematic review presented in the next sections and to highlight some of the most important concepts and issues.

The framework is divided into four parts. First, the test problem addressed must be clearly specified. Second, the MHS algorithms adopted must be clearly defined. Third, since any SBST research should always include empirical studies aiming at assessing the cost and effectiveness of the proposed approaches, the design of such studies must be carefully described so that its validity can be assessed. Last, results must be carefully reported so as to be clearly interpretable and reproducible. Whenever relevant, we will refer to Johnson's general guidelines on the experimental analysis of algorithms [27], either to point the reader to further, more general considerations, or to show that our more specific guidelines are a specialization of these more general ones.

3.1 Test problem specification

The test problem specification includes two main parts, the purpose of testing and the test strategy that will be employed. Each of these parts directly affects the form that the search-based software testing strategy will take. Figure 1 outlines the constituent parts of a test problem specification. The general purpose of software testing is to gain sufficient confidence in the dependability of a software artifact. Explicitly, this is usually done by targeting specific types of faults at different levels (such as unit, integration, and system testing). The targeted faults can be categorized in many ways depending on the view one takes of a system. At the highest level, one differentiates functional from non-functional faults, e.g., faults related to performance, security, robustness, and safety requirements.

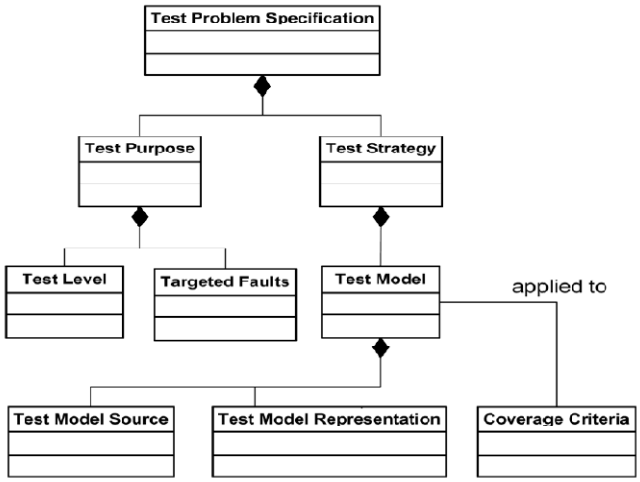


Figure 1 Concept diagram of test problem specification

A testing strategy is defined by a model of the SUT and some specific coverage criteria defined on that model. Such a model is typically referred to as a test model and the coverage criteria aim at systematically exercising the SUT based on the test model. This test model may be characterized by its source and representation (i.e., notation and semantics). Coverage criteria definitions depend on the test model representation. The source of the model implies constraints on the application of the test strategy as it depends on the availability and reliability of precise information in a specific form. As discussed in [5], possible sources for a test model can be the SUT specification, design artifacts or the source code itself. Based on the model source (specification, design or source code), different types of test models can be constructed. Typical examples of models derived from source code include control and data flow graphs, whereas test models based on SUT design include state machines or Markov usage models. To be systematic, a test strategy generates test cases to cover certain features of the test model. For instance, in the case of state machines, typical coverage criteria include the coverage of all states or all transitions, the latter being a stronger requirement, while, in the case of control flow graphs, a typical coverage criterion is branch coverage. It is important to clearly specify the coverage criteria as it is often used to measure the effectiveness of SBST techniques regarding test case generation.

3.2 Metaheuristic search algorithm specification

MHS algorithms are general strategies that need to be adapted to the problem at hand. When reporting a study, this implies describing and justifying the customizations and

parameter settings for each specific algorithm. This will be required for replicating the study and also for comparisons with other SBST techniques and future studies. Each type of MHS algorithm has specific parameter settings to be reported, but the general idea is to report all settings and adjustments that may have an effect on the performance of the algorithm or are needed for replicating the study. In Figure 2 we show how a typical genetic algorithm can be used for test case generation. The important parameters to report for a genetic algorithm would be the encoding of the chromosomes, the fitness function created to guide the search, the strategy for creating the initial population, the selection strategy for selecting parents for the next generation, the various recombination operators such as crossover and mutation operators and their values, the reinsertion strategy and the stopping criteria. We discuss in [5] how these parameters affect the results of empirical studies involving the use of genetic algorithms for test case generation.

3.3 Empirical study design

This section will define the most important items that should be reported about the study definition (through its objectives and hypotheses), design, and results.

3.3.1 Objectives and experimental hypotheses

One must define what is going to be empirically assessed and compared. The objective is usually to compare various SBST techniques and alternatives in terms of code coverage, fault detection, test suite size, or test case generation time. The empirical study can be an assessment of a single SBST technique, a comparison of two or more SBST techniques, or a comparison of SBST techniques versus non-SBST techniques (i.e., not relying on meta-heuristic search algorithms). The latter includes, for example, random search, static analysis, greedy algorithms or some other specific technique for the test problem under consideration, e.g., schedulability analysis in the case of real-time systems. In any case, what is going to be compared should be precisely specified through formal test hypotheses, thus leading to appropriate statistical significance testing. One notion important here is to

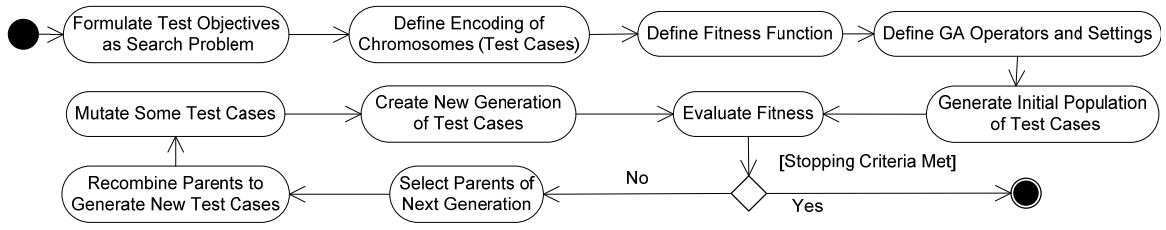


Figure 2 Test case generation using genetic algorithms

state the kind of hypothesis that will be used: either a one-tailed hypothesis or a two-tailed hypothesis [14]. This has an impact on how we interpret the results in terms of p-values (probability of type I errors). In the context of SBST, a one-tailed hypothesis would be used in the case when, based on the properties of the fitness function, we have a theoretical basis to assert the direction of the expected outcome. For example, when comparing a guided search algorithm such as genetic algorithm with random search, we may, based on an analysis of the fitness function, expect the genetic algorithm to be equally or more effective at hitting the search target – but not worse – and as such we would use a one-tailed hypothesis. However, as an example, when comparing two genetic algorithms with different fitness functions, where we cannot state upfront which one would fare better in terms of cost or effectiveness, we would use a two-tailed hypothesis. In other words, when the theory regarding the search algorithms under study allows us to be a priori confident regarding the possible direction of differences in cost or effectiveness, then we should use a one-tailed test as this will increase our chances to uncover a statistically significant difference.

3.3.2 Target application domain

Empirical studies should specify a target application domain in which their results are intended to be generalized. Example application domains are: real-time, concurrent, distributed, embedded, and safety-critical. Testing techniques typically target specific faults that are more relevant in certain application domains, e.g., slow response time in real-time systems. Moreover, assumptions are typically made regarding the availability of information required to build the test model. Such assumptions tend to be more or less realistic depending on the application domain. For example, if one assumes the use of the MARTE UML profile [3] to design a system and then derive a test model, this is of course more realistic in the context of embedded, real-time applications. Further, the selection of subject systems for empirical studies will then be partly determined by the target application domain.

3.3.3 Subject systems (Software Under Test or SUT) specification

After identifying the target application domain, specific SUTs fitting that domain are selected. It is important to carefully select SUTs and precisely justify why the selected SUTs are adequate matches for the target application domain as this will help the reader determine the extent to which the experimental results will generalize to this domain. This discussion should be in terms of the inherent properties of the SUT such as its size,

complexity, or structure. This is particularly important when one is creating artificial SUTs specifically for the experiment, a common situation when one is trying to account for SUTs of varying size and complexity. For each SUT in the empirical study, the function of the SUT together with relevant properties affecting its representativeness of the domain should be carefully reported in order to ensure the reproducibility of the experiment and help future comparisons of cost-effectiveness results. Johnson [27] discusses the general problem of instance selection (i.e., SUTs here) in experiments (Principle 3: Use instance testbeds that can support general conclusions) and defines reproducibility (Principle 6: Ensure Reproducibility) when experimenting with algorithms as the capacity to "perform similar experiments that would lead to the same basic conclusions". The goal is to make it possible to confirm the results of an original experiment independently from the precise settings and details of the experiment. In addition to SUT properties, the hardware platform that the SUT executes on is also important to specify. Johnson [27] provides an in-depth discussion of the latter issue (Principle 7: Ensuring Comparability), which is not specific to SBST, and suggestions to address it. In its Principle 9, about well-justified conclusions, Johnson [27] also discusses the danger of drawing conclusions from small instances that are then generalized to much larger instances, as the former do not always predict well the latter, and recommends to use instances that are as large as possible.

3.3.4 Measures of cost and effectiveness for SBST techniques

Measuring effectiveness and more particularly cost in our context is inherently difficult and the validity of measures is very often context-dependent. As discussed by Johnson in [27] (Principle 6: Ensure Reproducibility), just reporting effectiveness and cost values is not very informative as it does not provide direct insights into what these values actually imply. It is nevertheless crucial, in order to draw useful conclusions from studies involving SBST techniques, to be able to use appropriate comparison baselines. In our context, one usually resorts to comparing the investigated technique to simpler, existing techniques (see Section 5 on baselines of comparisons) in order to assess the relative goodness of a search. The measures should be relevant for the particular study and comparable across the different techniques being investigated. Studies may use slight variations of an existing measure or introduce new ones, hence, it is important to explain the reasoning behind the effectiveness and cost measures and justify why they are applicable in the context they are being used. Along with the measure, the method used to collect the data related to the measures should be thoroughly explained. In the context of SBST, the effectiveness of a

test case generation technique is closely related to the “quality” of the test suite generated by the technique. A good test suite can be characterized by its ability to uncover faults or to give confidence in the SUT by fulfilling a certain coverage criterion. Thus we can say that, in practice, there are two main categories of measures of effectiveness, which can be referred to as coverage-based measures and fault-based measures. In the former category, there may be many different types of measures depending on the adequacy criteria being used, for example, control-flow coverage criteria such as branch or path coverage may be used. The fault-based measures are typically fault detection scores. They can be computed based on real, known faults or are estimated through mutation analysis [48]. In the latter case, the program is seeded with faults based on mutation operators and depending on the number of faults caught, a so-called mutation score is calculated. The techniques are assessed upon how successful they are at detecting the seeded faults.

Cost measures are generally related to the speed of the technique to converge towards the test objective (in some cases it is referred to as the search technique’s “efficiency”). Some common cost measures used in the SBST domain are: (a) the number of iterations, which shows how many times a SBST technique needed to iterate in order to find its best solution, e.g., the number of generations in genetic algorithms, or cycles in ant colony optimization algorithms, (b) the cumulative number of individuals in all iterations (usually each individual represents a test case in SBST), (c) the number of fitness evaluations an algorithm needs, to find the final solution, which depends on the number of newly generated individuals (usually each new population is made up of some individuals from the previous iteration and some newly generated ones), (d) the time spent by a MHS algorithm to find test cases meeting the targeted test objective, which is sometimes referred to as “test case generation time”. This time can be either measured using clock time or CPU cycles. Clock time is the time from the “wall” clock and not easily comparable across different hardware architectures. However it is a practical measure that can be used to assess if a technique can be used in practice. CPU cycles on the other hand is a measure that can be used across techniques for comparison on other hardware architectures as well, and (e) the size of the resulting test suite, which is a surrogate measure for the cost of the time it would take to execute the resulting test suite since a larger test suite would require more resources to execute.

Among the first three cost measures, the number of iterations is a very coarse grained measure and is not as precise as the number of individuals, which in turn is not as precise as the number of fitness evaluations. The number of fitness evaluations is more precise

than the number of individuals because in each iteration there are some individuals that are kept from the previous population and there is no cost for generating them. Therefore, the number of evaluations can more precisely estimate the real cost of a SBST technique. All these three measures are surrogate measures for the time used to generate the final test suite but none is perfect, because different search techniques may require a different amount of time per iteration, per creation of an individual (test case), or per fitness evaluation. For instance, it would not be a good idea to compare simulated annealing (SA) and genetic algorithms (GA) based on the number of iterations because the amount of time required for each iteration in GA and SA is likely to differ significantly.

The cost of a technique is generally measured for one of two purposes: either to compare two techniques to assess which one will cost less for the same effectiveness or to assess whether a technique can be used in practice given expected time constraints. From the measures discussed above, “test case generation time”, if it has been measured under similar conditions, is the only measure that can give users an intuitive idea of whether they can apply a particular technique to their situation within the time constraints that they have. When comparing the cost of different techniques, it is also necessary to make sure that any other required resources are kept equal amongst the techniques. The fact that two techniques require the same amount of time does not mean that they have the same cost if one technique consumes much more memory than the other. Therefore all relevant types of resources must be accounted for when comparing the cost of SBST techniques.

3.3.5 Measures for scalability assessment

Scalability assessment is the process of assessing how the cost-effectiveness of a SBST technique evolves as a function of the size of the test case generation problem to be addressed. This involves one or more measures of SUT size and the analysis of their relationships with the cost or effectiveness of the SBST techniques under investigation. Some examples of measures that can be scaled up include the size of the SUT in terms of lines of code or the size of search space in terms of number and range of input data parameters. The effect of this scaling is then observed on different cost and effectiveness measures to see if the SBST technique is still cost-effective as the SUT gets larger and more complex.

3.3.6 Baselines for comparison

A SBST technique can only be assessed if it is compared with a carefully selected, meaningful baseline since the optimal solution is normally not known. Since it is difficult

to assess SBST techniques in absolute terms, it is therefore important to show, as a minimum, that the problem at hand could not be addressed by some simpler means. In other words, every study should have one or more baselines of comparison when assessing SBST techniques and the minimum to be expected is a comparison with random search. The SUT investigated may, for example, be small and simple, and the fact that a SBST technique performs well may not mean much. Random search can then serve as a basic verification that the search problem cannot be addressed by a simple random search and warrants the use of a SBST technique. It is also preferable to use other simple SBST techniques, such as HC, as a comparison baseline for other more expensive SBST techniques. This further demonstrates that the use of a SBST technique is justified given the test case generation problem at hand. In addition—but this is context dependent—other SBST techniques, previously published or considered plausible alternatives, can also be used as baselines of comparisons for the proposed SBST techniques.

As discussed in [27], once baseline techniques are selected, one must ensure that reasonably efficient implementations are used for all techniques in order for cost and effectiveness to be comparable. Documentation, source code, URLs for downloadable tools, or at the very least a careful description of the implementation, should be provided.

3.3.7 Parameter settings

Most SBST techniques require parameter settings which tend to have a significant impact on their performance. In many studies, alternative parameter settings are investigated and compared. It is therefore highly important, to make any study reproducible, to specify these parameters in a precise manner. It is also interesting to justify their values based on existing studies, when possible, as this provides insights into how cost and effectiveness could be affected if they were changed or if a different SUT with different properties was used. One particularly important parameter in our context is the stopping criterion of the search (Principle 6: Ensure Reproducibility). It can be based on whether the search objective has been reached (or one is sufficiently close), execution time or a surrogate measure (due to practical constraints), or any significant progress is observed over a period of time.

3.3.8 Accounting for random variation in SBST results

Since SBST techniques use MHS algorithms; their results can vary from one execution to another. So, it is important to ensure that we run the algorithms a sufficient number of times to capture the random variation of results and be able to perform statistical

comparisons with other search techniques. It is difficult to precisely specify the number of runs required in general but, as a ballpark number, it should probably be above ten, so as to allow the use of basic statistical hypothesis testing and obtain a reasonable statistical power to detect large differences [52]. Based on the expected (minimum) difference between techniques (if this can be estimated) and the statistical tests used to compare cost and effectiveness across techniques, the minimum required number of runs can be estimated using power analysis [18].

When dealing with multiple runs, in our context, we are often interested in the best run, yielding the best test suite or test case according to some fitness function (e.g., bringing the execution time of a task as close as possible to its deadline). Another frequent case is when we are interested in the frequency with which a certain target was reached across runs (e.g., test input data satisfying certain constraints). In both cases, it is important to report the execution time and other cost measures of all runs and, when relevant, information about their fitness distribution. The basic principle is that it should be possible to estimate the total cost of achieving the best solution or, depending on what is relevant, the expected cost to achieve the search target. From a more general standpoint, Johnson (Principle 6: Ensure Reproducibility) [27] warns against reporting only effectiveness and cost data for the best run.

3.3.9 Data analysis

During the design of an empirical study, it is important to decide about the data analysis methods that will be applied to cost-effectiveness and scalability results.

Data analysis methods for comparing cost-effectiveness. Performance in the case of SBST usually relates to measuring the cost-effectiveness of the various search techniques. The cost and effectiveness of a SBST technique are used together for assessing its performance. For example, a technique that has higher coverage than another technique may not be considered to have better performance, because it uses significantly more fitness evaluations (higher cost) to achieve that effectiveness, thus making it impractical for larger SUTs. Any claims of better performance should be backed by empirical evidence demonstrating lower cost or higher effectiveness when compared to the baseline and alternative techniques. In the ideal case, a study that is concentrating on measuring cost, should keep the effectiveness measures constant. For example, the study may measure the number of fitness evaluations needed to achieve 100% branch coverage. If, however, the aim is to measure effectiveness, then this can be done by keeping the cost constant, for

example, by measuring how much branch coverage is achieved in some constant amount of time or number of fitness evaluations. The reported performance results should include the results of the comparison baselines. At a high level, reported results should follow the structure below:

Reporting descriptive statistics. Both cost and effectiveness distributions should be reported (e.g., as a table with descriptive statistics) and analyzed. Looking at their standard deviation may indicate the level of uncertainty in terms of cost and effectiveness associated with a SBST technique. This in turn may help determine how many runs would in practice be necessary to guarantee that we obtain a satisfactory result, i.e., achieve the objective.

Results of hypothesis testing. The purpose of statistical testing is to determine whether differences across SBST techniques in terms of central tendencies for cost and effectiveness can be attributed to chance or whether they really capture a trend. Statistical hypothesis testing is necessary as SBST techniques are always associated with a certain level of random variation in terms of cost or effectiveness. Because statistical testing is a standard practice, we will not detail it further here and interested readers may consult reference [40] for more details.

Statistical hypothesis testing should be used to accept/reject research hypotheses related to the cost-effectiveness analysis of SBST techniques and comparison baselines. The choice of a specific statistical test depends on the specific objective of SBST. In our context, hypothesis testing falls into three broad categories: (1) Comparing samples of runs in terms of effectiveness and cost. For example, comparing average or maximum branch coverage achieved across runs of alternative SBST techniques and baselines of comparison. (2) Comparing samples of runs in terms of "successful" runs. For example, comparing the proportion of runs that find a deadlock across alternative SBST techniques and baselines of comparison. (3) Comparing samples of targets (e.g., control flow branches) in terms of cost (e.g., iterations) or effectiveness (e.g., percentage of runs reaching that branch). In this last case, the samples are not independent, because observations in each sample are paired (identical targets). This leads to the application of specific statistical tests for paired samples. Moreover, though this is a standard issue, there can be two or more samples, and this will also affect the specific statistical test to be used. Moreover, as usual in other contexts, specific statistical tests have to be selected and justified based on the data distributions of the samples being compared to avoid drawing incorrect conclusions from the analysis. Statistical tests are usually classified as parametric and non-parametric [52]. When the sample follows a specific distribution (e.g., normal),

certain parametric tests are applicable (e.g., t-test). Alternatively, non-parametric statistical tests are used when no appropriate assumptions can be made about the sample distributions. The issues related to selecting appropriate tests are however discussed in standard textbooks and will not be further addressed here. In Table 1, as a guideline, we provide a mapping between the analysis situations we have encountered in SBST studies and the type of statistical tests that are suitable (for the sake of simplicity, we are assuming two samples, that is, the comparison of two techniques). This mapping is illustrated with examples.

Data analysis should both address statistical and practical significance of differences among alternative search techniques. The former assesses whether differences among search techniques can be due to chance. The latter assesses whether the difference can be considered of practical significance, that is, whether they would make any difference in the day-to-day practice of test case generation given the specific test objectives being considered. For example, if statistical testing based on a large number of runs show that there is a significant difference between the cost of two search techniques in terms of time required for finding the best test suite, the actual difference may not be of practical importance if it is in the range of a few minutes. On the other hand, a lack of statistical significance despite a visible difference may be due to small samples, and therefore a lack of statistical power, which in our context means that the number of runs for each compared

Table 1 Mapping of SBST problems to statistical tests

SBST Analysis Type	Type of Statistical Comparison	Example in the Context of SBST	Type of Statistical Test (assuming two samples)
Comparing samples of runs in terms of effectiveness and cost	Comparing central tendencies of two or more independent samples, each corresponding to a SBST technique	Comparing maximum branch coverage achieved across all runs between two SBST techniques	Parametric *t*-tests or Non-Parametric Mann-Whitney U test
Comparing samples of runs in terms of "successful" runs	Comparing proportions in independent samples, each corresponding to a SBST technique	Comparing the proportion of runs finding deadlocks across different SBST techniques	z-score test for proportions
Comparing samples of target in terms of cost to reach them or frequency at which runs reach them	Comparing central tendencies of matched pairs across samples	Comparing the frequency, across samples of runs matching each SBST technique, according to which a branch (target) is covered. Note that the observations across samples are paired as they correspond to identical branches.	Parametric Paired *t*-tests or Non-Parametric Wilcoxon or Sign test

search technique may be too small. The larger the number of runs, the more likely one is to obtain statistical significance when observing differences.

Data analysis methods for scalability. Scalability is used to assess whether a SBST technique can be applied to either larger or more complex SUTs and still have satisfactory effectiveness and cost. If the aim of the empirical study is to show the scalability of a SBST technique then appropriate measures of size and complexity should be clearly defined. There will be at least two measures involved – one size measure that will be scaled up through successive SUTs and the other that will measure the corresponding performance (cost and effectiveness). Then the effect of scaling up a particular measure can be reported in terms of a statistical relationship (recall the unavoidable random variation). For example, we may investigate several SUTs of variable sizes in terms of lines of code and then assess whether a SBST technique can still reach a certain level of coverage at acceptable cost (e.g., measured as the number of generations) for larger SUTs and analyze how this cost evolves with the size of the SUT. A positive, exponential relationship between size and cost might then be problematic, for example, as it would undermine the applicability of the technique for large scale test models and systems. Similarly, if effectiveness (e.g., in terms of achieved coverage) is strongly decreasing as a function of SUT size, we also have a scalability problem.

As for scalability analysis, we need to characterize relationships between SUT size variables and measures of the SBST technique's cost and effectiveness. Such techniques are typically analyzed through regression analysis, though in practice, because the number of SUTs under study is likely to be small, such analysis is more likely to be qualitative, that is simply based on observing scatter plots in the cost-effectiveness and size space.

3.3.10 Discussion on validity threats

Validity threats should be considered throughout any empirical study, right from the study definition and design up to the analysis and interpretation of results [52]. The following types of threats can be discussed:

Construct validity threats. Measures of cost, effectiveness, and SUT size should be appropriate and justified given the context and objectives of investigation. No measure is expected to be perfect as the above concepts are usually not readily measurable. But in practice, by using several, complementary measures of cost, effectiveness, and SUT size, one is in a position to compare the cost-effectiveness and scalability of alternative search techniques.

Internal validity threats. If a SBST technique performs better than another one, whether regarding effectiveness or cost, can it be due to something other than the SBST technique? This could possibly be due to the following: 1) poor parameter settings of one or more of the SBST techniques, 2) the biased selection of SUTs that have certain characteristics that can favor a certain SBST technique.

Conclusion validity threats

- Has random variation been properly accounted for? Since SBST techniques use MHS algorithms, randomness in results (inherent to metaheuristic approaches) should be accounted for, as discussed above. Has it been done in such a way as to enable statistical comparisons? It implies that a sufficient number of independent runs be performed to obtain a sufficient number of observations.
- Was the right statistical test employed? Statistical test procedures should be carefully selected given the hypothesis method (e.g. one-tailed vs. two tailed hypothesis) and the data collected (distributions of cost and effectiveness). Otherwise, certain required properties of a particular statistical test could be inadvertently violated leading to incorrect conclusions. For example, many statistical tests assume that data distributions be normal [52].
- Is there any practically significant difference? To answer this question, the magnitude of the differences must be reported– this is known as the effect size and determines the practical significance of the results.

External validity threats. This is a difficult issue, as whether results can be generalized depends on whether the SUTs under investigation are representative of the targeted application domain and whether the faults considered (if used to assess test effectiveness) are representative of real faults. Ideally, SBST empirical studies should also be run on many different SUTs of the target type, but every research endeavor faces limitations in terms of time and resources. At the very least, the issue should be carefully discussed and a good case should be made as to why one should be able to trust that the observed results can be generalized.

4 Research Method

In this section, we will explain our review protocol. We define the research questions that this review attempts to answer, along with how we selected papers for inclusion and the data that we extracted.

4.1 Research questions

The most important stage of any systematic review is to precisely define the research questions. Once the research questions have been specified, the systematic review can then proceed with the search strategy to identify relevant studies and extract the data required to answer the questions [13]. In this paper, we are interested in investigating empirical studies in the domain of SBST. To proceed with our investigation, we defined the following three research questions:

RQ1: What is the research space of search-based software testing?

The objective of this question is to characterize the research that has been undertaken so far. Though the research space can be identified from different angles, because our systematic review is about SBST, basic features of software testing (such as test level, targeted faults, test model, type of test cases, and application domain) and the type of MHS algorithms seem relevant characteristics to define the research space. Because of size constraints, RQ1 will not be addressed in detail in this paper and the results will be simply summarized to provide context information to the reader and facilitate the interpretation of subsequent research results. Interested readers may consult the technical report [5] corresponding to this paper for a detailed discussion of the results.

RQ2: How are the empirical studies in search-based software testing designed and reported?

A study that has been properly designed and reported (as discussed in Section 3) is easy to assess and replicate. The following sub-questions aim at characterizing some of the most important aspects of the study design and how well studies are designed and reported:

- RQ2.1: How well is the random variation inherent in search-based software testing, accounted for in the design of empirical studies?
- RQ2.2: What are the most common alternatives to which SBST techniques are compared?
- RQ2.3: What are the measures used for assessing cost and effectiveness of search-based software testing?
- RQ2.4: What are the main threats to the validity of empirical studies in the domain of search-based software testing?
- RQ2.5: What are the most frequently omitted aspects in the reporting of empirical studies in search-based software testing?

RQ3: How convincing are the reported results regarding the cost, effectiveness, and scalability of search-based software testing techniques?

This research question attempts to synthesize the actual results reported in the studies in order to assess how much empirical evidence we currently have. To answer this question, we address the following sub-questions:

- RQ3.1: For which metaheuristic search algorithms, test levels, and fault types, is there credible evidence for the study of cost-effectiveness?
- RQ3.2: How convincing is the evidence of cost and effectiveness of search-based software testing techniques, based on empirical studies that report credible results?
- RQ3.3: Is there any evidence regarding the scalability of the metaheuristic search algorithms for test case generation?

4.2 Study selection strategy

This is the step of a systematic review that aims at ensuring the completeness of the selection of papers on which the review is based. Study selection involves two main steps: (1) selection of the source repositories and identification of the search keywords (2) inclusion or exclusion of studies based on certain inclusion and exclusion criteria.

4.2.1 Source selection and search keywords

The process of selecting papers is started by executing a search query on the source repositories, which provides a set of papers. Since this set of papers is then subsequently used for all manual inclusions and exclusions, the selection of appropriate repositories and search strings is of utmost importance as it directly affects the completeness of the systematic review. The repositories that we used are: *IEEE Xplore*, *The ACM Digital Library*, *Science Direct* (including *Elsevier Science*), *Wiley Interscience*, *Springer*, and *MIT Press*. The first two repositories covered almost all important conferences, workshops, and journal papers, which are published either by IEEE or ACM. The next four repositories were mostly used for finding papers that are published in leading software engineering journals.

We selected the following journals based on [13]: *IEEE Transactions on Software Engineering (TSE)*, *ACM Transactions on Software Engineering and Methodologies (TOSEM)*, *IEEE Software (SW)*, *Springer: Software Testing Verification and Reliability (STVR)*, *Springer: Empirical Software Engineering*, *Elsevier Science: Information and Software Technology (IST)*, and *Elsevier Science: Journal of Systems and Software (JSS)*.

Since our review is about SBST, we also included journals relating to software quality assurance and evolutionary computing: *Springer: Software Quality Journal, Springer: Genetic Programming and Evolvable Machines, IEEE: Transactions on Evolutionary Computation,* and *MIT Press: Evolutionary Computation*. Another important source of publications that we included was the *Genetic and Evolutionary Computation Conference (GECCO)*. Based on the impact factor, GECCO is one of the top conferences in the fields of artificial intelligence, machine learning, robotics, and human-computer interaction [1] and is directly related to the field of genetic and evolutionary computation. GECCO's proceedings were published by Springer in 2003 and 2004 and afterwards by ACM.

A systematic way of formulating the search string includes (1) identifying the major search keywords based on the research questions (2) finding alternative words and synonyms for the major keywords and (3) creating a search string by joining major keywords with Boolean AND operators, and the alternative words and synonyms with Boolean OR operators.

Based on our main research focus, which is investigating empirical studies in the domain of SBST, the following major search keywords are used in this paper: *software testing* and *metaheuristic search algorithm*.

We did not use *empirical study* as a keyword because we realized that not all papers that perform an empirical study, in the broad sense that we have defined it, use this keyword.

To formulate our search query we tried a number of search strings and came to the conclusion that *'software testing'* as an expression is not a good keyword because there are many papers which don't use these two words together but are nevertheless related to software testing. These papers may use terms such as testing, test case, test data and so on. On the other hand if we used the term testing alone, we would find too many unrelated papers. So we decided to use the terms *software* and *test* linked together with a Boolean AND instead of using *'software testing'* as an expression. Using *'software'* and *'test'* will find almost all related papers to software testing, but to make sure that we do not miss any interesting papers in test case generation we used the expression of *'test case generation'* as an alternative for software testing.

Metaheuristic search algorithm is the second major term and also has many alternatives. We used general terms such as '*evolutionary algorithm*', '*meta-heuristic*', and 'search based' to explore the domain. Also, names of different MHS algorithms were used to make sure that no related papers were missed.

We also wanted to make sure that we do not miss any papers that have explicitly used

the widely used term *'evolutionary testing'*, and thus included the expression of '*evolutionary testing*' as a separate search string joined with the main string by an OR Boolean operator. The above decisions lead to the following search string shown in Figure 3.

The whole string is searched in each repository in all titles, keywords, and abstracts. The expression '*evolutionary testing*' is searched in the entire contents of all papers in the repositories as well.

One problem that we realized after some manual checking of the results of the search query was the fact that some search engines, such as IEEE Xplore, differentiate between the singular and plural form of words. To deal with this, we had to add some more alternative words and expressions to the search string by adding a '*s*' to the end of all the words we already had. For example, we added '*evolutionary algorithms*', '*meta-heuristics*', '*genetic algorithms*' and so on.

After finalizing the search string, the search query was run on the search engines of different repositories.

4.2.2 Study selection based on inclusion and exclusion criteria

Metaheuristic search algorithms have been used to automate a variety of software testing activities such as test case generation, test case selection, test case prioritization, and optimum allocation of testing resources. Since the focus of this systematic review is on test case generation, it is therefore necessary to define suitable inclusion and exclusion criteria for selecting relevant papers. In this section, we will discuss and justify the inclusion and exclusion criteria that were used.

We executed our search query on all selected databases and found 450 (after removing duplicates from different repositories) research papers in total. We only included papers up to the year 2007. In order to select the relevant papers to answer our research questions, we applied a two-stage selection process. At the first stage, we excluded papers based on abstracts and titles. All the papers were divided into three sets and each set was read by a researcher. We applied the following exclusion criteria:

- Abstracts or titles that do not discuss test case generation or any of the alternate terms that we used were excluded.
- Abstracts or titles that do not discuss the application of any MHS algorithm to automate test case generation were excluded.

{(((*software* AND *test*) OR *'test case generation'*) AND (*'evolutionary algorithm'* OR *'hill climbing'* OR *'metaheuristic'* OR *'meta-heuristic'* OR *'genetic algorithm'* OR *'optimization algorithm'* OR *'search-based'* OR *'search based'* OR *'simulated annealing'* OR *'ant colony'*)) <in abstract, keywords, and title>} OR *'evolutionary testing'* <in abstract, keywords, title, and whole content>

Figure 3 The search string used for selecting the papers from repositories

If a researcher was unsure about a paper after reading its title and abstract, then the paper was included for the second phase of selection. After applying the inclusion criteria for the first phase, we were left with 122 papers.

At the second stage, we again divided the papers into three equal sets and divided them among three researchers to check the contents of each paper. We excluded papers based on the following exclusion criteria:

- Posters, extended abstracts, technical reports, PhD dissertations, and papers with less than three pages were excluded. Our goal was to account only for peer-reviewed, published papers that presented sufficient technical details.
- The papers that do not automate test case generation were excluded because this is the scope of our review.
- The papers that do not report any empirical study (see Section 2.3 for details on what we mean by empirical studies) were excluded.

In the cases where a researcher could not decide whether to keep or exclude a paper, then the paper was discussed with other researchers and a decision was made, by consensus. It is important to mention that we didn't exclude papers based on the realism of SUTs used in their case studies. The reason is that exclusion would then be subjective as no precise criterion can be defined and would probably lead to a very small number of selected papers. After applying the second phase of selection, we remained with 68 papers that contained empirical studies about test case generation using MHS algorithms. However, four of these 68 papers, presented empirical studies that had already been reported in some other paper. This occurred, for example, when the journal version of a conference paper was found. In these cases we extracted data about the study from both the conference and journal versions of the paper and reported them as one study. Thus in the rest of the review we mention only 64 papers in total, even though we did analyze 68 papers. Details on the number of papers found in each database and number of papers included after applying inclusion and exclusion criteria are listed in Table 2.

Table 2 Distribution of papers after applying inclusion and exclusion criteria

Repository	Number of Included Papers After Applying Search Query	Number of Papers After Stage 1 Exclusion Criteria	Number of Papers After Stage 2 Exclusion Criteria
IEEE Xplore	297	77	33
ACM Digital Library	117	27	22
Wiley Interscience	8	2	2
Science Direct	8	3	2
Springer	19	12	8
MIT Press	1	1	1
Total	450	122	68

4.2.3 Data extraction

We designed a data extraction form in Microsoft Excel to gather data from the research papers. We collected two sets of information from each paper. The first set included standard information [30] such as name of the paper, authors' names, a brief summary, researcher's name, and additional comments by the researcher. The second set included the information directly related to answering the research questions (see Table 3 for a summary list and [5] and Section 3 for details on each data item). To assess and improve consistency of data extraction among the researchers, a sample of papers were selected and read by all researchers and the relevant data extracted. The extracted data was then discussed by the researchers to ensure a common understanding of all data items being extracted and where necessary, the data collection procedure was refined. The final set of selected papers from each repository was then divided amongst three researchers. Each researcher read the allocated papers and extracted the data from the papers. In order to mitigate data collection errors, the data extraction forms of each researcher were read and discussed by two others. All ambiguities were clarified by discussion among the researchers.

Table 3 Research questions and type of data collected

Research Questions		Type of Data Collected
RQ 1		Type of MHS algorithms, test levels, targeted faults, test model, type of test cases, and application domain
RQ 2	RQ 2.1	Number of runs, analysis method
	RQ 2.2	Comparison baseline
	RQ 2.3	Measures of cost, measures of effectiveness
	RQ 2.4	Conclusion, external, internal, and construct validity threats
	RQ 2.5	All of the information from RQ2.1 to RQ2.4 is used, formal hypothesis, object selection strategy, data collection method
RQ 3	RQ 3.1	Test level, fault type, MHS algorithm
	RQ 3.2	Test purpose, comparison baseline, cost and effectiveness results
	RQ 3.3	Scalability results

5 Results

The following section outlines the results related to the research questions. No formal meta-analysis of the results of the empirical studies could be performed because of the variations in the way empirical studies are conducted and reported, and as such, results are compiled in structured, tabular form.

5.1 RQ1: What is the research space of search-based software testing?

As previously mentioned, we provide here only the most salient results to the research question. The reader is invited to read the technical report [5] corresponding to this paper to obtain detailed results. The results show that in the majority of the papers, SBST techniques have been applied at the unit testing level (75%). Moreover, most papers (78%) do not target any specific faults but rather focus on structural coverage of different test models. The most commonly used algorithm is the GA and its extensions (73%), followed by a more limited use of simulated annealing and its extensions (14%). There could be several reasons for this frequent use of genetic algorithms. First, there are numerous publications on the application of GA to various problems [21]. Furthermore, substantial empirical data is available for the different parameter settings required by GAs and this greatly helps the choice of appropriate parameters for a specific problem to be solved [46]. This, together with the many books [16, 26] that exist on genetic algorithms, makes it easier for researchers to learn how to adapt genetic algorithms to their context. Second, being a global search algorithm, GAs have been shown to usually perform better than local search algorithms [53], though there is no evidence showing that GA is better than other global search algorithm [21]. Last, GAs have many well known implementations in the form of commercial tools [42] and frameworks [2, 34], which greatly facilitate their practical application.

5.2 RQ2: How are the empirical studies in search-based software testing designed and reported?

The purpose of this research question is to investigate and assess the design and reporting of empirical studies in the domain of search-based software testing. To answer this question, we further divided this question into five sub-questions. By answering each sub-question individually, we will answer the main research question. Though the results are presented in tables that summarize the main findings, the reader can obtain a break-down of which papers led to these findings in the technical report [5] corresponding to this paper.

5.2.1 RQ2.1: How well is the random variation inherent in search-based software testing, accounted for in the design of empirical studies?

We discussed the necessity and importance of accounting for random variation and using appropriate data analysis methods in Section 3.3. To assess whether random variation has been accounted for, we classified the papers into two main categories: (1) papers which accounted for random variation in their design and reported this information and (2) papers which either did not account for random variation or did not report it well. To be classified in the first category, the study in the paper had to report the number of times the MHS algorithm was executed, sufficient information to determine whether the runs were independent, and report the data analysis methods used to compare alternative algorithms and baseline solutions. The independence of different runs can be determined in different ways in different MHS algorithms. For instance, in the case of the HC algorithm, if it is started from the same starting point in each run using the same strategy to select neighbors, then all the runs will not be independent and hence every time the algorithm will find the same solution. Different runs in HC are normally made independent by choosing different starting points in each run or by using a random strategy to select neighbors. Additionally, the number of runs for each MHS algorithm had to be at least ten, a ballpark figure to enable the application of statistical hypothesis testing with minimal statistical power. Papers that did not report the number of runs or were executed less than ten times were placed in the second category (Random Variation Not Accounted).

Within the first category, we further divided the papers according to the type of data analysis that had been performed. If only the average of the results or the percentage of successful runs over all runs was reported, then these papers were classified as having “poor” descriptive statistics (the definition of successful run varies across papers, but generally speaking, if the test target to be covered is found, then the run is considered successful. A test target, for example, could be a branch to cover). This is because the average does not convey any information about the dispersion of the results being examined. Papers which report the level of variation as well as the measures of central tendency are counted in the sub-category “good” descriptive statistics. The final category is the set of papers that in addition to reporting “good” descriptive statistics also reported the results of statistical hypothesis tests comparing MHS algorithms and baselines and establishing the statistical significance of differences. However, most of the papers did not have detailed information on sample distributions and the validity of statistical test assumptions. It was therefore usually not possible to determine if a paper used the correct

Table 4 Results of how random variation is accounted for in empirical studies

Random Variation Accounted			Random Variation Not Accounted	
Poor Descriptive Statistics	Good Descriptive Statistics	Statistical Data Analysis	Random variation not discussed or accounted for	Insufficient number of runs
24	8	7	20	5
38%	12%	11%	31%	8%

statistical procedure for a particular problem and data set.

The results in Table 4 show that 25 papers did not account for random variation. Most of these, 20 papers, either did not provide any information about the number of runs or just reported the result of one unknown run (the best or the only run). In five papers, the study was repeated less than ten times.

Amongst 39 papers which accounted for random variation, 24 papers reported only the average of the cost or effectiveness results across all runs, for example, the average number of killed mutants as an effectiveness result or the average number of iterations as a cost result. In some cases, the percentage of successful runs amongst all runs is reported instead of, or along with the average of the effectiveness results (e.g., average coverage or average mutation score). At least one measure of dispersion like standard deviation, variance, or the variation interval ([Min, Max]) was reported for eight papers. These papers are categorized as having “good” descriptive statistics. There were seven papers that reported statistical tests as well as good descriptive statistics. One or more of the following statistical tests were used: t-test, paired t-test, Mann-Whitney test, F-test, ANOVA, and Tukey test [40, 44]. There was one paper in this sub-category, which reported the use of statistical tests, but did not specify the specific test being used and did not provide any descriptive statistics. From the results, we can see that 39% of the papers did not account for random variation at all, and 38% of the papers only had “poor” descriptive statistics, so in total 77% of papers either did not account for random variation or reported it poorly. The remaining 23% of papers are divided between 12% providing only good descriptive statistics and just 11% performing some kind of statistical hypothesis testing to assess the statistical significance of differences that is whether they can be due to chance. To answer RQ2.1, this review suggests that SBST would greatly benefit from paying more attention to accounting for random variation in search heuristics and applying more rigor in analyzing and reporting cost and effectiveness results.

5.2.2 RQ2.2: What are the most common alternatives to which SBST techniques are compared?

In assessing the cost-effectiveness of any technique, the comparison baseline is an important factor. In order to classify the papers we defined four categories of comparison baselines: (1) 'Global SBST', where the baseline of comparison is a SBST technique using a global MHS algorithm, (2) 'Local SBST' includes the techniques that use a local MHS algorithm such as HC, (3) 'Non-SBST' baselines do not use a SBST technique and feature baselines such as random search, and (4) 'Not discussed' addresses papers that do not report any comparison baseline.

The comparison to non-SBST techniques or local SBST techniques serves a dual purpose: it helps determine if the problem at hand is simple enough to be satisfactorily solved by a simple search algorithm; otherwise it provides justification for why a more complex SBST technique is necessary. In addition, a simple baseline of comparison is necessary to assess the benefits of using complex SBST techniques.

As shown in Table 5, 16 studies did not discuss the comparison baseline at all. These studies did not include any kind of comparison; they usually introduced the use of a MHS algorithm for test case generation and performed an empirical study to show that the technique does indeed generate satisfactory test cases. These papers are missing the justification for why the SBST technique was necessary to address the test case generation problem at hand and how much better it actually is compared to other existing, simpler techniques that are available to solve the problem at hand.

There were 34 studies that reported 'Non-SBST' baselines within which random search is used in 24 studies, static analysis in three, greedy algorithm in three, constraint solving in one study and three studies used some other technique specific to their context. We see that random search is the most commonly used comparison baseline amongst Non-SBST techniques. There is limited use of 'Local SBST' baselines with only three studies using HC. There are many studies (33) that used Global SBST techniques as comparison

Table 5 Comparison baselines used in SBST in terms of number of papers

Global SBST baselines			**Local SBST baselines**	**Non-SBST baselines**					**Not Discussed**
GA and Ext.	SA and Ext.	Others	Hill Climbing	Random Search	Static Analysis	Greedy Algorithm	Constraint Solving	Others	
22	6	5	3	24	3	3	1	3	16

baselines. This is usually done when investigating the effects of different parameter settings of MHS algorithms. This is most evident within GA and SA where 22 studies used either GA or its extensions as baselines and six studies used SA and its extensions.

5.2.3 RQ2.3: What are the measures used for assessing cost and effectiveness of search-based software testing?

Assessing the cost-effectiveness of SBST techniques for test case generation is the main objective of empirical studies in our context. Therefore, measuring cost and effectiveness in a valid manner is a basic requirement for all empirical studies.

Effectiveness measures. As it is discussed in Section 3, effectiveness measures are categorized into two main classes: coverage-based and fault-based measures. Under the coverage-based category, we found three main sub-categories: (1) control flow based coverage criteria such as branch, statement, path, condition, and condition-decision coverage (2) data flow based coverage criteria such as all-DU coverage, and (3) N-wise coverage criteria, when SBST techniques are used for testing combinatorial designs [36]. In the category of fault-based measures, mutation analysis is the core strategy and mutation score and the number of mutants killed are measures that were found in this review.

We found some other measures for effectiveness, which are still related to the quality of the generated test cases, but do not fit into any of the above categories. In this review, these measures are labeled “Others”. Based on the papers included in this review, we identified two sub-classes among them and labeled the rest as miscellaneous. Papers in the first sub-category use different kinds of measures related to the execution time of test cases and we called these time-based measures. The second sub-category addresses the distribution of fitness values of individuals (solutions) as the measure of effectiveness (e.g., average, maximum fitness). Such a measure is usually used when the goal of a search algorithm is not finding a targeted solution, but the goal is to be as close as possible to the targeted solution. An example of such papers is in [8, 9], where the goal was stressing the real-time systems by scheduling input sequences to maximize delays in the execution of targeted aperiodic tasks. In this study, the cost is measured by fitness values, which shows how close the completion time of a specific task is to its deadline. Table 6 presents the number of papers in our review per the category of effectiveness measures.

Table 6 Distribution of effectiveness measures across empirical studies

Coverage-based measures			Fault - based measures	Others			No effectiveness measure
Control flow	Data flow	N-wise		Time-based measures	Fitness value of individuals	Miscellaneous	
43	2	2	11	6	5	3	3

The data we collected revealed 61 papers using one or more effectiveness measures in a total of 72 different effectiveness measurements across reported studies. There were three papers that did not discuss the effectiveness of the SBST technique at all. There were 47 instances (65%) that used some type of coverage criterion as the measure of effectiveness. The most often used criteria were control flow based criteria with 43 instances (60%). Among them, 23 instances (32%) used branch coverage, which is the most frequently used effectiveness measure. All-DU coverage, which is based on data flow analysis, was used in two instances and two instances used N-wise coverage as the coverage criterion.

There were 11 instances (15%) that used fault detection rate as the measure of effectiveness, where mutation analysis is used so as to report the mutation score or the number of killed mutants. In some cases, the fault-based measures are reported along with other effectiveness measures. Among the 14 instances (19%), which used the other measures for the quality of test cases, five papers used the fitness value of individuals and six papers used different kinds of execution-time based measures. Most of the time-based measures were related to CPU cycles spent for test case execution. They are used in studies which try to use SBST techniques to generate test cases that will find the best/worst case execution time of a program.

Looking at the results in Table 6, we can see that control flow based coverage criteria targeted at white-box testing are the most often used effectiveness measures and as we mentioned in the above discussion, branch coverage is the criterion that has received the most attention. As a result, this problem is now pretty well understood and there is a widely accepted, standard way of calculating fitness values based on approximation level and branch distance [37] on control flow graphs. Fault-based effectiveness measures received relatively little attention in the literature reporting SBST studies as compared to coverage-based measures. Similarly, the applications of SBST techniques to artifacts other than code are rare as white-box testing seems to have been by far the main focus.

Cost Measures. Based on the definition of cost measures in Section 3 and what we found in this review, we categorized cost measures into two main classes (1) ‘cost of finding the target’, which is related to the cost of automating test case generation and (2)

‘cost of executing the generated test suite’, which is related to the cost of test case execution. These are both relevant and complementary. Based on the measures found in the studies, the first category is classified into four sub-categories:

(a) the number of iterations

(b) the cumulative number of individuals in all iterations

(c) the number of fitness evaluations an algorithm needs to find the final solution

(d) test case generation time.

The only measure for the category of ‘the cost of executing generated test suite’ that we found in the papers was the size of the test suite, which is a surrogate measure for test execution time.

Table 7 shows that among 64 papers, seven papers did not perform any cost analysis and in the remaining 57 papers most empirical studies reported at least one cost measure in 70 different cost measurements reported across studies.

Based on the abovementioned classification, 62 instances (86%) used measures in the category “Cost of finding the target”. The most often used measure among them was the number of iterations, which is used in 27 instances (39%). A total of six instances (4%) used the number of individuals (test cases) and the number of fitness evaluations is used by 14 instances (20%) as the measure of cost. Finally, there were 15 instances (21%) that used the ‘test case generation time’ measure.

In the second main category, ‘cost of executing the final test suite’, the size of test suite was the only measure that we found and it was used in eight instances. Some of these instances, which report the number of test cases in the final solution, reported the cost of finding the target as well. In some of these instances, the target of the SBST technique was actually creating test suites with minimum size for covering a specific criterion such as a minimal test suite that exhibits pair-wise coverage [20].

Summarizing the results of cost measures, we can see that the most commonly used measure is the number of iterations. This measure is, however, the least precise measure based on the discussion in the framework in Section 3. Another conclusion is that most studies use cost measures only for comparison purposes with other alternative techniques. There are just 15 instances (21%) that used measures such as test case generation time, which conveys whether a particular technique is likely to be practical and scale up.

Table 7 Distribution of cost measures across empirical studies

Cost of finding the target				Cost of executing the final test suite	No cost Measure
Number of iterations	**Number of individuals**	**Number of fitness evaluations**	**Test case generation time**	**Size of test suite**	
27	6	14	15	8	7

5.2.4 RQ2.4: What are the main threats to the validity of empirical studies in the domain of search-based software testing?

In order to answer this question, we carefully assessed the studies using the proposed framework in Section 3. For the construct validity threats, we looked at the validity of the cost and effectiveness measures. The most frequently observed threat was using some measures of cost that have severe limitations as they are not precise. As discussed in the framework, the imprecision of cost measures such as 'the number of iterations' makes the comparison between different SBST techniques very coarse grained. In addition, measures such as the number of iterations, the number of individuals, and the number of fitness evaluations can only be used for comparison across SBST techniques and cannot demonstrate the practicality of SBST techniques. On the other hand, cost measures such as 'test case generation time', if measured as clock time, are suitable for showing the practicality of a technique under time constraints. Such measures are, however, platform dependent and therefore not easy to use for comparisons across techniques and studies.

The most frequently encountered conclusion validity threat is related to accounting for the random variation that exists in the results obtained from SBST techniques. As discussed in RQ2.1, 39% of the papers did not take the random variation of results into account and 38% did not analyze or report it properly. This leads to a frequent threat regarding the statistical significance of the results. Therefore, not accounting for randomness and not applying proper data analysis (Section 3.3 and RQ 2.1) makes it very difficult to confidently draw practical conclusions from the results reported in most studies. Moreover, among the 11% of papers that discussed statistical hypothesis tests, just one paper has discussed the practical significance of differences that is whether differences among techniques justify the use of more complex techniques.

Regarding internal validity threats, the most important concern is the instrumentation of code and the use of different tools for data collection without reporting sufficient information about them. If the data collection and code instrumentation is not done through

a well-identified and available tool, then detailed information about the process of data collection should be reported. An example of this would be the use of a tool that instruments the code to collect, for instance, branch coverage information. If the tool is developed for experimentation purposes only and has not been thoroughly tested, then the coverage information generated by the tool might not be reliable and hence lead to an internal validity threat. A possible way to deal with this validity threat is to use readily available (open source, downloadable, or commercial) tools for this purpose.

The lack of clearly defining the target SUTs and having a clear object selection strategy are the most common threats to external validity. Usually the algorithms are executed on very small programs and no clear justification is provided for their choice and why they may be representative of the target domain, if specified. This can result in invalid generalization of the results.

5.2.5 RQ2.5: What are the most frequently omitted aspects in the reporting of empirical studies in search-based software testing?

In the previous sections, we have discussed the lack of properly reported descriptive statistics and statistical hypothesis testing (statistical significance) as the most commonly missing aspects in many empirical studies. Only 23% of the reviewed papers reported proper descriptive statistics or statistical significance results. In addition to this aspect, as discussed in the framework, there are other aspects that are also important and should be reported. These aspects are: discussion of validity threats, specification of formal test hypotheses, object selection strategy, parameter settings, and data collection method. For validity threats, 10% discussed conclusion validity, 6% discussed external validity, 3% discussed construct validity, and only 3% of the papers discussed internal validity threats. We found that only two papers out of 64 specified formal hypotheses, 44% of the papers discussed object selection strategies, and 39% of the papers described their data collection methods. Parameter settings (see [5]) were discussed by most, but not all of the papers (88%). However, all papers did not discuss all parameters required for their study; usually there is only a partial discussion. In some cases the authors provide justification of why they chose particular values for the parameters but this was rare.

Summarizing the above information, Table 8 depicts the most frequently omitted aspects in the reporting of empirical studies. Not reporting this information makes the full interpretation of the results very difficult. For example, poor reporting may make it difficult to determine whether differences are statistically significant, and whether

differences are expected to matter in practice. It is also usually difficult to determine if results can be generalized and to what domain.

5.2.6 Conclusion

In our context, defining good and relevant cost and effectiveness measures is a prerequisite for a useful empirical study. Almost all of the papers use appropriate (though not perfect) cost and effectiveness measures to perform empirical studies. However, there were two major problems in the majority of the papers. First, most of the papers do not account for the random variation in cost and effectiveness of SBST techniques. Even the majority of the papers that did account for the random variation didn't use proper data analysis and reporting methods (descriptive statistics and statistical hypothesis testing). Thus, there is a general lack of rigor in the statistical analysis and reporting of results in most empirical studies assessing the use of MHS algorithms for test case generation. Second, most of the papers didn't demonstrate the benefits of SBST by comparing it with simpler, techniques such as random search or HC. These two factors are highly important for yielding interpretable empirical studies in the context of test case generation using SBST techniques. Furthermore, many other relevant aspects of empirical studies such as the reporting of validity threats, the definition of formal hypotheses, the object selection strategy, and data collection methods are not reported by most of the papers. We can therefore conclude that most empirical studies in the context of test case generation using SBST techniques are still not properly conducted and reported and that improving this situation should be an important objective of the research community for future studies.

Table 8 The most omitted aspects of empirical studies

The most omitted aspects in the reporting of empirical studies		**Number of papers**	**Percentage**
Good Descriptive statistics and statistical test		15	23%
Validity threats	**Construct**	2	3%
	Internal	2	3%
	Conclusion	7	10%
	External	4	6%
Formal Hypothesis		2	3%
Object selection strategy		28	44%
Data collection method		25	39%

5.3 How convincing are the reported results regarding the cost, effectiveness, and scalability of search-based software testing techniques?

There is a lot of research being conducted on test case generation based on MHS algorithms. In order to draw general conclusions from the current body of work, we need to assess how convincing is the evidence regarding the cost, effectiveness, and scalability of SBST techniques. The first step is to clearly identify studies that provide complete and credible evidence from an empirical standpoint. Credible results are the consequence of a well designed and conducted empirical study. Based on the discussions in Section 3, a well designed study in the context of SBST should account for the random variation present in the results and have a meaningful comparison baseline to show that the targeted test problem benefits from a MHS approach. Therefore, in order to answer this research question, we first selected papers that at a minimum account for the random variation of results and compare their technique with the results of a simpler, non-SBST technique (such as random search, static source code analysis, or some other technique applicable to the test problem under consideration) or with HC. The first sub question, RQ3.1, will provide an overview of these papers. The second step to answer RQ3 is to select those papers that performed and reported proper data analysis. To satisfy this criterion, we expect papers to report descriptive statistics on the variation in the results (cost, effectiveness), where relevant or results of statistical hypothesis testing comparing alternative test case generation algorithms, and in particular MHS algorithms with simpler baseline alternatives. We deemed this set of papers as having credible evidence regarding the cost, effectiveness, and scalability of SBST. In sub question RQ3.2, we provide detailed information about the cost and effectiveness results presented in these papers along with a short description of the test problem that they tackled.

5.3.1 RQ3.1: For which metaheuristic search algorithms, test levels, and fault types is there credible evidence for the study of cost-effectiveness?

This sub-question provides a summary of the research papers that met the minimum criteria of accounting for random variation in results and performing comparisons with a simpler non-SBST or local SBST techniques. Out of the 64 papers that we analyzed, we found 39 that accounted for random variation of results. This number was reduced to 18, after selection of only those papers that also had either a non-SBST or a simple, local MHS comparison baseline. Thus, based on the criteria that we used, we had to exclude 46 papers as not being applicable for answering our research question. It is worth mentioning that there were 14 papers among those 46 discounted papers that had the minimum requirement

of accounting for random variation, but did not have a non-SBST or local MHS comparison baseline. For example, they may have proposed an extension to a genetic algorithm that would possibly enhance its capacity for test case generation and compared their results to a genetic algorithm not having this extension. In this review, those studies are not considered as credible evidence, since they do not show, in any way, that a simple non-SBST technique such as random search or a local MHS such as HC could not, in this particular context, equal or outperform their technique. This is an important consideration, since there is no a priori reason to believe that a MHS algorithm is more cost-effective and efficient than simpler algorithms in all test case generation contexts. The size of the search space is only a weak indicator of the extent of the search challenge as the search difficulty also depends on the search space landscape and distribution of satisfactory solutions across this space. Table 9 summarizes this set of 18 papers in terms of the MHS algorithms used, the testing levels, and the fault types targeted in the empirical studies. These papers are referred to as 'Minimum Criteria papers' in Table 9.

As can be seen in Table 9, amongst the 18 papers that report credible evidence, most papers (13 out of 18) applied a SBST technique at the unit testing level. The most commonly investigated MHS algorithm is the genetic algorithm with 12 papers out of 18, followed by simulated annealing with just four papers. This trend is the same as that observed in the full set of 64 papers in Section 5.1 There are also only two papers that target specific faults, one targeting functional faults and the other non-functional faults.

5.3.2 RQ3.2: How convincing is the evidence of cost and effectiveness of search-based software testing techniques, based on empirical studies that report credible results?

Along with accounting for random variation in the results and having a non-SBST or local MHS comparison baseline, studies must also report proper descriptive statistics or statistical hypothesis testing results in order to present credible and interpretable evidence. After the application of these criteria, there were just eight papers left and the results of these papers, referred to as 'Sufficient Criteria Papers', are summarized in Table 10.

Based on the information presented in Table 10, it is apparent that there is a scarcity of convincing evidence regarding the cost-effectiveness of SBST techniques. Nevertheless, these papers are a representative sample from the different types of investigations that are performed with MHS algorithms for test case generation. MHS algorithms have been recently applied to increasingly diverse types of problems and this is seen in this sample of papers by comparing the content of the "test purpose" column across papers. This ranges

from specialized purposes such as testing the performance of real time systems to more general purposes such as testing non-public methods in object-oriented programs. Despite the diversity of objectives, we can see that in most of these papers, MHS algorithms, mostly GA, were compared with random search and the results show that GA outperformed random search for the test case generation problems at hand. This suggests that this type of problems indeed requires guided search algorithms. It would also be interesting to see how the quality of the empirical studies that have been performed in this field have improved over the years. In order to investigate this, we compare three series as shown in Figure 4.

The 'All papers' series shows the number of papers per year expressed as a percentage of the total number of papers (64 papers). The 'Minimum Criteria papers' series shows the percentage per year of the papers satisfying our minimum criterion of accounting for

Table 9 Test levels, fault types, and the type of metaheuristic algorithms used by 'minimum criteria papers'

	Test Level			Fault Type		Type of Metaheuristic Search Algorithm						
Paper	**Unit**	**Integration**	**System**	**Non-Functional**	**Functional**	**GA**	**EGA**	**SA**	**ESA**	**ACO**	**GP**	**PSO**
Jones *et. al.* [28]	√	–	–	–	–	√	–	–	–	–	–	–
Puschnerand Nossal [43]	√	–	–	–	–	√	–	–	–	–	–	–
Tracey *et. al.* [47]	√	–	–	–	–	-	–	√	–	–	–	–
Bueno and Jino [10]	√	–	–	–	–	√	–	–	–	–	–	–
Michael *et. al.* [38]	√	–	–	–	–		√	–	–	–	–	–
Wegener *et. al.* [51]	√	–	–	–	–	√	–	–	–	–	–	–
Shiba *et. al.* [45]	–	–	√	–	–	√	–	–	–	√	–	–
Briand *et. al.* [8, 9]	–	–	√	√	–	√	–	–	–	–	–	–
Miller *et. al.* [39]	√	–	–	–	–	√	–	–	–	–	–	–
Watkins and. Hufnagel [50]	√	–	–	–	–	√	–		–	–	–	–
Zhan and Clark [54]	–	–	√	–	√	–	–	√	–	–	–	–
Zhan and Clark [55]	–	–	√	–	–	–	–	√	√	–	–	–
Bueno *et. al.* [11]	–	–	√	–	–	–	–	–	–	–	–	√
Harman *et. al.* [33]	√	–	–	–	–	–	√	–	–	–	–	–
Harman and McMinn [24]	√	–	–	–	–	√	–	–	–	–	–	–
Harman *et. al.* [22]	√	–	–	–	–	√	–	–	–	–	–	–
Wappler and Schieferdecker [49]	√	–	–	–	–	√	–	–	–	–	–	–
Xiao *et. al.* [53]	√	–	–	–	–	√	–	√	√	–	–	–

random variation (as reported in Table 9 and the 'Sufficient Criteria papers' series shows the percentage per year of papers satisfying our secondary criteria of having an appropriate baseline and proper descriptive statistics or results of statistical hypothesis testing (as reported in Table 10). From Figure 4 we can see that 40% of all papers, 55% of all minimum criteria papers and 88% of all sufficient criteria papers were published in recent years (2006 and 2007). The trends that become apparent are that firstly, the number of SBST publications has been steadily growing over the years, and secondly, that the quality of empirical studies has increased dramatically in recent years.

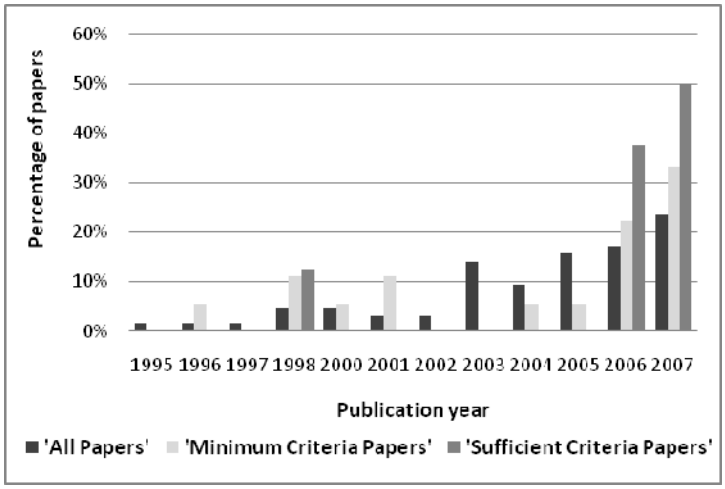


Figure 4 Quality trends of SBST empirical studies based on the publication year

5.3.3 RQ3.3: Is there any evidence regarding the scalability of metaheuristic search algorithms for test case generation?

During our systematic review, we did not find any paper specifically targeting the scalability of the MHS algorithm in the context of SBST. However, there was one paper where the authors performed a small scale scalability analysis [53]. The study was conducted on five small test objects written in C/C++. There were 36 to 87 test requirements to achieve full condition-decision coverage for all test objects and the size of the search space ranged from 26 to 232. The study was performed using different algorithms including GA, SA, Genetic Simulating Annealing (GSA), SA with Advanced Adaptive Neighbors (SA/AAN), and random search. In two of the SUTs used for the study, two different search spaces (one small and one large) were used to measure the performance (condition-decision coverage vs. the number of SUT iterations) of different MHS algorithms and random search. Based on the empirical evaluation, it was concluded

Table 10 Test purposes, comparison baselines, and result highlights for the 'sufficient criteria papers'

Paper	Test purpose	Comparison baseline	Result highlights
Puschner and Nossal, 1998	Creating an input data set with the worst-case program execution time	RS BEDG StA	In most cases, GA performed equal to or better than RS in terms of effectiveness measured as execution time of the SUT. For smaller size SUTs, GA had results as good as BEDG and StA
Briand *et. al*., 2005 and 2006	Stressing a real-time system by creating input sequences that maximize delays in the execution of target tasks and increase chances of missing deadlines.	ScA	The technique can schedule tasks to miss the deadline(s) even though schedulability analysis identified them as schedulable. The GA is successful in bringing task completion times closer to their deadlines, thus leading to stressing the system in that respect.
Miller *et. al*., 2006	Test case generation using genetic algorithms and program dependence graphs.	RS, GA	1) The results showed that, for simple programs there is little difference in the results (branch coverage) between RS and their proposed GA approach (TDGen). 2) The difference is seen in larger programs, where a much smaller number of generations are required to achieve 100% branch coverage. 3) It is also observed that for some SUTs, TDGen can achieve 100% branch coverage, where RS and GADGET cannot.
Watkins *et. al*., 2005	Comparison of different fitness functions for path coverage	RS	Based on the study, it was concluded that there is no single fitness function that works well in all cases. A two-step method using two best fitness functions is therefore suggested in the paper.
Harman and McMinn, 2007	Test data generation to answer three research questions formulated based on royal road theory (see [24]) for GA	RS, HC	1) GA was able to find inputs to exercise the branches that have royal road features and HC and RT were not successful at all. 2) GA was unable to find the inputs to exercise the branches that have royal road features if crossover operators were removed. 3) HC performed better or no worse than GA for the branches that do not have royal road features.
Harman *et. al*., 2007	Investigation of the relationship between the size of the search space (consisting of test inputs) and the performance of search algorithms measured as the number of fitness evaluations to cover a branch	RS, HC	1) There is no relationship between search space reduction and reduction in cost for random search. 2) There is significant improvement in cost reduction for both hill climbing and the genetic algorithm. 3) The reduction in cost is more for the genetic algorithm than for hill climbing. 4) There is no relationship between search space reduction and search effectiveness in terms of coverage for any of the search algorithms.
Wappler and Schieferdecker, 2007	An approach for testing non-public methods without breaking the encapsulation of the class, using an objective function specifically designed to cover non-public methods via public methods.	RS, GP	The new GP technique achieved higher overall branch coverage than RS and higher coverage of non-public methods than their existing GP based approach.
Xiao *et. al*., 2007	Empirical evaluation of different MHS algorithms and RS for test data generation.	GA, SA, two extensions of SA (SA/AAN, GSA), RS	GA performed better than all other algorithms including random search. After GA, SA/AAN performed better in terms of both cost (number of SUT executions) and effectiveness (condition decision coverage).

HC: Hill Climbing, RS: Random Search, GA: Genetic Algorithm, SA: Simulated Annealing, GP: Genetic Programming, SA/AAN: SA with Advanced Adaptive Neighbors, GSA: Genetic SA, ScA: Schedulability Analysis, BEDG: Best Effort Data Generation, StA: Static Analysis

that GA performed well for both the small and the large search space. SA/ANN was the second best. SA and GSA performed well only for the small search space. All MHS algorithms performed better than random search. As a result, we can say that scalability analyses of SBST techniques in the domain of test case generation are very rare and there is a need to focus more on scalability analysis in future studies.

5.3.4 Conclusion

Based on the discussions in the three sub-questions above, the number of papers which contain well-designed and reported empirical studies in the domain of test case generation using SBST is very small. As a result, there is a limited body of credible evidence that demonstrates the usefulness of SBST techniques for test case generation. This evidence is, in addition, very partial as it mostly focuses on the use of genetic algorithms at the unit testing level. This evidence, however, consistently shows that the genetic algorithms outperform random search in terms of structural coverage. However, this evidence is just based on eight papers and cannot be generalized to state that genetic algorithms at the unit testing level will always outperform random search regardless of the test objectives. More empirical studies must be conducted to provide strong and generalizable evidence about the suitability and applicability of different MHS algorithms for test case generation at different testing levels and for test objectives other than structural coverage.

6 Threats to the Validity of this Review

The main validity threats to our review are related to the possible incomplete selection of publications, inaccuracy of data extraction, and bias in quality assessment of studies.

6.1 Incomplete selection of publications

In Section 4.2, we have discussed and justified the systematic and unbiased selection strategy of publications. However, it is still possible to miss some relevant literature. One such instance is the existence of grey literature such as technical reports and PhD theses. In our case, this literature can be important if the authors report the complete study which is briefly reported in the corresponding published paper. In this review, we did not include such information.

Another instance that may lead to an incomplete selection of publications is the difficulty of finding an appropriate search string. In Section 4.2 we provide justification for the repositories that we selected and the search string that we used. However, there may

still be some papers, which have used some other related terms other than our keywords. We refined our search string several times because we found a paper missing from our selected papers, which was in the reference list of another paper. In order to deal with this problem, we refined our search string until it included all such papers and we were sure that our set of selected papers did not miss any paper that is referred to and relevant for this review.

6.2 Inaccuracy in data extraction

Inaccurate data can be the result of subjective and unsystematic data extraction or invalid classification of data items. In our review, we tried to deal with this problem by two means. First, we defined a framework, which clearly identified the data items that should be extracted. Second, all the data extracted was reviewed by three researchers and all discrepancies were settled by discussion to make sure that the extraction was as objective as possible. Therefore, the remaining problem is the validity of the framework itself. We have defined the framework based on the current guidelines for empirical studies in software engineering and adapted them to our domain of interest based on experience. Hence, we believe that it is a good starting point, but it can be further improved by feedback and discussion from other researchers in the domain.

6.3 Unbiased quality assessment

Assessing the quality of the papers for answering RQ3 was a challenging issue. Even though the data extracted from the papers to judge their quality was detailed and based on a well thought framework, the criteria used to select the papers themselves could be thought of as subjective. Our justification for the validity of this criterion is discussed in the Section 5.3 and we re-emphasize the fact that this is the minimum requirement for having a valid empirical study in the domain of SBST.

7 Conclusion

The automation of test case generation has been a long-standing problem in software engineering. Search-based software testing, or in other words the application of metaheuristic search (MHS) algorithms for test case generation, has shown to be a very promising approach for solving this problem by re-expressing test case generation problems as search problems. As a result, a great deal of research has been conducted and published. The time was therefore ripe to perform a systematic review of the state of the art

and appraise the evidence regarding the cost-effectiveness of such an approach. A systematic review is very different from more informal, traditional surveys, in the sense that it aims at being comprehensive in its coverage and repeatability by relying on well-defined paper selection and analysis procedures. This systematic review focuses, due to space constraints, on one specific but crucial aspect: the way SBST techniques have been empirically assessed. This aspect is highly important as all MHS algorithms are heuristics and therefore cannot guarantee their success in solving a test case generation problem or any other problem for that matter. Only an empirical investigation can provide the necessary confidence that a specific MHS algorithm is appropriate for a given test case generation problem.

In addition to a large-scale, systematic review, our contribution also includes guidelines, in the form of a framework, on how to conduct empirical studies in search-based software testing. Results of our review have shown that the research reported so far has mostly focused on structural coverage and unit testing. However, the research is increasingly more diversified in the types of topics being tackled. Results also show that empirical studies in this field would benefit from more standardized and rigorous ways to perform and report studies. More specifically, three important empirical issues stand out from our analysis. Studies need to, more systematically and rigorously, account for the random variation in the results generated by any MHS algorithm. Such random variation implies that alternative techniques can only be compared by statistical means, that is, statistical hypothesis testing. This, unfortunately, is not performed well in most published papers and our framework provides guidelines about which statistical test to perform in which circumstance. Last, another important issue is that it is impossible to assess how a MHS technique performs in absolute terms: to be able to conclude on its usefulness to tackle a specific test case generation problem, a proposed technique needs to be compared with simpler and existing alternatives to determine whether it brings any advantage. This is again missing in an important number of papers and needs to be carefully addressed by all studies in the future.

Despite the above limitations, credible results are available and existing results confirm that MHS algorithms are indeed promising for solving a wide variety of test case generation problems. Future research work will have to better establish their limitations and the types of problems for which they are applicable and required.

Acknowledgment

The authors wish to thank Simula School of Research and Innovation (SSRI) for funding this work.

References

[1] "Computer Science Conference Ranking," 2008, http://www.cs-conference-ranking.org/conferenceranking s/topicsii.html.

[2] "Genetic Algorithms Framework," Rubicite Interactive, 2004, http://sourceforge.net/projects/ga-fwork.

[3] "UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE)," Object Management Group (OMG), 2008, http://www.omg.org/cgi-bin/doc?ptc/2008-06-08.

[4] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology,* vol. 51, pp. 957-976, 2009.

[5] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A Systematic Review of the Application and Empirical Investigation of Evolutionary Testing," in *Technical Report Simula.SE.293*: Simula Research Laboratory, 2008.

[6] B. Beizer, *Software testing techniques* Van Nostrand Reinhold Co., 1990.

[7] A. Bertolino, "Software testing research: achievements, challenges, dreams," in *2007 Future of Software Engineering*: IEEE Computer Society, 2007.

[8] L. C. Briand, Y. Labiche, and M. Shousha, "Stress testing real-time systems with genetic algorithms," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '05)* Washington DC, USA: ACM, 2005.

[9] L. C. Briand, Y. Labiche, and M. Shousha, "Using genetic algorithms for early schedulability analysis and stress testing in real-time systems," *Genetic Programming and Evolvable Machines,* vol. 7, pp. 145-170, 2006.

[10] P. M. S. Bueno and M. Jino, "Identification of potentially infeasible program paths by monitoring the search for test data," in *Proceedings of the fifteenth IEEE international conference on Automated Software Engineering (ASE '00)* 2000, pp. 209-218.

[11] P. M. S. Bueno, W. E. Wong, and M. Jino, "Improving random test sets using the diversity oriented test data generation," in *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)* Atlanta, Georgia: ACM, 2007.

[12] E. K. Burke and G. Kendall, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*: Springer 2006.

[13] K. Y. Cai and D. Card, "An analysis of research topics in software engineering - 2006," *Journal of Systems and Software,* vol. 81, p. 8, 2008.

[14] J. A. Capon, *Elementary Statistics for the Social Sciences*: Wadsworth Publishing Co Inc, 1988.

[15] J. Clarke, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd, "Reformulating software engineering as a search problem," *IEE Software* vol. 150, pp. 161-175, 2003.

[16] D. A. Coley, *An Introduction to Genetic Algorithms for Scientists and Engineers*: World Scientific Publishing Company, 1997.

[17] R. Drechsler and N. Drechsler, *Evolutionary Algorithms for Embedded System Design*: Kluwer Academic Publishers, 2002.
[18] T. Dyba, V. B. Kampenes, and D. I. K. Sjoberg, "A systematic review of statistical power in software engineering experiments," *Information and Software Technology,* vol. 48, pp. 745-755, 2006.
[19] T. Dyba, B. A. Kitchenham, and M. Jorgensen, "Evidence-based software engineering for practitioners " *IEEE software,* vol. 22, p. 8, January/February 2005 2005.
[20] S. A. Ghazi and M. A. Ahmed, "Pair-wise test coverage using genetic algorithms," in *The 2003 Congress on Evolutionary Computation (CEC '03)* 2003, pp. 1420-1424.
[21] M. Harman, "The Current State and Future of Search Based Software Engineering," in *2007 Future of Software Engineering*: IEEE Computer Society, 2007.
[22] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, and J. Wegener, "The impact of input domain reduction on search-based test data generation," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering* Dubrovnik, Croatia: ACM, 2007.
[23] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology,* vol. 43, pp. 833-839, 2001.
[24] M. Harman and P. McMinn, "A theoretical empirical analysis of evolutionary testing and hill climbing for structural test data generation," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07)* London, United Kingdom: ACM, 2007.
[25] C. Hart, *Doing a Literature Review: Releasing the Social Science Research Imagination*: Sage Publications Ltd, 1999.
[26] R. L. Haupt and S. E. Haupt, *Practical Genetic Algorithms*: Wiley-Interscience, 1997.
[27] D. Johnson, "A theoretician's guide to the experimental analysis of algorithms," in *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges,* 2002, pp. 215-250.
[28] B. F. Jones, H. H. Sthamer, and D. E. Eyres, "Automatic structural testing using genetic algorithms," *Software Engineering Journal,* vol. 11, pp. 299-306, 1996.
[29] K. S. Khan, R. Kunz, J. Kleijnen, and G. Antes, *Systematic Reviews to Support Evidence-Based Medicine: How to Review and Apply Findings of Healthcare Research*: Royal Society of Medicine Press Ltd, 2003.
[30] B. A. Kitchenham, "Guidelines for performing Systematic Literature Reviews in Software Engineering," 2007.
[31] B. A. Kitchenham, T. Dyba, and M. Jorgensen, "Evidence-based software engineering," in *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*: IEEE Computer Society, 2004.
[32] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary Guidelines for Empirical Research in Software Engineering," *IEEE Transactions on Software Engineering,* vol. 28, p. 14, August 2002.
[33] K. Lakhotia, M. Harman, and P. McMinn, "A multi-objective approach to search-based test data generation," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '07)* London, England: ACM, 2007.
[34] S. Luke, L. Panait, G. Balan, S. Paus, Z. Skolicki, E. Popovici, K. Sullivan, J. Harrison, J. Bassett, R. Hubley, and A. Chircop, "A Java-based Evolutionary

Computation Research System," George Mason University's ECLab Evolutionary Computation Laboratory, 2007, http://www.cs.gmu.edu/~eclab/projects/ecj/.
[35] T. Mantere and J. T. Alander, "Evolutionary software engineering, a review," *Applied Soft Computing*, vol. 5, pp. 315-331, 2005.
[36] A. P. Mathur, *Foundations of Software Testing*: Pearson Education, 2008.
[37] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, p. 52, 2004.
[38] C. C. Michael, G. McGraw, and M. A. Schatz, "Generating software test data by evolution," *IEEE Transactions on Software Engineering*, vol. 27, pp. 1085-1110, 2001.
[39] J. Miller, M. Reformat, and H. Zhang, "Automatic test data generation using genetic algorithm and program dependence graphs," *Information and Software Technology*, vol. 48, pp. 586-605, 2006.
[40] D. S. Moore and G. P. McCabe, *Introduction to the Practice of Statistics*, Fourth ed.: W. H. Freeman, 2002.
[41] I. H. Osman and J. P. Kelly, *Metaheuristics: Theory and Applications*: Kluwer Academic Publishers, 1996.
[42] H. Pohlheim, "GEATbx - The Genetic and Evolutionary Algorithm Toolbox for Matlab," 2007.
[43] P. Puschner and R. Nossal, "Testing the results of static worst-case execution-time analysis," in *Proceedings of the 19th IEEE Real-Time Systems Symposium*, 1998, pp. 134-143.
[44] D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures*, Third ed.: Chapman & Hall/CRC, 2003.
[45] T. Shiba, T. Tsuchiya, and T. Kikuno, "Using artificial life techniques to generate test cases for combinatorial testing," in *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC '04)* 2004, pp. 72-77.
[46] M. Srinivas and L. M. Patnaik, "Genetic algorithms: a survey," *Computer*, vol. 27, pp. 17-26, 1994.
[47] N. Tracey, J. Clark, and K. Mander, "Automated program flaw finding using simulated annealing," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '98)* Clearwater Beach, Florida, United States: ACM, 1998.
[48] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '93)* Cambridge, Massachusetts, United States: ACM, 1993.
[49] S. Wappler and I. Schieferdecker, "Improving evolutionary class testing in the presence of non-public methods," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated Software Engineering (ASE '07)* Atlanta, Georgia, USA: ACM, 2007.
[50] A. Watkins and E. M. Hufnagel, "Evolutionary test data generation: a comparison of fitness functions," *Software: Practice and Experience*, vol. 36, pp. 95-116, 2006.
[51] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, pp. 841-854, 2001.
[52] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering: An Introduction*: Kluwer Academic Publishers, 2000.

[53] M. Xiao, M. El-Attar, M. Reformat, and J. Miller, "Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques," *Empirical Software Engineering*, vol. 12, pp. 183-239, 2007.

[54] Y. Zhan and J. A. Clark, "Search-based mutation testing for Simulink models," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '05)* Washington DC, USA: ACM, 2005.

[55] Y. Zhan and J. A. Clark, "The state problem for test generation in Simulink," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '06)* Seattle, Washington, USA: ACM, 2006.

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

Hadi Hemmati, Lionel Briand, Andrea Arcuri, Shaukat Ali,

Published in the proceedings of the 18th ACM International Symposium on Foundations of Software Engineering (FSE), pp. 267-276, 2010

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.

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, or 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 (e.g., code or model 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 [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 (GAs) [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 our 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 less demanding coverage criterion (i.e., a 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 [11], 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]. 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 [14], and execution traces (e.g., call stack [15]) 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 [16] 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 [17-20]. For example, in [11] a Greedy search selects, at every step, the test case that covers the most uncovered statements whereas in [12, 19] 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* [21]". 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 a 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. The 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 [22] where GAs are the most used and successful reported technique [22]. A more comprehensive study of selection algorithms will be part of our future work. Though further details on how we have employed a GA in a test case selection context will be discussed in Section 4, we provide below minimum background information on GAs.

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 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, minimization, and prioritization 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 techniques 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 [23] the similarity function is based on all def-use pairs coverage and they use a classification algorithm as a reduction 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 [24], [13], and [21, 25]. Greedy search, adaptive random selection, and clustering are used in these studies for selection/prioritization. In [26] 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 prioritization 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 [27]. In this 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 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 *et al.* in [9] can be considered as potential baselines of comparison for our study.

Some empirical studies [13, 26] 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, 23] do not have a comparison with coverage-based techniques, 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 fixed sample size *n* and test suite *TS* that detects a set of faults (*F*) in the system, our goal is to find a subset of *TS* of size *n* (s_n) with maximum $FD(s_n)$, where $FD(s_n)$ 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 $SimMsr(s_n)$:

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

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 (no guard). Note that tp_3 has no similarities with tp_1 and tp_2 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* ($SimMsr(s_n)$). We do not tune our GA parameters and use what is suggested in the literature to increase the applicability of the approach in industry. The selection mechanism is the rank selection which has been shown to work well [28]. The population size is set to 50 since population sizes between 20 and 80 have shown promising results for different search space sizes [29]. We also have tried some other sizes in this range and the GA was not very sensitive to the changes. A single point crossover is used 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, a high probability as suggested in [8]) 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:

$$s_4^1 =< tp_1 , tp_2 , tp_3 , tp_4 > and\ s_4^2 =< tp_a , tp_b , tp_c , tp_d >$$
$$\acute{s}_4^1 =< tp_1 , tp_2 , tp_c , tp_d > and\ \acute{s}_4^2 =< tp_a , tp_b , tp_3 , tp_4 >$$

The selected mutation operator is similar to what is typically used for bit strings [8, 29]. 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 $s_4^y =< tp_1 , tp_2 , tp_5 , tp_d >$ can be a mutated version of s_4^x. 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 GAs in this study: (1) stopping after specific number of iterations, (2) stopping after a fixed period of time (e.g., 1 second (sec)), 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 (ms)). However, in this paper, we only report the result taken from experiments with a fixed execution time stopping criterion since we wanted to keep cost constant when comparing GAs with Greedy search. 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*
$\left(s_n^{\acute{x}}, s_n^{\acute{y}}\right) :=$ *crossover* $(s_n^x, s_n^y, P_{xover})$
Mutate$(s_n^{\acute{x}}, s_n^{\acute{y}})$
If valid $(s_n^{\acute{x}}, s_n^{\acute{y}}) \wedge$ *min* $(f(s_n^{\acute{x}}), f(s_n^{\acute{y}})) \leq$ *min* $(\mathrm{f}(s_n^x), \mathrm{f}(s_n^y))$
Then $s_n^x := s_n^{\acute{x}}$ *and* $s_n^y := s_n^{\acute{y}}$

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. A total of 26 faults were found. They 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. The faults are due to both code and design level faults and belong 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. To which extent is using a GA for test case selection more cost-effective (in terms of time spent to find a solution) compared to a Greedy search?

RQ3. To which extent 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 a 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 non-parametric (*Mann-Whitney*) statistical tests, with a significance level $\alpha = 0.05$, to compare the *FDR* distributions of the proposed and alternative selection techniques. The *Mann-Whitney U-test* is more robust than a parametric test such as the *t*-test when there are strong departures from normality and a large enough sample of observations (in our case 100). In addition, we provide *FDR* means and medians over different runs.

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* distributions as Boxplots.

(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) [30] 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 1 and 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, Table 1 summarizes means and medians of ρ(i) for these techniques and reports the *Mann-Whitney U-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

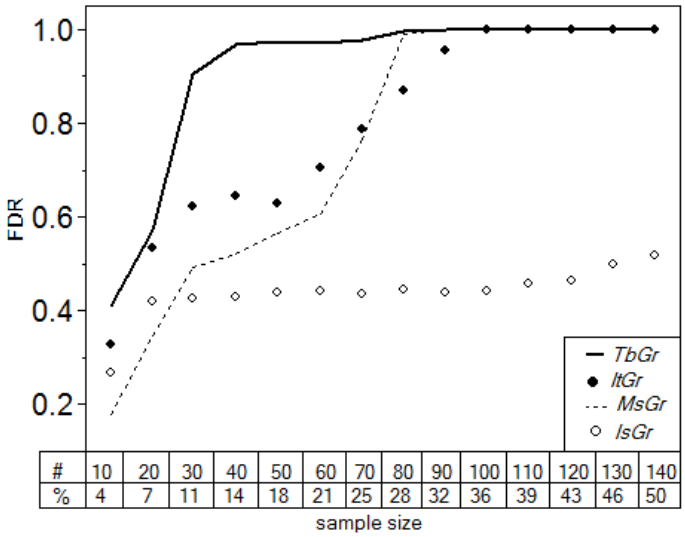


Figure 1 The average FDR of *TbGr, ItGr, MsGr, IsGr* for different sample sizes

Table 1 RQ1: The median and mean *FDR*s per sample size (10 to 100 by intervals of 10) over 100 runs and the *Mann-Whitney U-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 by intervals of 10) over 100 runs and the *Mann-Whitney U-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

that there are normally 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) $\rho(i)_{TbGr}$ is always higher than $\rho(i)_{ItGr}$ 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* ($AFDR_{140}^{10}(TbGr) \cong 0.88$ vs.

$AFDR_{140}^{10}(ItGr) \cong 0.76$). 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: $\min_{95}(Tb) \cong 35 \cong (12\% \text{ of the test suite})$ 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 in the worst case, selecting a subset of test cases can lead to a very low fault detection. In the next section we show how using GAs can help increase our confidence in *Tb* by decreasing its variance.

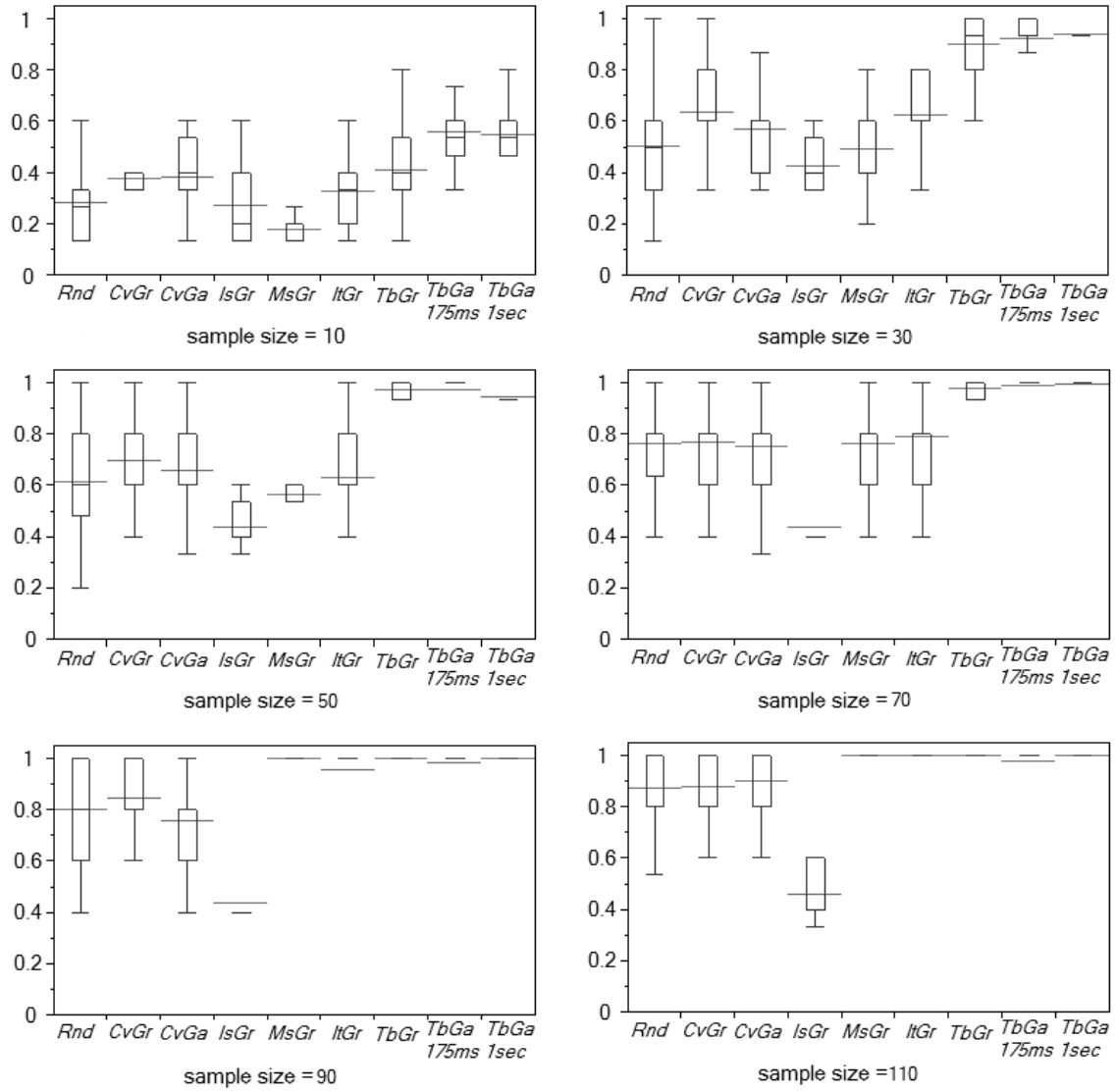


Figure 2 FDR (y-axis) Boxplots for different selection techniques (x-axis) for sample sizes ranging from 10 to 110 by intervals of 20 over 100. The Boxplots show the 10th, 25th, 50th, 75th, and 90th percentiles and means

6.3.2 To which extent is using a GA for test case selection 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 GAs 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 $\binom{281}{28} \cong 2.9*10\text{^}38$.

For a given sample size and a selection technique (here a GA vs. a Greedy both using the *Tb* similarity measure), our effectiveness measure is the FDR of the selected test cases and the cost is measured in 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 the 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 the GA which can potentially be improved within practical bounds.

Using our proposed similarity measure *Tb*, we investigate the extent to which a GA can improve *FDR compared to using Greedy search*. Using execution times of 175ms (as for Greedy search), Figure 2 and Figure 3 show *FDR* distributions for the 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 the 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 the GA to be as effective as Greedy search. The main reason is that for larger sample size the 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 the 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 the GA (Figure 2), especially for sample

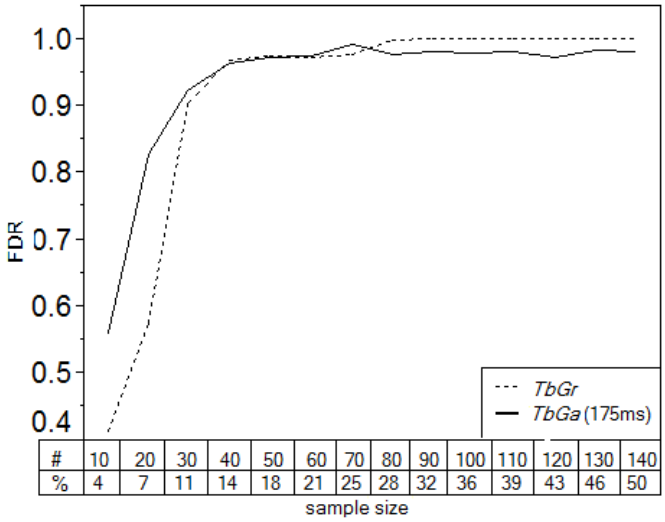


Figure 3 The average FDR of TbGr and TbGa(175ms) for different sample sizes

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, the 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 the 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 the GA for longer execution times does not seem to produce significantly better results. An explanation could be within 175ms the 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 the GA needs less time for smaller sample sizes. Therefore, the GA running 175ms starts to perform slightly worse than the GA running 1000ms for subsets larger than 70 (~25% of the test suite), as illustrated in Figure 2 (sample sizes>70).

6.3.3 To which extent 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 a GA for coverage-based selection as it is defined in [12, 19]. 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 ($\rho(i)_{TbGa}$, $\rho(i)_{ItGr}$, $\rho(i)_{CvGr}$, $\rho(i)_{Rnd}$) 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.

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

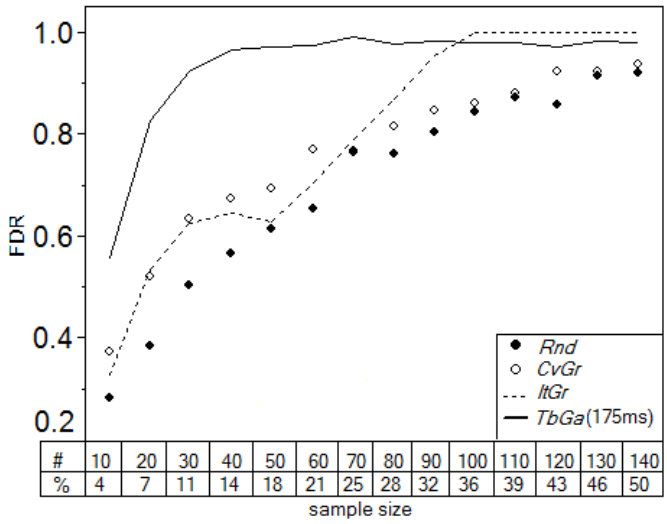


Figure 4 The average FDR of Rnd, CvGr, ItGr, TbGa(175ms), for different sample sizes

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 [31] 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 algorithm and the GA and strived to achieve the same level of optimization. The 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. However, that would compromise the applicability of the approach as tuning can be time consuming and difficult. 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 *Mann-Whitney U-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. The GA uses parameter settings recommended by studies in the literature and is therefore easy to apply. 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.

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, 510-514.

[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 18th international symposium on Software testing and analysis*, 2009, 201-212.

[14] 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.
[15] McMaster, S. and Memon, A., *Call-Stack Coverage for GUI Test Suite Reduction, IEEE Transactions on Software Engineering*, 34(1), 2008, 99-115.
[16] 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.
[17] 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, 569-574.
[18] 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, 141-149
[19] 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.
[20] 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.
[21] 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, 442-456.
[22] Harman, M., *The Current State and Future of Search Based Software Engineering*, In *Proceedings of the Future of Software Engineering*, 2007, IEEE Computer Society, 342-357.
[23] 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, 93-96.
[24] 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, 233-244.
[25] 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.
[26] 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, 667-673.
[27] http://www.levenshtein.net/
[28] 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, 116-121.
[29] Haupt, R. L. and Haupt, S. E., *Practical Genetic Algorithms*, Wiley, 1998.
[30] Gusfield, D., *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.
[31] 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.

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

Hadi Hemmati and Lionel Briand

Published in the proceedings of the 21st IEEE International Symposium on Software Reliability Engineering (ISSRE), pp. 141-150, 2010

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.

1 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 2 reports on background information about test case selection. Section 3 discusses on different similarity functions which are used in this study. Section 4 provides a brief overview of related works covering STCS techniques. Section 5 reports the experimentation results of applying different STCS techniques on an industrial case study. Section 6 concludes the paper and outlines our future work plan.

2 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.

2.1 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.

2.2 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).

3 Similarity Function

As we mentioned in Section 2.2, 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.

3.1 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].

3.1.1 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*.

3.1.2 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.

3.1.3 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:

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

3.2 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.

3.2.1 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.

3.2.2 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.

4 Related Work

As we discussed in Section 2, 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 2.2 (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 3 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.

5 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.

5.1 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) [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.

5.2 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 $0 \leq AFDR_m^{\gamma}(\Gamma) \leq 1$. 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, 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.

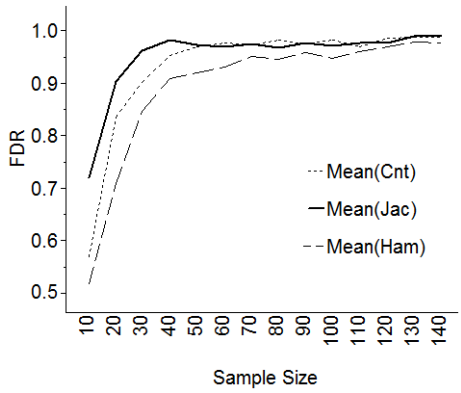
The three measures above are complementary and help interpreting the FDR from different angles. The experiments have been conducted on a PC with Intel Core(TM)2 Duo CPU 2.40 Hz and 4 GB memory running Windows 7.

5.3 Experiment results

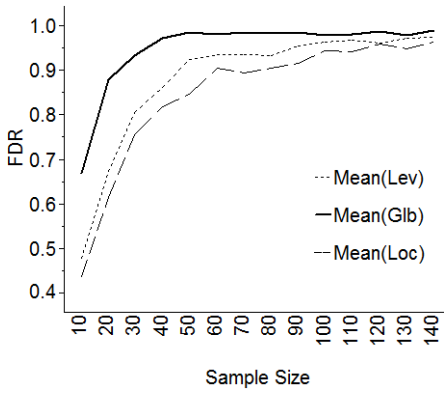
In this section we answer research questions RQ1 and RQ2 based on our case study.

5.3.1 Experiment results for RQ1

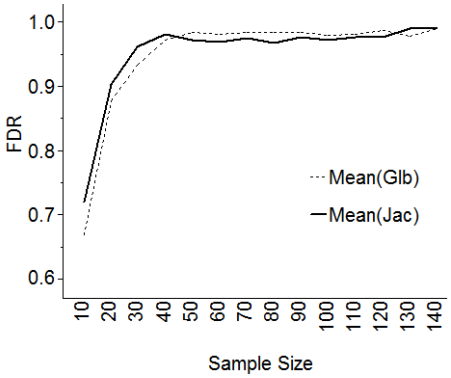
We start with RQ1.1 and first compare the effectiveness of set-based and sequence-based techniques separately and identify the best function in each class. We then compare the best set-based similarity versus the best sequence-based function. Figure 1.a shows the average FDR of the three set-based functions introduced in Section 3.1 ($\rho(i)_{Cnt}$, $\rho(i)_{Jac}$, $\rho(i)_{Ham}$). 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 $\min_{90}(\text{Jac}) \cong 20$ (~7% of the test suite) whereas $\min_{90}(\text{Cnt}) \cong 30$ (~11% of the test suite) and $\min_{90}(\text{Ham}) \cong 40$ (~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



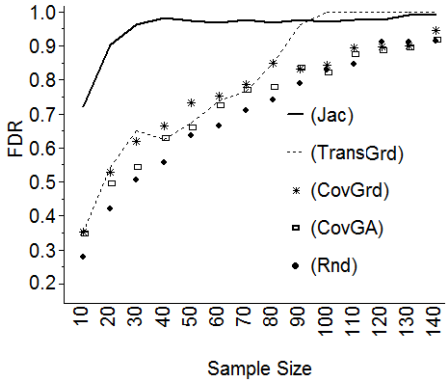
(a) Set-based similarity functions



(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

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 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 3.2 ($\rho(i)_{Lev}$, $\rho(i)_{Glb}$, $\rho(i)_{Loc}$). 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.

5.3.2 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

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	
		M	S	M	S	M	S	M	S	M	S	M	S
A	**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
B	**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
C	**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

M: Mean, S: Standard Deviation. A: Set-based, B: Sequence-based, C: Baselines

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.

5.4 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 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.

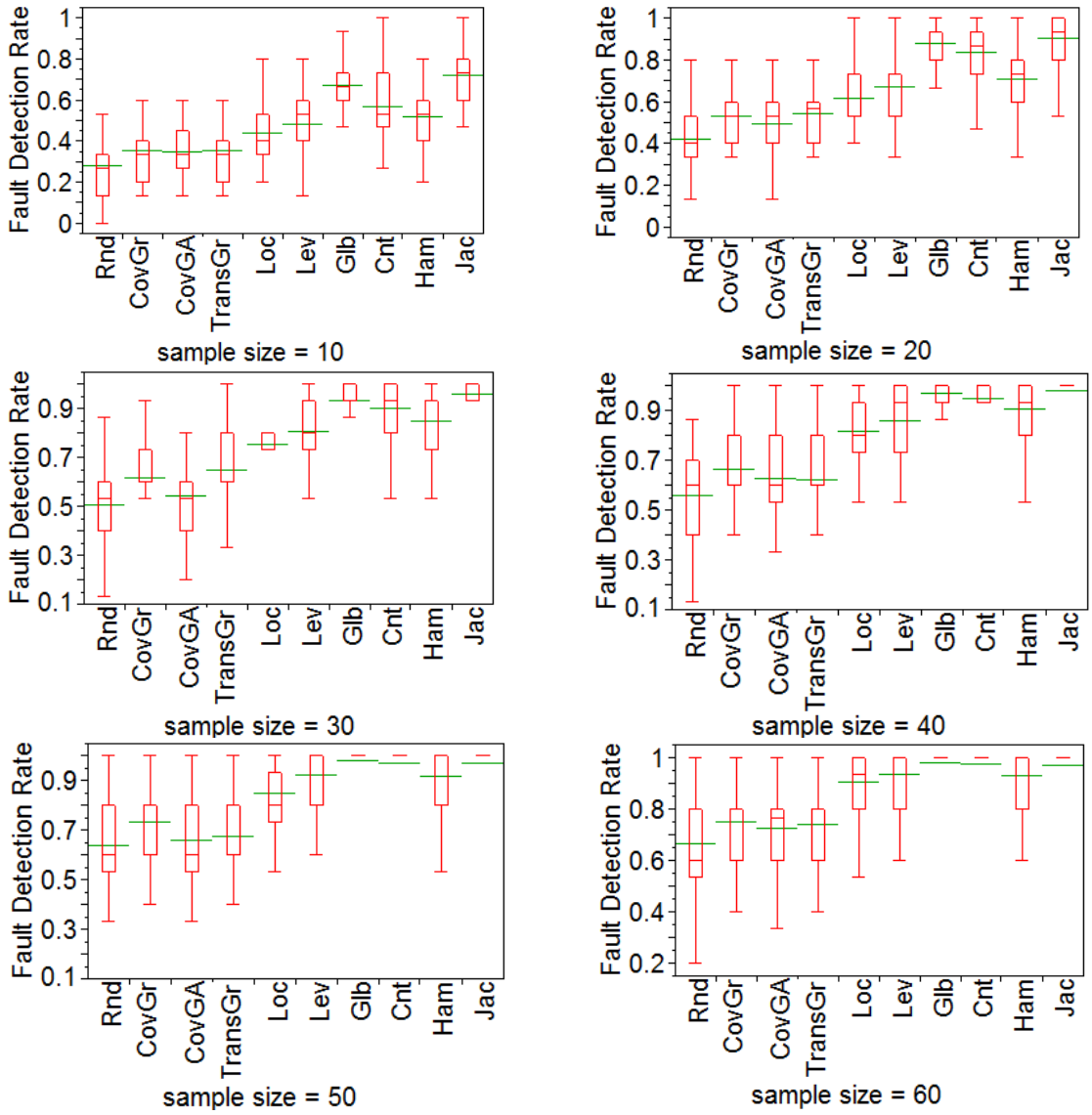


Figure 2 FDR (y-axis) Boxplots for different selection techniques (x-axis) for sample sizes ranging from 10 to 60 by intervals of 10 over 100 runs. The Boxplots show the 10th, 25th, 50th, 75th, and 90th percentiles and means

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.

6 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.

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.

Reducing the Cost of Model-Based Testing through Test Case Diversity

Hadi. Hemmati, Andrea. Arcuri, and Lionel. Briand

Published in the proceedings of the 22nd IFIP International Conference on Testing Software and Systems (ICTSS), formerly TestCom/FATES, pp. 63-78, 2010

Abstract— Model-based testing (MBT) suffers from two main problems which in many real world systems make MBT impractical: scalability and automatic oracle generation. When no automated oracle is available, or when testing must be performed on actual hardware or a restricted-access network, for example, only a small set of test cases can be executed and evaluated. However, MBT techniques usually generate large sets of test cases when applied to real systems, regardless of the coverage criteria. Therefore, one needs to select a small enough subset of these test cases that have the highest possible fault revealing power. In this paper, we investigate and compare various techniques for rewarding diversity in the selected test cases as a way to increase the likelihood of fault detection. We use a similarity measure defined on the representation of the test cases and use it in several algorithms that aim at maximizing the diversity of test cases. Using an industrial system with actual faults, we found that rewarding diversity leads to higher fault detection compared to the techniques commonly reported in the literature: coverage-based and random selection. Among the investigated algorithms, diversification using Genetic Algorithms is the most cost-effective technique.

1 Introduction

In The idea of model-based testing (MBT) [1] is to generate executable test cases by systematically analyzing specification models (e.g. represented as UML state machines) following a test strategy such as a coverage criterion, that aims to cover certain features of the model (e.g., all transitions). MBT brings many advantages but also entails the additional cost of modeling the software under test (SUT). In addition, there are two factors that significantly increase the cost of MBT: (1) the lack of automated oracle (e.g., when assessing the subjective perception of a media quality in a videoconference system), and (2) the high cost of test case execution (e.g., when testing must be performed on actual hardware or a restricted-access network). In both situations, the test suite must be as small

as possible while, to the extent possible, preserving its fault revealing power. However, for real world size models, MBT techniques usually generate large sets of test cases regardless of the applied coverage criteria. Therefore, a model-based technique is required to select an optimal subset of test cases to be executed, which is, in general, a NP-hard problem.

In similarity-based test case selection, the idea is to diversify the selected test cases with respect to a similarity measure. In [2, 3], we proposed a similarity-based selection technique for testing based on UML state machines (SMBT). We compared different similarity measures in terms of what information from the test cases they have to evaluate (test case encodings) and how this evaluation should be done (similarity functions). The results showed that, in the context of SMBT, the similarity measure that represents a test case as a set of trigger-guards [2] and uses Jaccard Index [4] as the similarity function [3] is the most effective measure in terms of fault detection rate (FDR).

In this paper, we take a deeper look into the idea of diversifying test cases and investigate why similarity-based selected test cases are effective in finding faults. We also study different strategies that, given a similarity measure and a test suite, we can use to select a subset of the test suite. We applied our experiments on an industrial software system and a set of actual faults, and the results clearly showed that rewarding diversity is effective. The main explanation is that the test cases that find different faults belong to distinct clusters based on the similarity measure. In addition we found that, among different selection strategies, Genetic Algorithms (GAs) [5] are the most cost-effective technique for similarity-based test case selection. We also have shown that, in our case study, we could save up to 80% of test case executions, and get more than 99% FDR, by using a GA compared to a coverage-based selection technique.

The rest of the paper is organized as follows. Section 2 introduces the similarity-based test case selection technique. Section 3 discusses the different strategies which are used in this paper to diversify test cases. Section 4 provides a brief overview of related works covering similarity-based selection techniques. Section 5 reports the experimentation results of applying the selection techniques on an industrial case study. Section 6 concludes the paper and outlines our future work plan.

2 Similarity-based Test Case Selection

Unlike coverage-based selection, where the goal is maximizing the coverage of the test model by the selected test cases (e.g. transition coverage in SMBT) to maximize chances

of fault detection, similarity-based selection techniques maximize diversity among the selected subset. Diversity is calculated using a (dis)similarity measure between pairs of test cases. A similarity measure is the value that a similarity function assigns to two inputs which are being compared. In a testing context, inputs are usually encoded as a set or sequence of elements. In the context of MBT, the inputs are abstract test cases defined on the test model rather than concrete test cases. We do not use the execution information of the test case as, in our context, the goal is to select them before execution. Abstract test cases are naturally generated as a first step by MBT and can hide 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 (e.g., event parameter values). Therefore, to compare different techniques, it is necessary to run the selected test paths with different input data and analyze their FDR distribution.

Representation (encoding) of the test cases has an important effect on the similarity measure. Though in SMBT a test path represents an encoded abstract test case, the test path can be described at different levels of details. In [2], we studied three encodings for a test path in UML state machine: state-based, transition-based, and trigger-guard-based, and reported that trigger-guard-based encoding is the most effective one in terms of fault detection, where a test path(*tp*) is represented as:

<tp> ::= <TrGu> | <TrGu> "," <tp>
< TrGu > ::= *trig* |*guard* | *id* | *guard* "+" *trig*

where *trig* is the identification of a trigger, and *guard* is the identification of a guard in the state machine. In this representation, 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 (*id*) as its identifier.

Given an encoding, one may use different similarity functions to calculate the similarity value. In [3] we studied different set-based and sequence-based similarity functions and proposed Jaccard Index as the most cost-effective. Given a set of n 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 similarity-based selection is using a strategy to select a subset of test cases with minimum average pair-wise similarity (*SimMsr*). In the rest of this paper, we focus on finding the best strategy for this selection.

3 Strategies for Maximizing Diversity

Given a similarity measure we have two strategies to select the most diverse test cases. One is based on clustering test cases and taking samples from each cluster and the second is searching for the most diverse subsets. In this section, we introduce one clustering and two search techniques that will be investigated.

3.1 Clustering-based techniques

Clustering algorithms divide data instances into natural groups by maximizing their internal homogeneity and external separation [6]. Regardless of the specific algorithm which is used for clustering, most clustering techniques use a proximity measure as a mean to determine the closeness (similarity), or dissimilarity (distance) between pairs of instances and pairs of clusters.

In this study, we are using one of the simplest clustering algorithms, which has been frequently used in software engineering, including software testing [7]: Agglomerative Hierarchical Clustering (AHC) [6]. AHC starts with forming clusters containing each exactly one object (a test case in this study). A sequence of merge operations is then performed until the desired number of clusters is achieved. At each step, the two most similar clusters will be joined together. The measure that we used for assessing similarity between two clusters, inter-cluster similarity, is Average Linkage and it is defined as the average of all pair-wise similarities between all instances of those two clusters [6]. After applying clustering, we need a sampling technique for selecting one or more test case per cluster. We use one-per-cluster sampling where the number of clusters is the same as the selected sample size and then randomly select one member from each cluster. The pseudo-code of the employed AHC follows:

(1) Make one cluster (C_k) per test path (tp_i).

(2) While the number of clusters is more than *sampleSize*

(3) Find the two most similar clusters C_x and C_y (with the maximum *InterClusterSim*(C_x, C_y)). Where:

$$InterClusterSim(C_x, C_y) = \frac{\sum_{tp_i \in C_x \wedge tp_j \in C_y} SimFunc(tp_i, tp_j)}{|C_x| * |C_y|}$$

(4) Merge the two clusters.

3.2 Test case selection using Adaptive Random Testing

Another technique that we investigate is Adaptive Random Testing (ART), which has been proposed as an extension to Random Testing [8]. Its main idea is that diversity among test cases should be rewarded, because failing test cases tend to be clustered in contiguous regions of the input domain. This has been shown to be true in empirical analyses regarding applications whose input data are of numerical type [8]. Recently, Object-Oriented software has been also shown to manifest such a property [9]. Therefore, ART is a candidate selection strategy in our context as well. In this paper, we use the basic ART algorithm described in [8], but we ensure that no replicated test case is given in output. The pseudo-code for ART is:

(1) Z={}

(2) Add a random test case to Z

(3) Repeat until |Z|= *sampleSize*

(4) Sample K random test cases that are different from Z

(5) For each of these test cases k

(6) k.maxSim = max(SimFunc(k , z ∈ Z))

(7) Add the k with minimum maxSim to Z

3.3 GA-based test case selection

A GA [5] 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 [10]. GAs are the most used and successful reported technique in this domain [10] and 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, selected 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. Eventually, after a number of generations, an individual that solves the addressed problem will be evolved.

In this paper, we use a steady state GA as a selection technique, in which only the offspring that are not worse than their parents are added to the next generations. An individual in our context is a subset of size n from the original test suite (denoted s_n). Given a similarity function $SimFunc(tp_i , tp_j)$, the fitness function f to minimize is the sum, for all pairs (tp_i , tp_j) in s_n, of $SimFunc(tp_i , tp_j)$, denoted $SimMsr$. We use a single point crossover with probability of P_{xover} to combine two different parents s_n^x and s_n^y. A mutated test path is replaced by a test path that is selected at random from the set of all possible test paths. A valid solution is a set of test cases in which there is no duplicate. We have applied two types of stopping criteria for the GA in this study: (1) stopping after specific number of fitness evaluations, and (2) stopping after a fixed period of time (e.g., 350ms). The pseudo-code of the employed GA follows:

(1) Sample a population G of m sets of test cases uniformly from the search space (i.e., the set of all possible valid sets with a given size n)
(2) Repeat until the stopping criterion is met
(3) Choose s_n^x and s_n^y from G
(4) $(\acute{s}_n^x , \acute{s}_n^y) :=$ crossover $(s_n^x , s_n^y, P_{xover})$
(5) Mutate$(\acute{s}_n^x, \acute{s}_n^y)$
(6) If valid $(\acute{s}_n^x , \acute{s}_n^y) \wedge \min (f(\acute{s}_n^x), f(\acute{s}_n^y)) \leq \min (f(s_n^x), f(s_n^y))$
(7) Then $s_n^x := \acute{s}_n^x$ and $s_n^y := \acute{s}_n^y$

4 Related Work

There are three approaches reported in the literature to select a subset of test cases from a test suite that can be applied in our context: (1) Random or semi-random selection [11],

where there is no guidance to select test cases; (2) Coverage-based selections, where we hypothesize that "the test cases which have more coverage are more likely to detect faults" (e.g., in [12] a Greedy search selects, at every step, the test case that covers the most uncovered statements whereas in [13, 14] a GA is used to find the maximum coverage.); (3) Similarity-based selections try to diversify test cases, given a similarity measure, assuming that maximizing diversity among the selected test cases maximizes the number of detected faults.

Diversifying test cases has been studied on code-based test case selection and mostly in the context of regression testing. The similarity measure in such cases is usually based on code coverage [7, 15-18]. In [19] a sequence of memory operations is used to calculate the similarity and in [20] the authors use the whole test script in string format as the input for similarity function. The work in [21] is the only one where the similarity function is based on model-level information. Test cases are represented as sequence of transitions in a LTS model of the system and the number of identical transitions in the sequence is the similarity function. Our similarity measure is different from theirs, both in terms of encoding and similarity function—we use trigger-guard sets on UML state machines and apply the Jaccard Index. In [2, 3] we have compared the effectiveness of our similarity measure with the measure in [21] and the results showed a great improvement using our technique, which therefore is applied in this study as well. Given a similarity measure, different strategies have been used to diversify the selected subsets: Greedy search in [17, 19-21], Neural network based classification in [18], ART in [17], AHC in [7, 15, 16]. In this paper, using our similarity measure, we compare ART, AHC, and a GA.

5 Empirical Evaluation

5.1 Case study description

The SUT under study is a typical safety monitoring component in a safety-critical control system implemented in C++ and modeled as UML state machines complemented by constraints specifying state invariants and guards. This SUT is typical of a broad category of reactive systems interacting with sensors and actuators. The first and the subsequent maintained versions of the system (including models and code) were developed and verified by company experts and our research team. The correct and the most up-to-date UML state machine, representing the latest version of the SUT's behavior, consists of 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.

The correct latest UML state machine was given to our test case generation tool (TRUST) [22] as an input model. Using All-Transitions coverage, 281 test paths and corresponding executable test cases were automatically generated. 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. As it is typical in many embedded systems, the average execution time for these test cases is in the order of minutes, which makes running all the 281 test cases very time consuming.

We use 15 faulty versions of the code that are made by introducing one real fault per program. The 15 faults used in the study were introduced during maintenance activities by developers and re-introduced for the purpose of the experiment in the latest version of the SUT. Each of these faults belongs to one of the following categories: wrong guards on transitions, wrong state invariant, missing transition, and wrong OnEntry action in states. 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 for the 15 faulty versions 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.

5.2 Experiment design

In our industrial case study, we investigate the following research questions:

RQ1. Why does diversifying test cases improve fault detection?

- **RQ1.1.** Do test cases that find the same faults tend to be more similar to each other than with other test cases?
- **RQ1.2**. Do test cases that find different faults tend to be more different from each other than test cases that find the same faults?

RQ2. What is the most cost-effective way to diversify (given our similarity measure) a set of test cases?

- **RQ2.1.** Does clustering-based test case selection improve the average FDR compared to coverage-based and random selection?
- **RQ2.2.** Are search-based techniques more cost-effective than clustering-based selection in terms of fault detection?

RQ3. How cost-effective is diversifying test cases compared to state of practice techniques for test case selection?

In RQ1 we are analyzing why diversifying test cases improves FDR. In other words, are test cases distinctly clustered with respect to different faults? We have carried out an exhaustive analysis based on our industrial case study. Given N=281 test cases, we ran all of them on the actual SUT and all its faulty version to check which of the M faults they are able to detect (in our case study M=15). We then calculated the similarity of each pair of test cases, for a total of N*(N-1)/2 pairs. Note that the exhaustive analysis of the search space landscape is based on the similarity values of all test case pairs. However, test case selection is performed for any arbitrary *sampleSize* where using an exhaustive search is not an option, since the search space size for selecting a subset of size *sampleSize* is equal to the number of possible *sampleSize* combinations within a test suite of a given size. In our case, as an example, the search space size for *sampleSize* =28 (~10% of the test suite) is 2.9*10^38.

To address RQ1, we investigate two hypotheses: (1) For each fault cluster, the similarity between pairs of test cases that find the same faults is, on average, significantly higher than the similarity of other test case pairs in the test suite, and (2) For each pair of fault clusters, the similarity between test cases that find different faults is significantly lower than the similarity of test case pairs that find the same fault in the test suite. If hypothesis (1) holds, then test cases finding the same faults will cluster in close areas of the test case space. As a result, rewarding diversity in test case selection would be beneficial. But hypothesis (2) should also hold, otherwise diversity might be harmful since we would need more than one test case from the same area to detect all faults.

In RQ2, we are interested in how to diversify the test cases, given the similarity measure used in RQ1. Our baselines of comparison are random selection (Rnd) and a coverage-based selection technique (CovGr) which is based on one of the most used selection techniques in the literature: it applies a Greedy search to maximize the coverage of the selected test cases [12]. In this paper, in each step of the Greedy search in CovGr, we look for the test cases which cover the most yet uncovered transitions on the UML state machine representing the SUT. Finally, in RQ3 we look at the practical benefits of our proposed approach based on our industrial SUT. In this study, as mentioned in Section 3, AHC is used as our clustering algorithm and a GA and ART as search-based techniques. Our measure of effectiveness is the FDR of the selected subset from the original test suite. Ideally, given the same amount of computational cost, we would say that a technique is

better than the other if it obtains higher average FDR. For practitioners, such cost would typically be measured as the time that an algorithm takes before completing its task. Comparing algorithms using time is not a robust option from a practical standpoint though. Low-level implementation details may have a strong effect on computational time. If we use time as stopping criterion, then we may not truly compare algorithms but instead their implementations [23]. To cope with this problem, a measure that is independent from implementation details would be useful. For example, when comparing search algorithms, it is a common practice to allow each algorithm to run until a maximum number of fitness evaluations is executed (e.g., 100,000 [24]). However, the assumption here is that the total search cost is proportional to the number of fitness evaluations and the cost of other operations than fitness evaluation is either equal or negligible in both algorithms.

To compare GAs with ART, following the same general reasoning, we use the number of similarity comparisons (C) as stopping criterion, where *n* is the size of the output test case set. We hence can run both the GA and ART with the same preset number of similarity comparisons. For a GA that runs for W fitness evaluations (each consisting of Q similarity comparisons), we have that C(GA) = W * Q = W * n * (n-1)/2 whereas for ART we have [8]: C(ART) = K * n * (n-1)/2.

We would like to run both ART and the GA such that C(ART)=C(GA), but that might not be possible because K (the size of the candidate set in ART) is a constant that is upper bounded by N (281 in our case). In other words, the basic ART cannot be run for an arbitrary amount of computational resources as it is the case for GAs (for which we can choose arbitrarily high values for W). To cope with this problem, we can just run ART several independent times (e.g., J times), and then take the best result out of these J runs. Therefore, to obtain fair comparisons using similarity measures, we can simply enforce W=J*K.

Whenever we could not use a fair metric (as the number of fitness evaluations) for comparing different algorithms for test selection, we used the time expressed in milliseconds as stopping criterion, which is the time spent by our implementation of the algorithms on a PC with Intel Core(TM)2 Duo CPU 2.40 Hz and 4 GB memory running Windows 7. As we previously discussed, though this is not particularly robust in general, it is a reasonable option in our context as a significant effort was made to optimize implementations and the execution environment was stable.

To account for the randomness of the results, which exists for all selection algorithms, we ran each experiment 100 times and analyzed distributions. We report the results for

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. (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 and typically detect most faults. We have performed non-parametric (Mann-Whitney U-test) statistical tests, using a significance level of 0.05, to compare the FDR distribution 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 for large enough samples, as this is the case in this study (100 observations).

5.3 Experiment results

5.3.1 Why does diversifying test cases improve fault detection?

For each of the M=15 faults, we calculated the similarity of the test case pairs that both found each of these faults (groups of test case pairs, from F1 to F15). Mann-Whitney U-tests were performed (α =0.05) to see whether there was a difference in similarity value between the pairs in F1 to F15 and the set of all remaining pairs of test cases (T - Fi). Table 1 summarizes the results where bold median values represent statistically significant differences between the distributions of these Fi with T - Fi. Note that F1 and F2, F3 and F4, and F7 to F15 are on the same table row, as they have the same descriptive statistics. This is due to the fact that most test case pairs are the same and those that are not the same have high similarity values (according to our similarity measure).

The results show that the difference is significant for the first six groups. The other groups also show a high difference in terms of mean and median but, since there are only three observations for each of those groups, we cannot get statistically significant differences. Therefore, the first hypothesis of RQ1.1: “Test cases that find the same faults tend to be more similar to each other than with other test cases” is confirmed.

To investigate RQ1.2, for each pair of fault clusters Fi and Fj, let us consider the similarity distribution (Dd) of test case pairs which belong to two different clusters, i.e., test cases that find different faults. We compare Dd with the similarity distribution (Ds) of test case pairs which both are in one of those two clusters, i.e., test cases that find the same fault. The median of Dd and Ds per cluster pair is reported above the diagonal in Table 2.
There are cases where fault clusters Fi are exactly the same, i.e., their respective faults are found by exactly the same set of test cases. Distinguishing them does not have any effect

on the FDR results (either all or none of the faults will be revealed by a selected set of test cases) and therefore such clusters are not distinguished. As a result, there are seven distinct fault clusters (labeled as A to G) matching the columns and rows of Table 2. Their mapping to the 15 fault clusters is as follows: A(F1 and F2), B(F3 and F4), C(F5), D(F6), E(F7 to F9), F(F10 to F12), G(F13 to F15).

The bold values show the cases where there is a statistically significant difference between Dd and Ds, based on a Mann-Whitney U-test. The presence of significant differences support the claim that fault clusters are far away from each other and therefore that rewarding diversity is useful. In cases where two clusters are overlapping, the size of the overlap compared to the size of their union will determine whether rewarding diversity is harmful. If the ratio of the overlapping part (intersection) over the union is high, a test case that finds one of the two faults would have a high probability of finding the other. In this case, rewarding diversity is still a reasonable option. We measure this ratio by dividing the size of two clusters' intersection |I| by the size of their union |U|: I_U=|I|/|U|. The cells below the diagonal of Table 2 report this measure per cluster pair.

Among 21 cluster pairs, 15 contain distinct clusters with significant differences between Dd and Ds. There are three clusters (E, F, and G) that only contain a few test cases (three per cluster), which are not amenable to statistical analysis and show no statistically

Table 1 Min, max, median, mean, and standard deviation of similarity values of the test cases that find the same faults

Groups	Pairs	Min	Median	Mean	Max	SD
T	39340	0.076	0.250	0.291	1.000	0.166
F1,F2	2080	0.181	**0.4**	0.432	1.000	0.173
F3,F4	91	0.375	**0.571**	0.561	0.833	0.143
F5	28	0.200	**0.464**	0.475	0.800	0.168
F6	28	0.714	**0.714**	0.714	0.714	0.000
F7 to 15	3	0.375	0.428	0.434	0.500	0.062

Table 2 Each cell above the diagonal shows the median of Dd and Ds (Dd/Ds) and each cell below the diagonal shows the overlapping measure (IU), per cluster pairs. Bold median values highlight significant differences (Mann-Whitney U-test) between the Dd and Ds

	A	B	C	D	E	F	G
A	-	**0.33/0.42**	**0.33/0.40**	**0.71/0.40**	**0.18/0.40**	**0.18/0.40**	**0.18/0.40**
B	0.21	-	**0.37/0.57**	**0.71/0.66**	**0.37/0.57**	**0.37/0.57**	**0.37/0.57**
C	0.12	0	-	0.71/0.71	**0.37/0.42**	**0.37/0.42**	**0.37/0.42**
D	0.12	0.57	0	-	**0.11/0.71**	**0.11/0.71**	**0.11/0.71**
E	0	0	0	0	-	0.37/0.42	0.37/0.42
F	0	0	0	0	0	-	0.37/0.42
G	0	0	0	0	0	0	-

significant differences. Clusters B and D which are not significantly different from each other show a high overlapping value (0.57), implying that although these clusters are not distinct, there is a 57% probability that a test case that is selected from their union can find both faults. Two cluster pairs, <A,D> and <C,D>, show unexpected results—Dd median lower than the Ds median—and they are not highly overlapping. Therefore, since among 21 pairs, 15 pairs fit the situation where similarity-based selection is effective, two do not, and four are neutral, we can conclude that, overall, in most cases "test cases that find different faults tend to be more different from each other than test cases that find the same faults".

Overall, the results of our analysis confirm that diversity in test case selection should be encouraged and that our similarity measure is adequate. It also seems that since test cases finding the same faults are clustered together and these clusters are mostly distinct, clustering algorithms are a reasonable candidate approach to achieve diversity, though we will investigate what is the best strategy in the next research question.

5.3.2 What is the most cost-effective way to diversify a set of test cases?

To answer RQ2 we first compare the AHC clustering algorithm with CovGr and Rnd introduced in Section 5.2. Figure 1 shows the FDR results of the algorithms.

Overall, the results show that for all sample sizes AHC is more effective than its two alternatives except that for sample sizes less than 30 (~10% of the test suite) the difference between the average FDRs of CovGr and AHC is not statistically significant (based on Mann-Whitney tests). Considering the fact that in practice the results for smaller sample

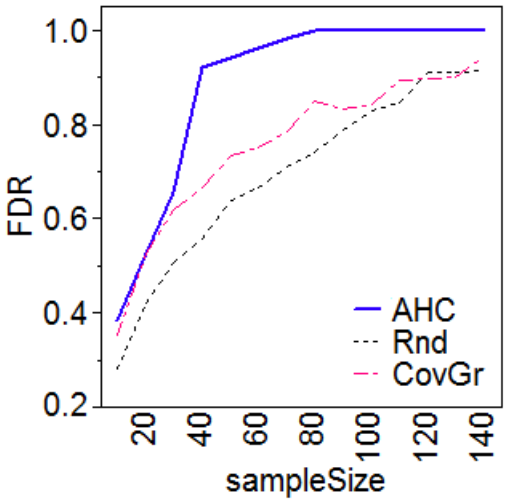


Figure 1 The average FDR of AHC, Rnd, and CovGr for different sample sizes

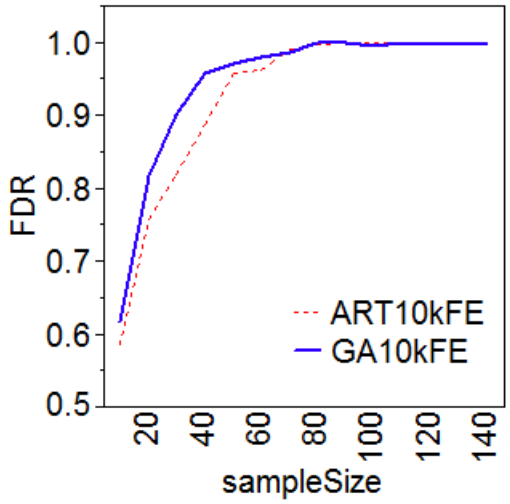


Figure 2 The average FDR of ART and the GA with 10,000 fitness evaluations for different sample sizes

sizes are more important, AHC may not be preferred to CovGr given the high cost of a clustering technique compared to simple Greedy search. On average (for all sample sizes over 100 runs) each selection requires 350ms, 10ms, and less than 1ms when using AHC, CovGr, and Rnd, respectively. Though those time differences may not seem relevant, they may become so on much larger test suites of thousands of test cases. However, for sample sizes higher than 40, there is a huge (up to 40%) improvement using AHC compared to CovGr. In addition, AHC ensures 100% FDR with 80 test cases whereas CovGr and Rnd find less than 95% of the faults even with 140 test cases.

Note that, in theory, since Rnd does not use any heuristic to increase FDR, we cannot improve it. However, we can improve CovGr by running it several time with different random selections, wherever the coverage among alternative test cases is equal, and reporting the best result out of all runs. To compare the FDR results of CovGr when it costs exactly the same as AHC, we let CovGr improve its results by random reselection and stopped the algorithm after 350ms. The results showed that in our case, there is no practically significant difference in CovGr FDR for 10 and 350 ms of running time.

Addressing RQ2.1, given that the FDR of AHC is always equal or superior to that of CovGr or Rnd, and the fact that we cannot predict for a given test suite the sample size threshold above which AHC will be certain to fare significantly better, we favor the systematic use of AHC over CovGr and Rnd. Moreover, in practice, this strategy makes even more sense when considering that test case execution time (which in our case is in the range of minutes) is usually much higher than selection time for any of the techniques (which in our case is in the range of milliseconds).

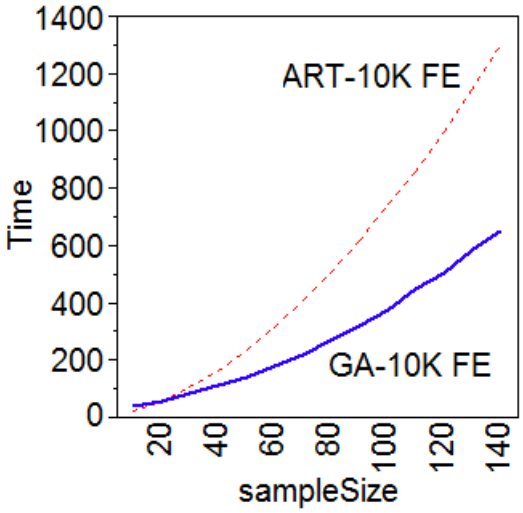


Figure 3 The time in milliseconds required by the GA and ART to run 10,000 fitness evaluations for different sample sizes

Comparing search-based techniques with AHC, first we need to find out which search technique is more cost-effective. In this study, we compare the FDR of ART and a GA. The GA is stopped after 10,000 fitness evaluations, and ART is run 1000 times with K=10 (so both algorithms use the same number of similarity comparisons). Figure 2 shows the average FDR of the techniques for each sample size over 100 runs. In general, the GA fares better and more particularly so from sample size 20 (~7% of the test suite) to 70 (~25% of the test suite). For sample sizes larger than 70, the FDR of both techniques converges to 1.0. The differences for smaller sample sizes are statistically significant but, because these differences may not practically significant (at most 10% improvement for the GA), we need to look closely at the relative cost of ART and the GA.

As we mentioned earlier, the number of fitness evaluations is usually a good platform-independent measure for the cost of search techniques. However, in our implementation, a matrix made of all pair-wise similarities is created before any search. Therefore, this overhead is the same for all search algorithms and the fitness evaluation is not an expensive part of the search. Therefore, we cannot be sure that total cost is proportional to the number of fitness evaluation. In Figure 3, we have plotted the actual time spent by the two algorithms (ART and the GA with 10,000 fitness evaluations). The required time for 10,000 fitness evaluations using both techniques is exponentially increasing and they both spend almost the same time for very small sample sizes (less than 20). For sample sizes higher than 20 (~7% of the test suite), ART quickly gets more expensive than the GA.

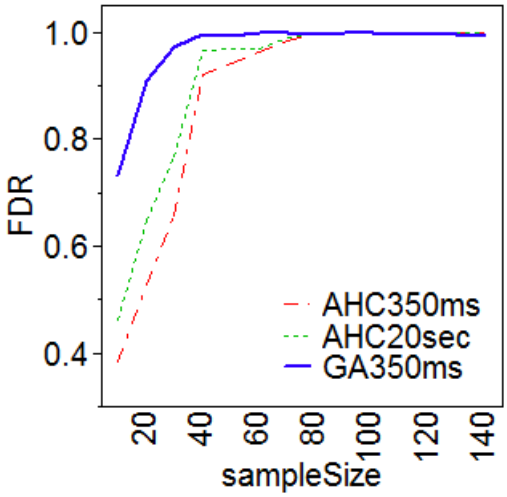


Figure 4 The average FDR of the GA with 350ms and AHC with 350 and 20,000ms for different sample sizes

Given that it always has equal or worse FDR results, there is no reason for choosing ART over the GA.

In the next step, we compare AHC with the GA but using the same execution time that AHC requires for its selection (350ms). Figure 4 shows that the GA is clearly preferable over AHC considering that spending the same time as AHC, the GA fares in general much better and can almost double the average FDR results of AHC for small sample sizes. It is also more effective in finding all faults: AHC requires 80 test cases whereas the GA only needs 40 test cases to achieve 100% FDR. To draw more conservative conclusions regarding the superiority of the GA, we even conducted another experiment and ran AHC a relatively long time (20,000ms) to compare its FDR result with the GA using 350ms (Figure 4). However, even when letting AHC work for almost 60 times longer than the GA it still yields much lower FDR. Therefore, our suggested answer to RQ2 is using the GA over the other alternatives.

5.3.3 How cost-effective is diversifying test cases compared to state of practice techniques for test case selection?

To answer RQ3, we compare our best candidate based on RQ2, which is similarity-based selection using a GA, with a coverage-based Greedy search (CovGr). Looking at Figure 1, the first observation is that the GA can save more than 80% of the test case execution cost, given the fact that the GA, on average, finds more than 99% of the faults by 40 test cases whereas CovGr requires more than 220 test cases to achieve the same (not plotted in the figure). To have a more detailed cost-effectiveness assessment, we look at the

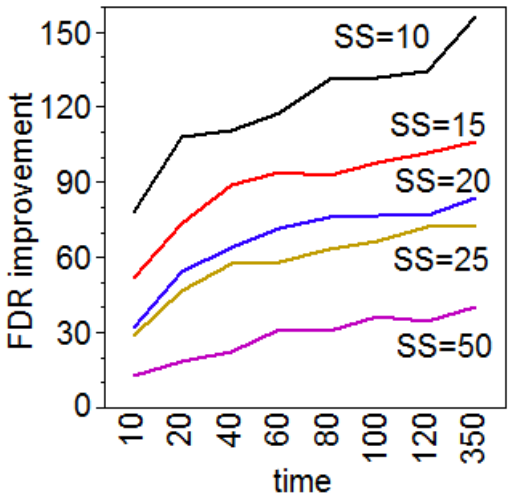


Figure 5 The percentage of improvement of similarity-based selection using GA over CovGr for different sample sizes (SS) in time range of 10 to 350ms

improvement that the GA may provide over time. Since this improvement varies over sample sizes as well, we plotted in Figure 5 the percentage of FDR improvement provided by the GA over CovGr for five sample sizes: 10, 15, 20, 25, and 50 (ranging from 4 to 18 % of the test suite), over a time period of 10ms (the average time required by CovGr to select test cases) to 350ms (the average time required by AHC to select test cases). Note that as we mentioned earlier, as opposed to the GA, CovGr does not improve over time.

A first observation from Figure 5 is that the smaller the sample size, the larger the improvement provided by the GA. Also, it is interesting to see that the GA, even with 10ms execution time, always detects more faults than CovGr. For example, the average FDR of the GA is 80% larger than the CovGr FDR for 10ms. Finally, a cost analysis shows that in cases where we can afford spending more time for selection, the GA can be greatly improved. For all five sample sizes shown in Figure 5, the GA's improvement over CovGr almost doubles if we give it 350ms instead of 10ms. This improvement over time gets very large when the sample size gets smaller. For example, for sample size 10 the GA can yield a 160% higher FDR than CovGr, which in practice is a great benefit given that the cost for this improvement is only 350ms for a test suite of 281 test cases where the cost of running one extra test case is in the order of minutes.

6 Discussion on Validity Threats

The main threats to the validity of this study are firstly the fairness of comparisons in terms of cost and secondly the generalizability of the results.

Similarity comparisons of test cases and clusters are the most influential part of selection techniques. In our implementations of the algorithms, all pair-wise similarities are pre-calculated in a similarity matrix which is given to the selection algorithm as an input parameter. Obviously, this implementation is not scalable and the similarity matrix will face memory limitations for large test suites. However, if we can afford pre-calculation, then the most expensive part of the search algorithms may not be the fitness evaluation anymore. We can see its effect on comparing ART and the GA where having the same number of similarity evaluations ART requires much more time. We have not studied on-demand similarity calculations, which might give different FDR results using the same stopping time. In addition, inter-cluster similarity calculation in AHC is very expensive and in our implementation it is repeated for each iteration of the algorithm. The code can be optimized by caching the similarities between clusters in each iteration and in the next iteration only calculate the similarities if it is not already available. However, implementing this improvement is not trivial since saving similarities of all combinations of clusters in all iterations may be not possible due to memory limitations. There is a tradeoff to be made between memory cost and execution speed.

The second issue is due to the fact that all our results and conclusions rely on a single industrial case study using a given set of real faults. Though running such studies is time consuming, it must obviously be replicated. 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 Conclusion and Future Work

In practice, executing test cases generated by model-based testing (MBT) techniques is costly. This cost is due to the large test suites which are typically generated by MBT tools on industrial-scale systems to systematically achieve a coverage/adequacy criterion. However, for system level testing, in many situations testing should take place on the deployment platform where the cost (time and resource) of each test execution may be high. This may be due to the cost of using actual hardware, potential damages in case of failure, or access to restricted infrastructure (e.g., test network). 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 cases such as the ones mentioned above, one must execute a

subset of the generated test suite whose size is dependent on context. In this paper, we propose a new approach for test case selection from UML state machines, by maximizing the diversity of the selected test cases. To measure diversity we used a specific test case representation for UML state machines (triggers-guards sets), which should be adapted in case of using other models, and a model-independent similarity function (Jaccard Index). We investigated why diversifying test cases with respect to our similarity measure increases fault detection rates and compared different strategies to diversify the test cases: Clustering, Adaptive Random Testing, and Genetic Algorithms (GAs). The results of our study on an industrial software system and actual faults showed that: (1) rewarding diversity leads to finding more faults, (2) our proposed similarity-based selection (using Jaccard Index on the set of trigger-guards with a GA selection) is the most cost-effective approach compared to the other alternatives. In addition, we showed that in practice this approach can reduce the cost of test case execution in MBT by selecting a small set of test cases which can find all (or most) faults in short amount of time. In the future, we plan to replicate the study on another industrial system. In addition, we will evaluate alternative optimization and search techniques.

References

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

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

[3] Hemmati, H., Briand, L., Arcuri, A.: Investigation of Similarity Measures for Model-Based Test Case Selection. Simula Research Laboratory, Technical Report(2010-05) (2010)

[4] Teknomo, K.: Similarity Measurement. http:\\people.revoledu.com\kardi\tutorial\Similarity.

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

[6] Xu, R., Wunsch II, D.C.: Survey of Clustering Algorithms. IEEE Transactions on Neural Netwoks 16 (2005) 645-678

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

[8] Chen, T.Y., Kuoa, F.-C., Merkela, R.G., Tseb, T.H.: Adaptive Random Testing: The ART of test case diversity. Journal of Systems and Software 83 (2010) 60-66

[9] Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: ARTOO: Adaptive Random Testing for Object-Oriented Software. 30th IEEE International Conference on Software Engineering (ICSE) (2008)

[10] Harman, M.: The Current State and Future of Search Based Software Engineering. Future of Software Engineering. IEEE Computer Society (2007) 342-357
[11] Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Professional (1999)
[12] Elbaum, S.G., Malishevsky, A.G., Rothermel, G.: Test Case Prioritization: A Family of Empirical Studies. IEEE Transactions on Software Engineering 28 (2002) 159-182
[13] Li, Z., Harman, M., Hierons, R.M.: Search Algorithms for Regression Test Case Prioritization. IEEE Transactions on Software Engineering 33 (2007) 225-237
[14] Ma, X.Y., Sheng, B.K., Ye, C.Q.: Test-Suite Reduction Using Genetic Algorithm. Advanced Parallel Processing Technologies, Vol. 3756. Springer Berlin / Heidelberg (2005)
[15] Leon, D., Podgurski, A.: A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases. 14th IEEE International Symposium on Software Reliability Engineering (ISSRE) (2003)
[16] Masri, W., Podgurski, A., Leon, D.: An Empirical Study of Test Case Filtering Techniques Based on Exercising Information Flows. IEEE Transactions on Software Engineering 33 (2007)
[17] Jiang, B., Zhang, Z., Chan, W.K., Tse, T.H.: Adaptive random test case prioritization. 25th IEEE/ACM International Conference on Automated Software Engineering (ASE) (2009)
[18] Simão, A.d.S., Mello, R.F.d., Senger, L.J.: A Technique to Reduce the Test Case Suites for Regression Testing Based on a Self-Organizing Neural Network Architecture. 30th Annual International Computer Software and Applications Conference (COMPSAC) (2006)
[19] Ramanathan, M.K., Koyutürk, M., Grama, A., Jagannathan, S.: PHALANX: a graph-theoretic framework for test case prioritization. 23rd Annual ACM Symposium on Applied Computing (2008)
[20] Ledru, Y., Petrenko, A., Boroday, S.: Using String Distances for Test Case Prioritisation. 24th IEEE/ACM International Conference on Automated Software Engineering (ASE) (2009)
[21] Cartaxo, E.G., Machado, P.D.L., 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 (2009)
[22] Ali, S., Hemmati, H., Holt, N.E., Arisholm, E., 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)
[23] Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K.: 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)
[24] Harman, M., McMinn, P.: A Theoretical and Empirical Study of Search Based Testing: Local, Global and Hybrid Search. IEEE Transactions on Software Engineering 36 (2010) 226-247

Empirical Investigation of the Effects of Test Suite Properties on Similarity-Based Test Case Selection

Hadi Hemmati, Andrea Arcuri, and Lionel Briand

To appear in the proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation (ICST), March 21-25, 2011

Abstract— Our experience with applying model-based testing on industrial systems showed that the generated test suites are often too large and costly to execute given project deadlines and the limited resources for system testing on real platforms. In such industrial contexts, it is often the case that only a small subset of test cases can be run. In previous work, we proposed novel test case selection techniques that minimize the similarities among selected test cases and outperforms other selection alternatives. In this paper, our goal is to gain insights into why and under which conditions similarity-based selection techniques, and in particular our approach, can be expected to work. We investigate the properties of test suites with respect to similarities among fault revealing test cases. We thus identify the ideal situation in which a similarity-based selection works best, which is useful for devising more effective similarity functions. We also address the specific situation in which a test suite contains outliers, that is a small group of very different test cases, and show that it decreases the effectiveness of similarity-based selection. We then propose, and successfully evaluate based on two industrial systems, a solution based on rank scaling to alleviate this problem.

1 Introduction

Rewarding diversity in test cases has been shown to lead to higher fault detection in numerical applications, because failing test cases tend to cluster in contiguous regions [1]. In previous work [2-4], we proposed similarity-based test case selection (STCS) techniques for model based testing (MBT), which applied the idea of rewarding diversity on abstract test cases generated from UML state machines.

Our motivation was that running all the system-level test cases generated by a standard criterion, e.g., round trip path for UML state machines, was not feasible, due to the high cost of running them on the deployed platform or test network. So, from a practical

standpoint, to solve the testing problems of our industrial partners, it was necessary to devise techniques to select smaller test suites. Our approach is, given a small budget of test cases that can be run, to reward diversity (i.e., penalize similarity) in the chosen test cases.

We assessed different similarity measures and diversified test cases using Genetic Algorithms (GAs) and Adaptive Random Testing (ART). The proposed techniques were applied on one industrial case study where the goal was to decrease test execution cost down to an affordable number of test cases while preserving, to the maximum extent, the fault detection rate (FDR) of the original test suite. Results showed that, compared to random and coverage-based selection, much higher FDR can be achieved when using STCS.

These promising results motivated the need to gain a better understanding of STCS, which is essential to develop novel and more effective techniques. Unlike our previous work [2-4], where we were exploring alternative STCS techniques, in this paper we analyze their variation in effectiveness, in a controlled manner and using simulation, when varying the relationship between similarity distributions and fault detection among test cases. In other words, the goal is to investigate under which circumstances STCS is more effective. The results shed light on the best and worst conditions for STCS, thus preparing the ground for improved similarity measures and STCS results.

The intuition is that STCS would perform better when test cases which detect distinct faults are dissimilar and test cases that detect a common fault are similar. Such a condition was verified [2] in one industrial case study, where we found that test cases finding a common fault were indeed clustered together in the test case space and these clusters were mostly distinct.

In this paper, to investigate the above intuition in a more precise and systematic way, we resort to a large number of experiments based on simulation. Two industrial case studies were used to guide the simulations and thereby obtain more realistic results. On one hand, the results of our empirical study confirm our intuition about what drives the effectiveness of STCS, though they provide insights that are more complex than what was originally expected. On the other hand, such analyses pointed out a particular characteristic of MBT (compared to numerical applications) that can make STCS less effective. The situation appears when there is a small clustered set of test cases that is far away from all the others (referred to as *outliers*), which is not uncommon, for example, in state machine-based testing when a small group of transition paths is mostly disconnected from the rest of the state machine. Our empirical analyses show that, in that case, the FDR of STCS can

significantly decrease. We hence propose an approach, based on rank scaling, to manipulate similarity values so as to alleviate this problem.

The rest of the paper is organized as follows. The next section provides background information about similarity-based test case selection. Section 3 discusses the problems related to outliers and outlines our solution to alleviate it. Section 4 describes the experiment design and reports the results. Section 5 provides a brief overview of related works covering similarity-based selection techniques. Finally, Section 6 concludes the paper and outlines our future work plan.

2 Similarity-based Test Case Selection

Unlike coverage-based selection, where the goal is maximizing the coverage of a test model (e.g., transition coverage in a state machine) by the selected test cases to maximize chances of fault detection, STCS techniques maximize diversity among the selected test cases. Diversity is calculated using a (dis)similarity measure between pairs of test cases. A similarity measure is a value that a similarity function assigns to the pair. Inputs of the function are usually an encoded test case as a set or sequence of elements. In the context of MBT, the inputs are abstract test cases defined on the test model rather than concrete test cases. We do not use the execution information of the test case as, in our context, the goal is to select them before execution. Abstract test cases are naturally generated as a first step by MBT and can hide the unnecessary information for similarity comparisons. For example, in state machine-based testing, an abstract test case representation can be a path in the state machine specifying the software under test (SUT). In general, different faults can be detected by the same test path instantiated with different test data (e.g., event parameter values). Therefore, to calculate the FDR of a technique, it is necessary to run the selected test paths with different input data and analyze its FDR distribution.

2.1 Encoding and similarity functions

The representation (encoding) of test cases has an important effect on a similarity measure. Though in MBT a test path represents an encoded version of a test case, the test path can be described at different levels of details. In [4], we studied three encodings for a test path in UML state machines: state-based, transition-based, and trigger-guard-based. We reported that trigger-guard-based encoding is the most effective one in terms of fault detection, where a test path (*tp*) is represented as:

<tp> ::= <TrGu> | <TrGu> “,” <tp>

< TrGu > ::= *trig* |*guard* | *id* | *guard* “+” *trig*

where *trig* is the identification of a trigger, and *guard* is the identification of a guard in the state machine. In this representation, a transition is identified by its trigger, a guard, or both. If there is a transition with no guard and trigger, we use the transition id (*id*) as its identifier.

Given an encoding, one may use different similarity functions to calculate the similarity value. In [3] we studied six set-based and sequence-based similarity functions. We proposed Jaccard Index [5] as the most cost-effective set-based and Needleman-Wunsch (NW) [6] as the best sequence-based similarity function. The Jaccard Index is defined as the size of the intersection divided by the size of the union of the two encoded test cases, whereas the NW algorithm assigns a similarity value based on the global alignment [6] of the two encoded test cases by arranging the sequences of elements in the test cases to identify regions of similarity between the sequences.

From an STCS point of view, the only required constraints on the similarity measures are that they must be positive and symmetric, which is true for all proposed set and sequence based measures. This means that properties such as reflexivity (*Sim*(tp_i , tp_j)=*maximum* iff $tp_i = tp_j$) and triangular inequality (*D*(tp_i , tp_j)+ *D*(tp_j , tp_k)>= *D*(tp_i , tp_k) where *D*(tp_i , tp_j) = 1/ *Sim*(tp_i , tp_j)) do not necessarily hold among different pairs of test cases [7]. For example, NW values can be in any range and, except for symmetry, does not feature any other well-known property of distance/similarity measures [8].

Given a set of *n* encoded test cases (s_n) and a similarity function (*Sim*), the test case selection problem is reformulated as minimizing *SimMsr*(s_n):

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

where *Sim*(tp_i , tp_j) returns the similarity of two test paths (encoded abstract test cases in MBT) in s_n represented by tp_i and tp_j. The last step in STCS is using a strategy to select a subset of test cases with minimum average pair-wise similarity (*SimMsr*). This test case selection problem is NP hard (traditional set cover) [9] and using an exhaustive search in

our cases (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. As an example from our case studies, the search space size for *n*=28 in a test suite of size 281 (~10% of the test suite) is $\binom{281}{28}$ ≅2.9*10^38.

In [2] we have examined GA, ART, and a hierarchal clustering algorithm as selection algorithms and found out that GA was the most cost-effective technique among them. In this paper, we show our analyses when both using GA as our proposed technique (called STCS_GA) and ART as the most well-known algorithm (called STCS_ART) for diversifying test cases. The algorithms are introduced in the following subsections.

2.2 Adaptive Random Testing

ART has been proposed as an extension to Random Testing [1]. Its main idea is that diversity among test cases should be rewarded, because failing test cases tend to be clustered in contiguous regions of the input domain. This has been shown to be true in empirical analyses regarding applications whose input data are of numerical type [1]. Therefore, ART is a candidate selection strategy in our context as well. In this paper, we use the basic ART algorithm described in [1], but we ensure that no replicated test case is given in output. The pseudo-code for ART is:

```
(1)   Z={}
(2)   Add a random test case to Z
(3)   Repeat until |Z|= sampleSize
(4)   Sample K random test cases that are different from Z
(5)         For each of these test cases k
(6)               k.maxSim = max(Sim(k , z ∈ Z))
(7)               Add the k with minimum maxSim to Z
```

2.3 Genetic Algorithms

In this paper, we use a steady state GA with the same settings as it has been used in [2], in which only the offspring that are not worse than their parents are added to the next generations. An individual in our context is a subset of size *n* from the original test suite (denoted s_n). Given a similarity function $Sim(tp_i , tp_j)$, the fitness function *f* to minimize is the sum, for all pairs (tp_i , tp_j) in s_n, of $Sim(tp_i , tp_j)$, denoted *SimMsr*. We use a single point

crossover with probability of P_{xover} to combine two different parents s_n^x and s_n^y. A mutated test path is replaced by a test path that is selected at random from the set of all possible test paths. A valid solution is a set of test cases in which there is no duplicate. The stopping criterion is 10,000 fitness evaluations, which is equal to the cost of 1,000 runs of ART with a candidate size 10 in terms of the resulting number of distance calculations. The pseudo-code of the employed GA is as follows:

(1) Sample a population G of *m* sets of test cases uniformly from the search space (i.e., the set of all possible valid sets with a given size *n*)
(2) Repeat until the stopping criterion is met
(3) Choose s_n^x and s_n^y from G
(4) $(\acute{s}_n^x, \acute{s}_n^y) := \text{crossover}\ (s_n^x, s_n^y, P_{xover})$
(5) Mutate $(\acute{s}_n^x, \acute{s}_n^y)$
(6) If valid $(\acute{s}_n^x, \acute{s}_n^y) \wedge \min(f(\acute{s}_n^x), f(\acute{s}_n^y)) \leq \min(f(s_n^x), f(s_n^y))$
(7) Then $s_n^x := \acute{s}_n^x$ and $s_n^y := \acute{s}_n^y$
(8) Else If valid $(\acute{s}_n^y) \wedge \min(f(s_n^x), f(\acute{s}_n^y)) \leq \min(f(s_n^x), f(s_n^y))$
(9) Then $s_n^y := \acute{s}_n^y$
(10) Else If valid $(\acute{s}_n^x) \wedge \min(f(\acute{s}_n^x), f(s_n^y)) \leq \min(f(s_n^x), f(s_n^y))$
(11) Then $s_n^x := \acute{s}_n^x$

3 Impact of Outliers and Rank Scaling Solution

Unlike test suites for numerical applications, where the population of all possible input test data is distributed with similar density in the input space, it is not uncommon in MBT that a subset of test cases be very dissimilar to the rest of the test suite (outliers). For example, if the test suite is derived from a state machine and (1) the state machine contains a partition which is initiated by a transition from the initial state, (2) this partition has no transition to/from the rest of the state machine and (3) the triggers of the transitions in the segment are very different than the triggers of the transitions in the main part of the state machine, then the test suite generated from such a model will contain a small set of test cases, which will be very dissimilar to the rest of the test suite, that covers that segment. Investigating the behavior of STCS in such a situation is necessary in order to gain confidence about STCS effectiveness in the context of MBT. But because we are

evaluating STCS on industrial case studies, and such artifacts are difficult to obtain in large numbers to support a systematic investigation, we perform simulations based on industrial case studies to increase realism.

We can show that both STCS_GA and STCS_ART will try to select half of the test cases from the outlier clusters (for simplicity, we will assume the presence of only one outlier cluster). The reason is that the similarity between any pair, in which one test case is from the main set of test cases and the other from the outlier set, would have a very low value compared to the other similarity values in the matrix. Therefore, to minimize $SimMsr(s_n)$, the selection algorithm is guided to select as many as possible of these pairs. Given a minimized test suite of size *n*, there will be *m* test cases from the main set, and *o* test cases from the outlier set, with the constraint *m+o=n*. The number of pairs in which the two test cases are from different sets would be *m*o*. Under the constraint *m+o=n*, the term *m*o* is maximized when *m=o*, from which it follows *o=n/2* (Schur-concave function [10]). Therefore, nearly half of test cases will be chosen from the outlier cluster regardless of the proportion of the cluster sizes. Consequently, if the outlier cluster is small and does not contain any fault revealing test case, the FDR of STCS will likely be low.

Since we expect poor results in the presence of no-fault revealing outlier, we suggest using a rank scaling technique to alleviate the problem. In this technique, the raw values in a similarity matrix are replaced by their rank. The rank is simply the index of the value after ordering all similarity values of the matrix in an array. This rank scaling approach is derived from solutions for solving outlier problem in statistics [11] and help decreases the large similarity differences between test cases from the outlier cluster and the rest of the test suite.

Notice that, in this paper, we are assuming *N* (test suite size) small enough such that a *N*N* matrix can be stored without significant overheads (this was the case for the two industrial case studies analyzed in this paper). When this is not possible, and we need to compute the similarity values on the fly each time we evaluate the similarity of a set of test cases, a dynamic rank scaling is needed. For example, a data structure (e.g., a hash-table) could be used to store all the unique similarity values encountered so far during the search (e.g., while using STCS_GA). Rank scaling would hence be based on those values.

4 Empirical Study

In this section we report the design and results of our empirical analysis. The high-level goal of this study is to investigate under which circumstances, characterized by the correlation between similarities of test cases and their fault detection, and the distribution of test cases in their definition space, a STCS is most effective in terms of fault detection rate (FDR).

4.1 Test suites description

In this study, we test different hypotheses regarding the effectiveness of STCS on different input test suites to minimize. Given a test suite of size *N*, we can consider a *N***N* matrix to represent the test suite in which all similarity pairs are stored (actually, only half of it is necessary, due to the symmetric property of the similarity functions).

These matrices are all based on the modification of test suites from two industrial case studies. However, we had to manipulate the matrices to create all the possible situations of a test suite with respect to the properties we want to investigate, as further explained below.

The SUT in case study A is a safety monitoring component in a safety-critical control system implemented in C++. A flattened version of the state machine representing the SUT consists of 70 states and 349 transitions. There are 15 real faults in the SUT which are detectable by a test suite automatically generated from a UML state machine representing the SUT's behaviour. The test suite, which is generated using our MBT tool (TRUST) [12], contains 281 abstract test cases (test paths) covering all round trip paths [13] in the state machine. Each test path either detects a certain fault or not regardless of its input data. In other word, the FDR values of the test paths of this case study are independent from input data.

The SUT in case study B is the core subsystem of a video-conference system which manages sending and receiving of multimedia streams implemented in C. As the previous case study, we deal with real faults detectable by an automatically generated test suite using TRUST. Case study B is smaller than A with 11 states, 70 transitions, 59 test cases (covering all round trip paths in the state machine) and only four detectable faults. But unlike case study A, the FDR of the test paths are not independent from input data. Depending on which data are chosen as input parameters for the events on the state machine, a fault may or may not be detected.

4.2 Research questions

The high level goal of this study leads to the following research questions:

RQ1. Under which conditions, with respect to the similarity of fault revealing test cases in a test suite, STCS performs best?

RQ1.1. Is STCS more effective if test cases which detect distinct faults are dissimilar?

RQ1.2. Is STCS more effective if test cases which detect common faults are similar?

These questions directly target the hypothesis on rewarding diversity, as discussed in Section 1, and seek to confirm it in the context of MBT. The diversity hypothesis is investigated with respect to two distinct properties through RQ1.1 and RQ1.2.

RQ2. Is rewarding diversity robust to small clusters of test case outliers (test cases which are very dissimilar to the rest of test suite)?

RQ3. What is the effect of rank scaling in the presence and absence of outliers?

RQ3.1. Does using rank scaled similarities improve STCS effectiveness in the presence of outliers?

RQ3.2. Does using rank scaled similarities impact negatively STCS effectiveness when there is no outlier?

The problem of outliers, discussed in Section 3, is being examined in RQ2. Our motivation, as mentioned above, is that in contrast to numerical applications, outliers are not a rare feature of test suites when they are generated using MBT. In question RQ3, the first sub-question RQ3.1 asks whether rank scaling is useful to alleviate the effect of outliers. RQ3.2 investigates whether rank scaling can reduce FDR when there is no outlier. If results show that rank scaling does not reduce the FDR in such situations and that rank scaling alleviates the outlier problem, then it would be recommended to always apply rank scaling in STCS.

4.3 General settings of the experiments

We designed two experiments Exp1 to answer RQ1 and Exp2 to answer RQ2 and RQ3. In both experiments we use STCS_GA and STCS_ART based on trigger-guard encoding and NW as similarity function when it must be specified. Note that the results of our study in [3], which was based on only case study A, showed the same level of effectiveness for both NW and Jaccard Index. We had recommended the Jaccard Index in that study since it is easier to apply than NW. However, in case study B, NW provides much better results. The most plausible explanation for this difference is that the sequence of test path elements

matters regarding fault detection in case study B and NW is a sequence-based function. Therefore, in this paper NW is used for both case studies.

Since we have built this study based on our previous work, the overall settings of the algorithms are the same as our previous study settings. For GA, the stopping criterion is 10,000 fitness evaluations, the crossover probability is 0.75 and the population size is 50. For ART, the candidate size is 10 and 1000 repetitions (from which we select the best) are performed for each run of the algorithm to ensure fair comparisons with GA. More details regarding the settings and rationale behind our choices can be found in [2].

Each experiment uses input matrices which are generated by modifying similarity matrices of the case studies A and B, which we refer to as simulations. We repeat the experiments on different sample sizes (four for case study A and six for case study B) to check whether the results are consistent across the size range. Although the actual sizes for the sample sets are different for the two case studies, the percentage of selected test cases among all test suites is almost the same: sample sizes for experiments driven from case study A are equal to 5, 15, 25, and 35, and sample sizes for the experiments driven from case study B range from 3 to 8. The important point here is running the experiments on relatively small sample sizes, since this is the most interesting case in practice, when it is not possible to run many test cases on the actual hardware and platform (as for the industrial systems used as a case study in this paper). Furthermore, for larger sizes all techniques converges to 100% FDR and differences will not be significant.

For both algorithms and all sample sizes, each experiment is repeated 1000 times (100 runs for search technique with different random seeds and 10 different input matrices per each matrix type to account for random variation in both search techniques and matrix generation). A rigorous statistical procedure has been used to evaluate and compare the effectiveness of these randomized algorithms [14].

4.4 Design and results of Exp1

To answer RQ1, we designed Exp1 where STCS_GA and STCS_ART, are applied on nine different types of input similarity matrices. These similarity matrices are artificially built—though based on case study A—to simulate all possible combinations of two properties of a test suite with respect to its test cases' similarities. Property X denotes the similarity between test cases that detect a common fault and Property Y denotes the similarity between test cases that detect distinct faults. In other words, RQ1.1 and RQ1.2 address the effect of Property Y and X on STCS effectiveness. In our simulations, each of these two

properties can have three values: *High* (top 10%: [0.9,1.0]), *Low* (bottom 10%: [0.0,0.1]), and *Random* (randomly picked from the valid range: [0.0,1.0]), which makes nine possible combinations of the properties as an identifier for a test suite. For example, a test suite where test cases that detect a common fault are highly similar and those that detect different faults are very dissimilar, is identified by Property X=*High* and Property Y=*Low*. Note that, since the similarity functions we use only need to be positive and symmetric, when we generate matrices for our experiments, we do not need to validate each similarity value by checking its relationship with other values for other test case pairs in the same matrix.

To generate matrices with different property combinations while remaining as realistic as possible, we kept the original number of faults (15) and same failure rate as in case study A (74/281) and built matrices with sizes 300, 600, 6,000, and 12,000 (nine matrices for each matrix size). Recall that the reason for using different sizes is to test the independence of the results from test suite size and therefore help the generalization of the results to larger case studies (i.e., does the technique scale?). Though this is only realistic when the system under test has already undergone significant verification, to make the analysis tractable, we assumed that each test case can find at most one fault. At this stage it is difficult to assess the consequences of this assumption and it therefore constitutes a threat to validity.

For each matrix type, 10 instances are generated. Both STCS_GA and STCS_ART are applied on these matrices 100 times, which yields a total of 1000 runs. In total, given that there are four sample sizes, nine matrix types, 1000 runs, and two selection techniques, then 288,000 (4*9*10*2*100*4) observations are collected in Exp1, each with an FDR value for the selected test cases. The FDR is the average number of faults detected by the selected test cases, for each run of the STCS, divided by the total number of faults (15).
Figure 1 and Figure 2 shows partial results for Exp1. Due to space constraints, we chose to present only the FDR results for two sample set sizes (15 and 25) and two matrix sizes (300 and 12000) for each of the STCSs, but the same trend was observed over all sample sets and matrices as illustrated in Figure 3 for effect sizes. The first observation is that, regardless of the type of SCTS, sample size, and matrix size, test suites with a *Low* value for Property Y show higher FDR. This means that the most important factor for ensuring the success of STCS is having test cases detecting distinct faults as far (dissimilar) as possible from each other. This confirms our hypothesis and answers RQ1.1.

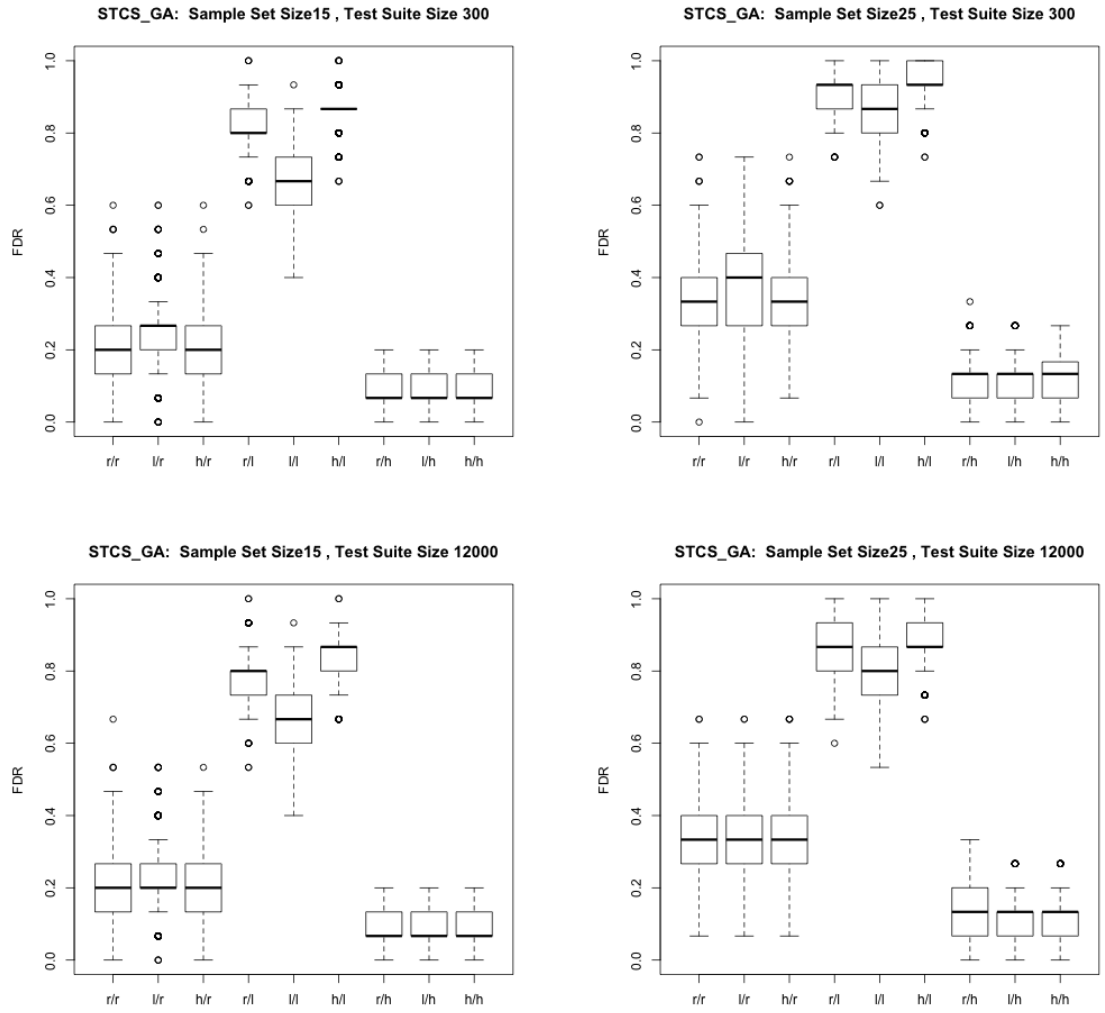


Figure 1 FDR of a sample set (of size 15 and 25) of test cases selected by STCS_GA from different matrix types of size 300 and 12000. Matrix types on X_Axis are identified as Property X/Property Y where each property can be random (r), low (l) or high (h). Each boxplot shows 1000 observations (100 STCS runs per matrix on 10 different matrix instances)

To answer RQ1.2, if we first look at cases where Property Y has a *Low* value, we can see significant differences in test suites with *High* values for Property X when compared to the others. This means that the combination of *High/Low* values for property X/Y is the best combination for STCS. This directly confirms the hypothesis discussed in RQ1. However, Property Y seems to have stronger effect since its value completely overrides the effect of Property X.

To gain more confidence in the conclusions drawn from this empirical study, we also carried out a series of statistical tests. For each of the 16 combinations of matrix sizes and test sample sizes, we used a Mann-Whitney U-test to compare the performance of the property combination *High/Low* against the other eight combinations. This test verifies

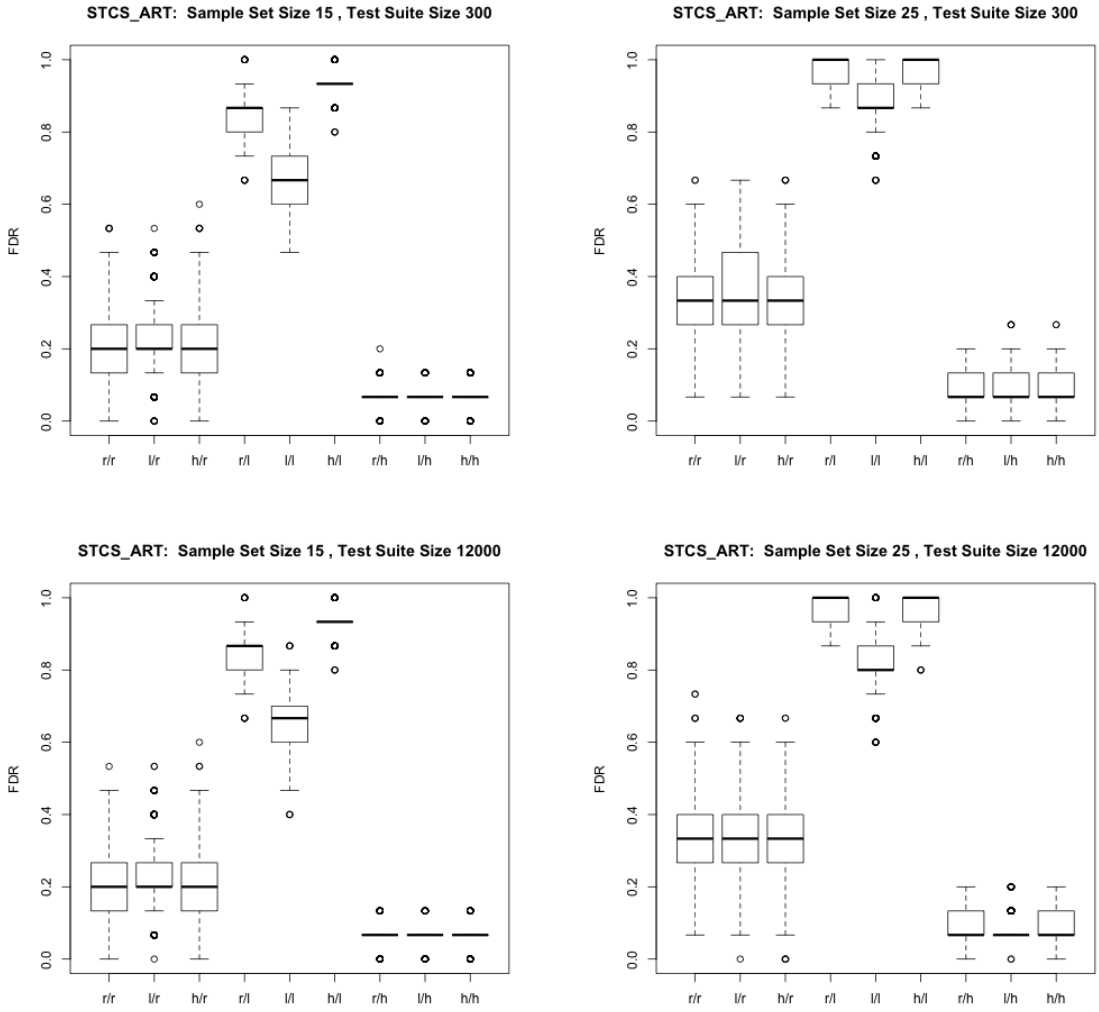


Figure 2 FDR of a sample set (of size 15 and 25) of test cases selected by STCS_ART from different matrix types of size 300 and 12000. Matrix types on X_Axis are identified as Property X/Property Y where each property can be random (r), low (l) or high (h). Each boxplot shows 1000 observations (100 STCS runs per matrix on 10 different matrix instances)

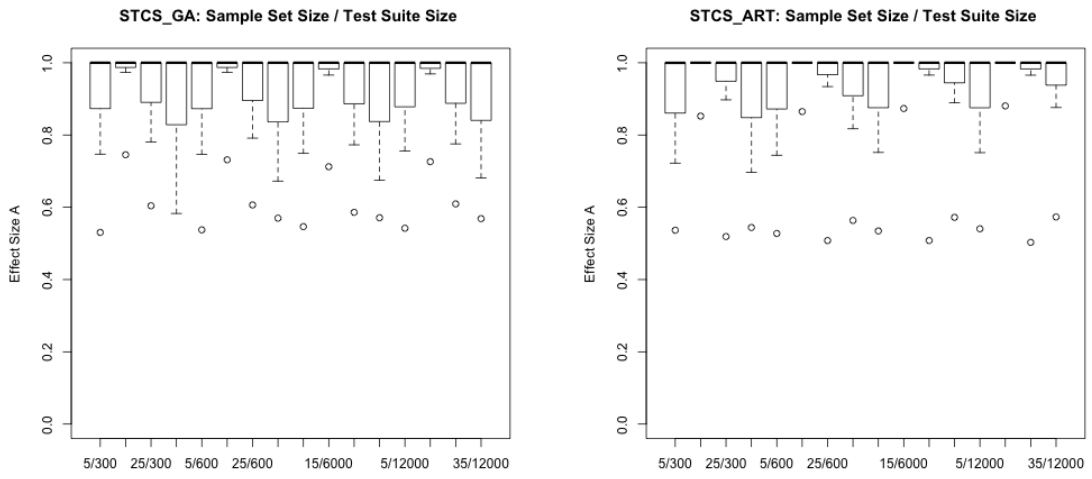
whether two FDR distributions are statistically different. For STCS_GA, the p-values were always lower than our selected level of significance (0.05). For STCS_ART, resulting p-values were lower than 0.05 in all cases but four out of the 16*8=128 comparisons. This provides strong statistical evidence to support the claim that *High/Low* is the best condition under which to use STCS.

To quantify the magnitude of improvement in a standardized way, in Figure 3 we plot the effect size measure of STCS_GA and STCS_ART for different sample and matrix sizes using the Vargha-Delaney's *A* statistic. This statistic estimates the probability that a data point randomly taken from a set (i.e., a probability distribution) will have higher value than another point randomly taken from a second data set. When the two distributions are the same, we would have *A=0.5.* The results in Figure 3 show that, most of the time, the *A*

values are close to 1. This means that, for the *High/Low* combination, it is nearly certain that STCS will yield better results than in the other eight cases, even when we take into account the variance of the results due to the randomized nature of these algorithms.

4.5 Design and results of Exp2

For Exp2, we apply STCS_GA and STCS_ART on four types of matrices per case study. We manipulated the original matrices from each case study to examine the effect of outliers on the FDR of the test suites. We did so by adding extra percentages of outlier test cases. For case study A and B, respectively, we built matrices with 1, 2, 5, 10, and 20 and 5, 10, and 20 percent extra test cases (1 and 2 percent would not make sense for the smaller case study B with only 59 test cases). Four types of matrices are generated for each case study and size: (1) *Random/Base*: The original matrix from the case study plus extra test cases with random similarity values in the same range of similarity values as in the original matrix. This matrix is the baseline for the FDR comparisons; (2) *Cluster/Base*: The original matrix plus extra test cases with random similarity to each other but very low similarity (outliers) to the rest of the test suite (original test cases). This low similarity value must be set to be much lower than the minimum values within each of the groups containing the original and additional test cases. If *min* and *max* are the minimum and maximum values in the original matrix, we first change the matrix by replacing every value *x* with *x*+10*(*max*-



(a) (b)

Figure 3 Effect size measure A (each calculated out of 1000 observations) for FDR of a sample set selected by STCS_GA (a) and STCS_ART (b) shown as boxplots for the eight comparisons. The effect size compares the High/Low matrix type with the all other eight matrix types of Figure 1 and Figure 2. X_Axis shows the sample size/matrix size

min) to ensure much higher NW similarity values among the original test cases compared to such values with outliers. The NW values between outliers are then generated to be in the same range as the original matrix. Last, to simulate a low similarity between the outliers and the original test cases, we set the NW value between them to zero. The constructed matrix therefore represents the situation where outlier test paths are present in the test suite; (3) *Random/Ranking*: The same matrix as *Random/Base* but after applying rank scaling as introduced in the research question subsection; (4) *Cluster/Ranking*: The same matrix as *Cluster/Base* but after applying rank scaling.

To answer RQ2, we compare the FDR of a selected subset of test cases (for four different sizes) from a test suite represented by the *Cluster/Base* matrix with the FDR of a same size subset using the *Random/Base* matrix. This comparison investigates the effect of outliers on the STCS effectiveness.

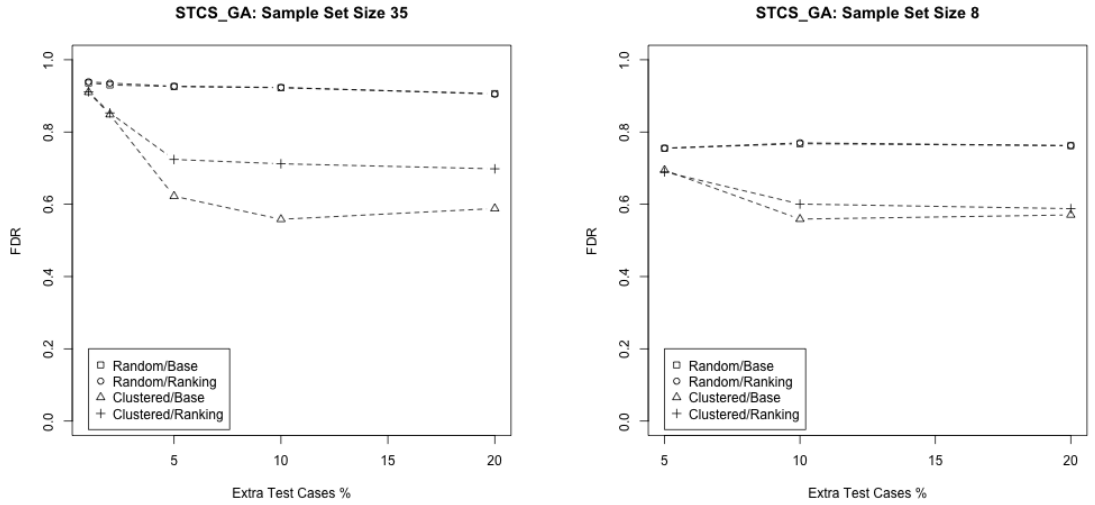
To investigate RQ3, we compare STCS effectiveness on the matrices from *Cluster/Ranking* and *Cluster/base*, we will assess whether rank scaling has significantly alleviated the effect of the outliers (RQ3.1). We also compare the effectiveness of STCS on the *Random/Ranking* and *Random/Base* matrices to check for possible negative effects of rank scaling when there is no outlier (RQ3.2).

We generate 10 instances of each of the 32 matrices (four matrix types and eight outlier percentages in the two case studies) to account for random variation in matrix generation. Both STCS_GA and STCS_ART are applied on these matrices 100 times to account for random variation in search techniques. In total, given that there are four sample sizes in case study A and six sample sizes in case study B, 320 matrices, 100 runs, and two selection techniques, then 640,000 (10*320*100*2) observations are collected for Exp2. Each observation has an FDR value for the selected test cases. The FDR calculation for case study A is the same as for Exp1 but is different for case study B, since, in the latter case, whether each test path detects a fault depends on which input data is used. For case study B, we randomly (with equal probability for each input data value) generated 10 different test cases per test path. Therefore, probability P_f of finding a specific fault f with the selected subset of test paths is equal to one minus the probability of not finding the fault by any of the test paths in the chosen set: $P_f = (1 - \prod_{i=1}^{n}(1 - p_i))$ where n is the size of the subset and p_i is the estimated probability of detecting fault f with test path i in the subset: number of times the fault is detected by the 10 test cases generated for that test path divided by 10. The FDR is hence computed by averaging these probabilities $\sum P_f / |F|$,

where $|F|$ is the number of faults. From the results of Exp2, answering RQ2 and RQ3, Figure 4 and Figure 5 are chosen to show one representative example since the trend is again the same over different sample sizes and algorithms for case study A. In Figure 4.a, the clear gap between *Cluster/Base* and *Random/Base* shows a strong drop in STCS effectiveness in the presence of outliers (RQ2). Comparing *Cluster/Base* and *Cluster/Ranking* we can clearly see that rank scaling helps STCS improve its effectiveness in the presence of outliers (RQ3.1) and comparing *Random/Base* and *Random/Ranking* clearly shows there is no reduction in FDR when there is no outlier in the test suite (RQ3.1).

In case study B, outliers also decrease effectiveness of STCS, though to a lesser extent (RQ2), and rank scaling once again does not compromise the potential FDR for test suites without outliers (RQ3.2). However, as it can be seen in the Figure 4.b, the improvement for case study B when comparing *Cluster/Base* and *Cluster/Ranking* is relatively small (RQ3.1), perhaps in part because the impact of outliers is already smaller to start with in this case study.

As in the previous experiment, to get more reliable results, we also carried out a rigorous statistical procedure using Mann-Whitney U-tests and Vargha-Delaney's *A* statistics (effect size) to compare FDR distributions across the four types of matrices. Comparing the performance of *Random/Base* with *Random/Ranking* (RQ3.2) yields p-



(a) (b)

Figure 4 FDR of a sample set selected by STCS_GA from test suites based on case study A with size 35 (a) and case study B with size 8 (b). Four combinations are compared: with (Clustered) or without outliers (Random), and using rank scaling (Ranking) or not (Base). The graphs show the average FDR over 1000 STCS_GA runs. X_Axis shows the percentage of outliers

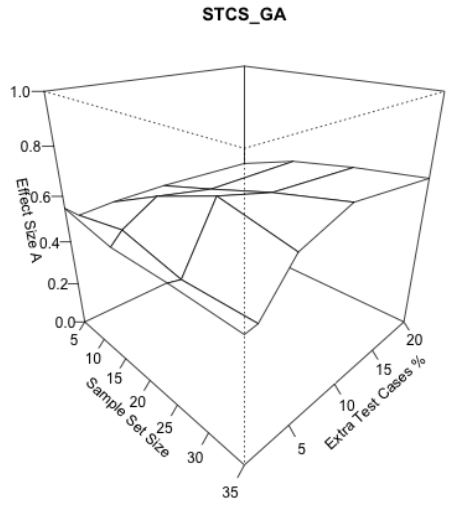


Figure 5 The effect size measure A for FDR of sample sets selected by STCS_GA from the test suite driven from case study A. X and Y axes show the outliers percentage and the sample set size

values lower than 0.05 in only one case out of 20 comparisons (five extra test case percentages time four matrices) for STCS_GA and two out of 20 comparisons for STCS_ART, where, even in those cases, the FDR difference between *Random/Base* and *Random/Ranking* is practically negligible. This statistically confirms that rank scaling is not particularly harmful in most cases when no outlier is present. However, when we compare *Cluster/Base* against *Cluster/Ranking* (RQ3.1), we obtain 11 cases with significant p-values for STCS_GA, and six cases for STCS_ART.

In Figure 5, we plot the effect size measure of STCS_GA for different sample and matrix sizes when we compare *Cluster/Base* against *Cluster/Ranking* (RQ3.1) in case study A. For small sample sizes and small outlier cluster, the effect is minimal (i.e., very close to 0.5). However, for larger sizes, the effect gets much stronger (close to 0.7).

As explained before, the main reason for which we apply rank scaling is to balance the distribution of the selected test cases from each cluster of outliers (if present). To examine this phenomenon, we considered one scenario (case study A with 20% extra test cases

Table 1 Average number of test cases selected by STCS_GA from the outlier cluster (20% extra test cases on the case study A) with and without rank scaling

Sample Size	Best	No Ranking	Ranking
5	1	2.95	2.94
15	3	7.06	6.77
25	4	12.01	10.61
35	6	17.00	14.41

forming a cluster of 56 test case outliers) and applied STCS_GA for selecting test case subsets (four sample sizes). Table 1 shows the average number of test cases taken from the outlier cluster with and without rank scaling. The best column shows the optimal number of test cases if we would select by only considering the size of the test suite and its outlier cluster, as expressed by the formula below.

$$Best = 1 + [\frac{clusterSize * (sampleSize - 1)}{testSuiteSize}$$

Based on the results in Table 1—Note the relation between Table 1 and Figure 4: the last row of Table 1 corresponds to Figure 4.a, with 20% extra test cases — it is clear that without rank scaling roughly half of the sample set is taken from the outlier cluster. The data suggest that rank scaling partially alleviates the problem. We get better improvement for larger sample sizes and the reason why this is the case will require further investigation.

One possible alternative to rank scaling for solving the outlier problem could be an approach that can be summarized as (1) finding the outlier cluster(s) using a clustering technique and identifying an outlier cluster based on the ratio of the inter-cluster distances to the intra-cluster distances (2) assigning a sample size to the outlier cluster based on the proportion of its size to the entire test suite size (3) and finally applying the STCS separately on the outlier and the main test cases. In previous work [2], we found that clustering techniques were less effective than STCS. Furthermore, rank scaling is easier and computationally cheaper than clustering techniques. However, hybrid combinations would be promising areas for further research.

4.6 Discussion on threats to validity of the results

This study was conducted according to recently proposed guidelines for conducting empirical studies in search-based testing [15] and using statistical tests to assess randomized algorithms in software engineering [14]. Regarding *construct validity* of the experiments, the most important factor is the validity of the measures used for assessing FDR and similarity comparisons. These measures are taken from previously published studies [2-4] and their validity are already discussed there. Another remaining concern is the artificially generated similarity values in the experiments. As discussed in the background section, we are using the NW similarity measure, which entails no constraint on the different pairs of similarities. Therefore, the assignment of *High*, *Low*, and *Random*

values cannot lead to incorrect matrices. However, the assumption in Exp1 that each test case can find at most one fault constitutes a threat to validity of the results. A more general experiment where each test case can find each fault with a certain probability should be conducted to achieve more reliable results.

The randomized nature of the employed algorithms poses a threat to *internal validity*. To account for it, the experiments were run many times with different random seeds, thus leading 1000 observations for each case study/sample size/search technique/matrix type combination (100 runs of search technique on 10 randomly generated input matrices). In addition, a rigorous statistical procedure (comprising significance tests and effect size measures) has been used to strengthen the *conclusion validity* of the results.

To cope with *external validity*, we conducted experiments using many different combinations of sample sizes, test suite sizes, case studies, and STCS techniques. In particular, the use of two industrial systems to drive the simulations (by retaining some of their characteristics such as failure rate of test cases and number of faults) provides stronger support to the applicability of our approach to other industrial systems. But, as for all empirical studies, our results might not generalize to other case studies and only replications will help build confidence.

5 Related Work

STCS for MBT was first introduced in [16], where sequences of transitions in a Labeled Transition System model of the SUT are used for representing test paths. The similarity function is simply counting the common transitions in two test paths and a Greedy Search is used for minimizing the sum of pair similarities. Later, Hemmati *et al.* [4] introduced and improved STCS for UML based testing by using a trigger-guard based encoding of test paths, by using better similarity measures [3] and by resorting to more powerful search techniques [2].

Except for these works on model-based STCS, diversifying test cases has been studied on code-based test case selection, minimization and prioritization, mostly in the context of regression testing. The basis for computing test case similarity in these studies is usually on code coverage or on some other execution information. For example, in [17] and in [18], all def-use pairs coverage and a sequence of memory operations are used to calculate the similarities, respectively.

To the best of authors' knowledge, no existing study systematically investigates the impact of test suite properties on STCS in the context of MBT. Similar studies published to date have been conducted in the numerical application domain to examine the effect of test suite properties, with respect to test case similarities and their fault detection, on the ART algorithm. Several papers have been published on this subject [1], in which for example optimal conditions for ART have been theoretically studied [19]. However, as discussed in Section 3, MBT is very different from the unit-testing of numerical applications in terms of the distribution of input test data in the input space (e.g., clusters of outliers are unlikely in the numerical application domain).

6 Conclusion and Future Work

In previous studies we proposed similarity-based test case selection (STCS) techniques to reduce the cost of model-based testing (MBT) [2-4]. Though the technique was successfully applied on one industrial system, we needed more empirical evidence to support the idea that maximizing the diversity of test cases was a good principle for test case selection and understand under which conditions.

In this paper, we conducted a large scale simulation, based on two industrial case studies, to investigate, in a controlled manner, how relevant properties of a test suite affect the effectiveness of STCS. When considering properties are about the relationship between fault detection and similarity distributions among test cases, our results showed that the most ideal situation for a STCS is when, in a test suite, (1) test cases that detect a common fault are similar and (2) test cases which detect distinct faults are dissimilar. Our empirical study shows that property (2) is much more important than property (1). This result will help us devise improved similarity functions in the future, which in turn will result into more effective STCS.

In this paper, we also investigated the problem of outliers in a test suite—which are not unlikely to happen in MBT—that could compromise the performance of STCS. Results confirmed the significant impact of outliers and an approach, based on using rank scaling measurement instead of raw similarity values, was proposed to address the outlier problem. Though rank scaling had a positive effect, it only partially addressed the outlier problem and additional strategies remain to investigate.

Future work will examine other solutions for the outlier problem based on combining clustering and STCS techniques. We will also use the insights that we gained from this study to develop techniques to improve STCS.

Acknowledgement

The authors wish to thank Marius Liaaen, from Tandberg AS, now part of Cisco, for helping us in conducting experiments.

References

[1] T. Y. Chen, F.-C. Kuoa, R. G. Merkela, and T. H. Tseb, "Adaptive Random Testing: The ART of test case diversity," *Journal of Systems and Software*, vol. 83, pp. 60-66, 2010.

[2] H. Hemmati, A. Arcuri, and L. Briand, "Reducing the Cost of Model-Based Testing through Test Case Diversity," in *22nd IFIP International conference on Testing Software and Systems (ICTSS), formerly TestCom/FATES*, pp. 63-78, 2010.

[3] H. Hemmati and L. Briand, "An Industrial Investigation of Similarity Measures for Model-Based Test Case Selection," in *21st IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pp. 141-150, 2010.

[4] 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)*, pp. 267-276, 2010.

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

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

[7] R. Xu and D. C. Wunsch II, "Survey of Clustering Algorithms," *IEEE Transactions on Neural Netwoks*, vol. 16, pp. 645-678, 2005.

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

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

[10] P. J. Boland, H. Singh, and B. Cukic, "Comparing partition and random testing via majorization and Schur functions," *IEEE Transactions on Software Engineering*, vol. 29, pp. 88-94, 2003.

[11] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*, 3 ed.: Chapman & Hall, 2003.

[12] 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.

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

[14] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," Accepted for publication in *the*

proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE), 2011.

[15] 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)*, vol 36, pp 742-762, 2010.

[16] 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.

[17] A. d. S. Simão, R. F. d. Mello, and L. J. Senger, "A Technique to Reduce the Test Case Suites for Regression Testing Based on a Self-Organizing Neural Network Architecture," in *30th Annual International Computer Software and Applications Conference (COMPSAC)*, 2006.

[18] 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.

[19] T. Y. Chen and R. Merkel, "An upper bound on software testing effectiveness," *ACM Transactions on Software Engineering and Methodology*, vol. 17, 2008.

Achieving Scalable Model-Based Testing Through Test Case Diversity

Hadi Hemmati, Andrea Arcuri, and Lionel Briand

Submitted to ACM Transactions on Software Engineering and Methodology (TOSEM), 2010

Abstract— The increase in size and complexity of modern software systems entails scalable, systematic, and automated testing approaches. Model-based testing (MBT), as a systematic and automated test case generation technique, is being successfully applied to verify industrial-scale systems and is supported by commercial tools. However, scalability is still an open issue for large systems as in practice there are limits to the amount of testing that can be performed in industrial contexts. Even with standard coverage criteria, the resulting test suites generated by MBT techniques can be very large and expensive to execute, especially for system level testing on real deployment platforms and network facilities. Therefore, a scalable MBT technique should be flexible regarding the size of the generated test suites and should be easily accommodated to fit resource and time constraints. Our approach is to select a subset of the generated test suite in such a way that it can be realistically executed and analyzed within the time and resource constraints, while preserving the fault revealing power of the original test suite to a maximum extent. In this paper, to address this problem, we introduce a family of similarity-based test case selection techniques. We evaluate 320 different similarity-based selection techniques and then compare the effectiveness of the best similarity-based selection technique with other common selection techniques in the literature. The results based on two industrial case studies show significant benefits and a large improvement in performance when using a similarity-based approach. We complement these analyses with further studies on the scalability of the technique and the effects of failure rate on its effectiveness. We also propose a method to identify optimal tradeoffs between the number of test cases to run and fault detection.

1 Introduction

Model- Model-based testing (MBT) [1] has been used for many years with the intent of generating executable test cases by systematically analyzing specification models (e.g., represented as UML state machines) following a test strategy such as a coverage criterion, that aims to cover certain features of the model (e.g., all transitions). One of the main obstacles to the transfer of MBT technology into industrial practice is scalability [2-5]. Scalability is an issue spanning all steps of the MBT procedure [1, 6], from handling large system models to generating and executing large test suites [2]. In this paper, we focus on an important but neglected scalability aspect of MBT: Given a software under test (SUT), how to optimize MBT fault revealing power within resource and time constraints?

In practice, system testing must be at least partially performed on the actual hardware platform (e.g., with actual sensors and actuators) or on a network specifically configured to help controlled and systematic testing (e.g., emulating IP traffic). This can have a large effect on the overall cost of testing since (a) test case execution time may be much higher than what can be expected, for example, at the unit test level, and (b) test case execution may require dedicated physical resources (e.g., specific assigned machines and restricted-access network) of limited availability. In an example from one of our industrial case studies, which will be introduced later in Section 5.1.1, each test case execution requires several communicating machines (video conferencing systems) dedicated to the test execution and takes a couple of minutes to complete. Therefore, in this context lowering the cost of test suite execution, both in terms of time and resource usage, is crucial for the scalability and therefore applicability of MBT. Our experience in applying MBT on two industrial case studies, with different sizes and from different application domains, suggests that the cost of executing test suites generated by MBT (given standard coverage criteria) can entail the use of far higher resources and time than what the budget and deadlines permit.

To address this problem, we introduced in [3] a flexible technique to allow the tester to adjust the size of the test suites according to the project's budget and deadlines while maximizing the test suite fault revealing power. The technique, that we call *similarity-based test case selection* (STCS), is based on selecting the most diverse subset of test cases among those which are generated by applying a coverage criterion on a test model (denoted the *original test suite*). In other words, the choice of test cases to execute is optimized with respect to their pair-wise similarity, based on the underlying assumption

that there is a positive correlation between the diversity of test cases and their fault detection [2, 7].

In this paper, we introduce 320 different STCS techniques (STCS variants), which result from different combinations of decisions regarding the three components (parameters) characterizing any such technique: the encoding (representation) of abstract test cases (ATCs, that are the platform-independent representations of test cases), the similarity function, and the algorithm employed to minimize similarities. We apply all these alternative STCS techniques on two industrial case studies, spanning different application domains. First, based on analyzing the fault detection rate (FDR) and selection cost of the techniques, we found that the choices made for any of the abovementioned three parameters has a significant impact. Second, we obtain the best results with an STCS that encodes ATCs using a state-trigger-guard-based encoding, generates similarity matrices using a Gower-Legendre similarity function [8], and applies an (1+1) Evolutionary Algorithm [9] to minimize average pair-wise similarities in the selected ATCs. Third, to assess the effectiveness of STCS when compared to simpler and common options for selection in the testing literature, we further compare our best STCS variant to random selection and coverage-based selection techniques, where one maximizes model coverage in the selected ATCs. The results of such comparisons show a staggering reduction in cost (50% to 80%) and improvement in FDR (e.g., up to 45% over coverage-based and 110% over random selection) when using the best STCS variant.

To obtain more general results, we also study the effect of varying the failure rate (θ) on the effectiveness of STCS by manipulating one of the original test suites from the case studies and generating different input test suites with various θs. The results show that the best STCS is never worse than non-STCS techniques regardless of the θ value. We also analyze the relationship between test case similarities and FDRs and devise a heuristic to estimate when increasing test suite size is unlikely to increase FDR. This heuristic enables practitioners to select a tradeoff between test suite size and FDR by analyzing the variation in average similarity among selected test cases. In summary, the main contributions of this paper are:

- We analyze the impact of the three STCS parameters on STCS effectiveness
- We identify the best STCS among 320 possible variants resulting from the setting of three parameters

- We compare the best STCS with other, more common test selection alternatives
- We analyze the impact of the test suite failure rate on STCS effectiveness
- We study the scalability of STCS
- We propose a heuristics which helps select a tradeoff between test suite size and fault detection

The rest of the paper is organized as follows. The next section motivates the study by explaining the importance of test suite scalability in MBT. Section 3 provides background information about model-based test case selection. Section 4 introduces our approach for test case selection (STCS). Section 5 describes the experiments' design and reports the results. Section 6 provides an overview of related works covering similarity-based selection techniques. Finally, Section 7 concludes the paper and outlines our future work plan.

2 Test Suite Scalability in Model-based Testing

The cost of test suite execution is an important factor for applicability of any test case generation technique. The number of generated test cases, which are going to be executed, has a direct relation with this cost. However, test suites generated by MBT approaches tend to be very large and they get exponentially larger with increasing model size (larger SUTs). Further, the problem gets even worse when the testing is semi-automated (e.g., automatically generating oracles may be very difficult or impossible, such as in a subjective quality assessment of a video stream) and human-effort is necessary in the execution and analysis of the test cases.

Using different coverage criteria seems to be a solution for this problem, since one may apply a less demanding criterion to end up with a smaller (less costly) test suite. However, our previous investigation [3] showed that such an approach does not solve the problem because (a) A coverage criterion is a mean for systematically targeting specific types of faults, e.g. in UML state machine-based testing (USBT), if one changes the coverage criterion from all transitions to all states to reduce the size of test suite, the new test cases may not detect the same type of faults anymore; (b) Even if one is flexible regarding the targeted type of faults, there is a limited number of standard coverage criteria that are applicable on a given model. Therefore, often this is not a practical solution as one cannot

ensure that the number of test cases will be below a required threshold corresponding to the testing budget.

The above discussion suggests there is a need for a more flexible approach to solve the problem of test suite scalability in MBT. Such an approach should be based on applying a reasonable coverage criterion (based on the domain and project information) and then eliminating some of the generated ATCs to only produce a concrete test suite of manageable size, which can be completely executed and analyzed within the project deadlines and resource constraints. This elimination step is usually based on one criterion (e.g., maximizing a code coverage measure such as statement coverage) that is assumed to have a correlation with the FDR of the test suite. Applying the criterion on the original test suite can be done in three ways: test case selection, test suite minimization, and test case prioritization. A selection technique, given a maximum number of test cases, selects a subset of the original test suite that optimizes the chosen criterion. The goal of a test suite minimization is to minimize the test suite by removing redundant test cases with respect to the criterion. Note that the main difference between selection and minimization is that a selection technique requires the output test suite size as an input parameter, but minimization techniques may generate test suites of any size. Therefore, in our context we favor a selection technique that ensures a maximum number of test cases. However, it is always useful to be able to minimize the test suite (while preserving its original FDR) in cases where there is no restriction or no clear criteria to select the test suite size. It is also possible to order the execution of all test cases in the test suite using a prioritization technique, but this is not required to solve our problem. Therefore, in this paper, we focus on test case selection and extend the idea to minimization when we try to estimate the optimal size of the test suite.

3 Model-based Test Case Selection

Test case selection/minimization is mostly studied in the context of regression testing, where the goal is to find a subset of the original test suite that guarantees the execution of fault-revealing test cases [10-13]. The main differences between model-based test case selection and selection in the context of regression testing are that, in our context: (a) we are not interested in identifying the changed parts of the system and (b) we do not have test execution information, as it is the case in regression testing, because selected test cases will be executed for the first time. Therefore, heuristics such as using component metadata [14],

and execution traces (e.g., call stack [15]) 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 [16] and additional coverage [10]). Rather, MBT selection heuristics are based only on the characteristics of the (abstract) test cases.

There can be different classes of applicable selection techniques in MBT. The simplest technique is Random Testing (RT) [17], where there is no guidance to select test cases. Maximizing coverage has been a common practice over the years in selection and prioritization [10, 18]. In MBT, coverage is defined at the model level, which can be extracted from ATCs without execution. For example, transition coverage in a state machine [19] can be determined if traceability has been preserved between an ATC 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 presented in [10] uses a Greedy search to select, at every step, the test case that covers the most uncovered statements (additional coverage technique). Similarly, in [20] a Genetic Algorithm is used to achieve maximum coverage in the selected subset of test cases. STCS is a newly introduced [3, 21, 22] category of selection techniques which can be applied in both code and model-based testing.

An STCS technique selects the most diverse test cases with respect to a similarity measure, which requires assigning a similarity value to each pair of test cases and minimizing the average pair-wise similarities between the selected test cases. In the next section, we will explain STCS steps in details. The underling idea behind STCS techniques is borrowed from rewarding diversity among input data [23]. The same idea is applied by STCS to diversify the selected test cases assuming that “the more diverse the test cases, the higher their fault detection rate”. To investigate, in a controlled manner, the relationship between fault detection and similarity distributions among test cases, we have conducted a large scale simulation [7], based on two industrial case studies. Our results showed that the most ideal situation for an STCS is when, in a test suite, (1) test cases which detect distinct faults are dissimilar and (2) test cases that detect a common fault are similar. We have also studied these hypotheses on one industrial case study [2] and found that test cases finding a common fault were indeed clustered together in the test case space (defined by the similarity measure) and that these clusters were mostly distinct.

4 Similarity-based Test Case Selection

In this section, we explain the procedure of STCS and introduce all alternative techniques that we have used in this study in each step of the STCS. As we mentioned earlier, the basis of STCS is minimizing the average pair-wise similarity between the selected test cases. This requires identifying a similarity measure for pairs of ATCs and an optimization algorithm to minimize the output set of ATCs with respect to that measure. Therefore, an STCS is composed of three phases: (1) encoding of ATCs, (2) similarity matrix generation, and (3) minimizing similarities.

4.1 Encoding of abstract test cases

Before identifying any similarity measure, the inputs to the similarity function should be represented at a proper level of abstraction, containing relevant information and no unnecessary details. In the context of MBT, the inputs are ATCs instead of concrete test cases since we do not need platform dependent information and ATCs are naturally generated as a first step by MBT. As a result, we reduce the cost of test case generation by only generating executable test cases for the selected ATCs and also by hiding the unnecessary information for similarity comparisons. Encoding of the ATCs has an important effect on the effectiveness of the STCS. Though in USBT a test path represents an encoded ATC, the test path can be described at different levels of details. We consider four possible encodings for a test path in UML state machine: state-based (SB), transition-based (TB), trigger-guard-based (TGB), and state-trigger-guard-based (STGB):

- SB: <tp> ::= *state* | *state* “,” <tp>
- TB: <tp> ::= *tran* | *tran* “,” <tp>
- TGB: <tp> ::= <TG> | <TG> “,” <tp>
  < TG > ::= *trig* |*guard* | *id* | *guard* “+” *trig*
- STGB: <tp>::= *state* | *state* “,” <TG> “,” <tp>

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 TGB 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. SB encoding focuses on state level faults whereas TB and TGB can better extract relevant

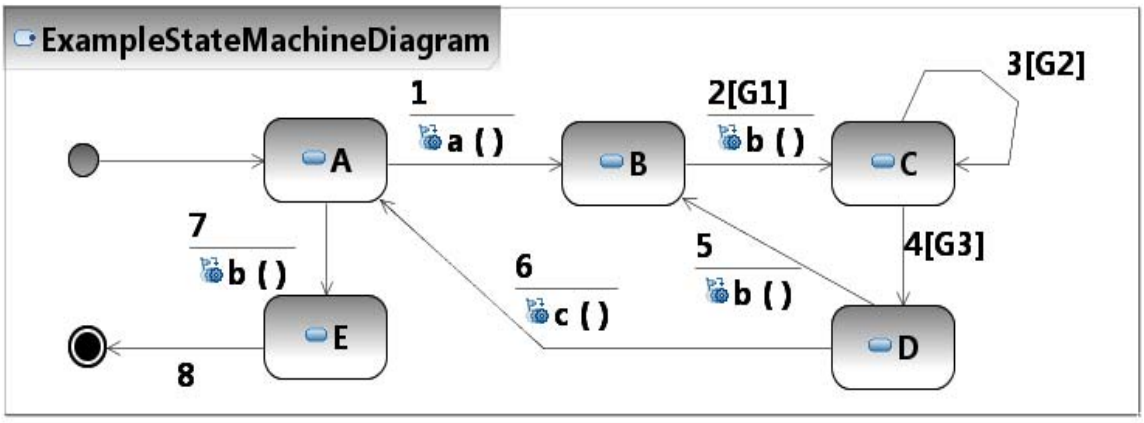


Figure 1 Example UML State Machine

information to detect transition-based faults. Note that the difference between TGB and TB encoding is in the level of abstraction, because TGB does not differentiate between transitions with the same trigger-guard but different source or target state. STGB contains both state and trigger-guard information and has therefore the highest level of details. But the extra information may introduce noise when existing faults are of a certain type that could be more directly detected if the encoding contained only the relevant information for those faults. For example, if the existing faults are all detectable by traversing certain states of the system, regardless of how that state was reached (state-based faults), then including triggers and guards in the encoding would result in unnecessary noise in the similarity calculations, as we will show in our empirical analysis.

As an illustrative example, assume that the UML state machine in Figure 1 represents the SUT. Applying an all-transition criterion (with a breadth-first search) on this model results in the transition tree of Figure 2 and a test suite *ts_example* containing *tp*1 to *tp*4. Table 1 shows these four ATCs encoded with SB, TB, TGB, and STGB.

4.2 Similarity matrix generation

Once the ATCs are encoded, they are given to a similarity function (*SimFunc*) which takes two sets/sequences of elements (we use "{ }" to represent sets and "< >" for representing sequences) and assigns a similarity value to each pair. The results of measuring all these similarity values are recorded in a similarity matrix (in case of large test suites we can replace this matrix generation phase with an on-the-fly similarity calculation, which will be discussed in Section 5.4). The similarity matrix can be an upper/lower triangular matrix, since the similarity measure should be symmetric (the similarity between test case A and

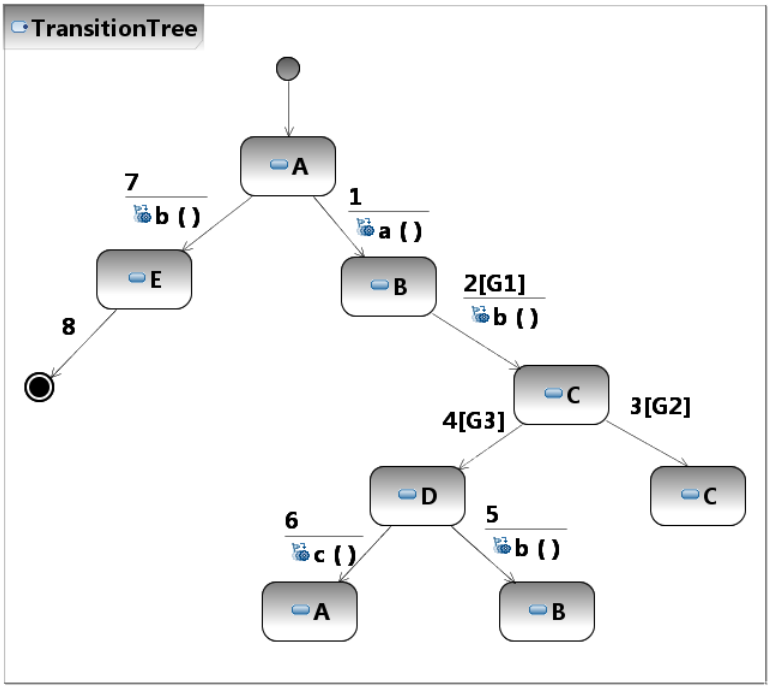


Figure. 2 Example transition tree

test case B is equal to the similarity between test case B and test case A). Therefore, we only need to store half of the matrix.

Given an encoding, one may use different set/sequence-based similarity functions [24]. The main difference between them is that set-based similarity measures, as opposed to sequence-based ones, do not take the order of elements into account. For example, if the encoding is SB, and the first test case corresponds to a path in the state machine that visits state A and then state B, whereas the second test case corresponds to a path that visits state B and then state A, set-based similarity functions, unlike sequence-based functions, assume these two test cases as identical. In [24] we have introduced three set-based and three sequence-based functions. In the following of this section, those functions and two more set-based functions, which are used in this study, are defined and explained by examples.

Table 1 Encoded ATCs using SB, TB, TGB, and STGB

Abstract Test Case	SB Encoding	TB Encoding	TGB Encoding	STGB Encoding
*tp*1	<A,E,END>	<7,8>	<b,8>	<A,b,E,8,END>
*tp*2	<A,B,C,D,A>	<1,2,4,6>	<a,[G1]b,[G3],c>	<A,a,B,[G1]b,C,[G3],D,c,A>
*tp*3	<A,B,C,D,B>	<1,2,4,5>	<a,[G1]b,[G3],b>	<A,a,B,[G1]b,C,[G3],D,b,B>
*tp*4	<A,B,C,C >	<1,2,3>	<a,[G1]b,[G2] >	<A,a,B,[G1]b,C,[G2],C >

4.2.1 Set-based similarity functions

Set-based similarity measures are widely used in data mining [25] to assess the closeness of two objects described as multidimensional feature vectors, where the set is composed of the features' values [8]. In our case, each ATC is a vector of elements. Each element is either a state, a transition, or a trigger-guard, depending on the encoding of the ATC (SB, TB, TGB, and STGB). Each element in the vector is taken from a limited alphabet of possible states, transitions, or trigger-guards in the model. However, the vector size can be different since the length of ATCs may vary.

Hamming Distance. Hamming Distance is one of the most used distance functions in the literature and is a basic edit-distance. The edit-distance between two strings is defined as the minimum number of edit operations (insertions, deletions, and substitutions) needed to transform the first string into the second [26-28]. Hamming is only applicable on identical length strings and is equal to the number of substitutions required in one input to become the second one [26]. If all inputs are originally of identical length, the function can be used as a sequence-based measure. However, in realistic applications test inputs usually have different lengths. Therefore, to obtain inputs of identical length, a binary string is produced to indicate which elements, from the set of all possible elements of the encoding, exist in the input. This binary string, however, does not preserve the original order of elements in the inputs and therefore leads to a set-based similarity function.

In our case, to use Hamming Distance, each ATC is represented as a binary string Ham_{tp}, where $|Ham_{tp}|$ is equal to the number of all possible elements for that encoding (e.g., $|Ham_{tp}|$ is the number of all states, if a SB encoding is used). A bit in Ham_{tp} is *true* only if the ATC contains the corresponding element (e.g., the state for SB). 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 strings, and not differences as in the standard Hamming Distance, and then divides it by the number of all possible elements for that encoding ($|Ham_{tp}|$).

As an example, let us take *tp3*={A,B,C,D,B} and *tp4*={A,B,C,C} as input sets from Table 1, where the encoding is SB. Let us assume that bits one to five in any Ham_{tpi} represent the existence of states A to E in the *tpi*, then Ham_{tp3} = {11110} and Ham_{tp4} = {11100}, and as a result: HAM(*tp3*,*tp4*)=4/5=0.8.

Jaccard Index, Gower-Legendre(Dice), and Sokal-Sneath(Anti-Dice) measures. This family of measures is defined based on commonalities and differences between two sets of inputs. The general formula for calculating similarity of two ATCs (denoted by A and B) with these similarity functions is:

$$sim(A,B) = \frac{|A \cap B|}{|A \cap B| + w(|A \cup B| - |A \cap B|)}$$

Where $|A \cap B|$is the size of intersection of A and B and $|A \cup B|$ is the size of union of A and B. When *w=1*, the above formula corresponds to Jaccard Index or Jaccard similarity coefficient (denoted JAC), that is the size of the intersection divided by the size of the union of the sample sets. When *w*=1/2, we get the Gower-Legendre or Dice measure (denoted GOW), and when *w*=2 this is called Sokal-Sneath or Anti-Dice (denoted SOK). The difference between these three measures is on the weight that the measure puts on differences between input sets ($|A \cup B| - |A \cap B|$), where for the same inputs, similarity is higher/lower for GOW/SOK than JAC. For instance, for *tp3* and *tp4* in the above example, with the same encoding SB, $|tp3 \cup tp4| = 4$, and $|tp3 \cap tp4| = 3$. Therefore, JAC(*tp3*,*tp4*) = 3/4 = 0.75, GOW(*tp3*,*tp4*) = 6/7 = 0.86, and SOK(*tp3*,*tp4*) = 3/5 = 0.6.

Counting function. The Counting function is defined based on the similarity measure used in [21] for comparing two sets of transitions in a specific modeling language (Labeled Transition System). We have defined a generalized version of this function (denoted as CNT) as the number of identical elements in the input sets divided by the average length of inputs (in our case ATCs). CNT is equal to GOW in cases where all elements are unique in the input ATCs. Note that, to be precise, inputs of CNT are not sets, since their elements are not unique, but for the sake of simplicity we consider them as set-based measures. As an example, CNT(*tp3*,*tp4*)=3/4.5=0.67 for *tp3* and *tp4* with SB encoding.

4.2.2 Sequence-based similarity functions

In sequence-based similarity functions, as opposed to set-based functions, the order of elements in the input sequences matters. As we discussed, edit distance functions such as the base version of the Hamming Distance are sequence-based. However, the Hamming Distance is limited to identical length input strings. In this sub-section, we introduce

Levenshtein as an edit-distance function. We also introduce the concepts of Global and Local alignment from bioinformatics and describe one similarity function per alignment.

Levenshtein. One of the most well-known algorithms implementing edit-distance, and which is not limited to identical length sequences, is Levenshtein [28]: 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 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) [28]. 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. For example, given the same inputs as previous examples (*tp3* and *tp4* using SB encoding), the first three elements in *tp3* and *tp4* match, and there is one mismatch and one gap at the end. Since matches increment the similarity value and mismatches and gaps do not change the value, then LEV(*tp3,tp4*)=3.

Global alignment and Needleman-Wunsch similarity function. An alignment of two sequences is a mapping between positions of their elements [27]. An alignment score is assigned to each pair of sequences, measuring the matches, mismatches, and gaps. The goal of an alignment algorithm is finding the best way of positioning the elements of input sequences to maximize the alignment score. Global alignment is an algorithm that aligns the entire input sequences. In our context, we are not interested in the actual aligned ATC pairs. However, the score assigned to each pair is actually a similarity value, which is defined based on matches, mismatches, and gaps. The most basic global alignment algorithm is Needleman-Wunsch (NW) [27] where the scoring function is actually the same as the Levenshtein similarity function. We use match score +3, mismatch -2, and a gap penalty of -1 (which are justified in the next paragraph) as the operation weights of NW similarity function for global alignment.

Note that the only difference between LEV and NW are the operation weights. In the case of LEV, we assume the basic Levenshtein [28] definition (with +1 for match and zero for mismatch and gap), and in the case of NW we use different operation weights as it is usual in Global alignment. The chosen weights are based on our context (USBT) rationale. Given the fact that STCS focuses on similarity, we do not want to miss any similarities between ATCs. Therefore, we give more weight to similarities as otherwise most values

would be negative. Every gap and mismatch decreases the total similarity value but we penalize mismatches more than gaps. That is because in USBT, when comparing two ATCs, gaps only represent missing behavior, but any mismatch distinguishes ATCs from each other. To assess this weighting scheme we also had a small tuning that compared the effectiveness of different NWs when match scores vary between 0 to 5 and gap penalty and mismatch scores between 0 to -5. The results showed that higher values for matches than for mismatches are the best. However, there is not a significant and consistent improvement while increasing the differences between these values. Therefore, we kept the relative weighing order but with the smallest differences in the actual values. One can argue that NW settings may not be the best possible weighting. Although this is indeed true, any tuning is expensive and problem dependant.

Let us look at one example. Given *tp3* and *tp4* with SB encoding from Table 1, NW(*tp3,tp4*)=3*(+3)+1*(-2)+1*(-1)=+6 and the actual aligned sequences are: <A,B,C,D,B> and <A,B,C,C, – >, where the dash symbol identifies a gap. The dynamic programming [29] implementation of the algorithms, along with examples, can be found in [27]. The scoring matrix F for NW 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 *x* and *y* are input sequences. 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 *x* and *y* is F[*N*][*M*] where *N* and *M* are the lengths of *x* and *y* respectively.

Local alignment and Smith-Waterman similarity function. In Local alignment, the goal is to find the best alignment for sub-sequences of the given input sequences. The output of a Local alignment is two aligned substrings with the highest alignment score. Like Global alignment, we are not interested in the actual aligned sequences, but the score assigned to each pair is a similarity function. The most basic Local alignment algorithms is Smith-Waterman (SW) [27], where the scoring matrix F is defined in a similar way as in the NW scoring matrix, but with a small change:

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

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

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 most similar subsequence between input sequences as well. We used the same operation weights as NW for SW with the same reasoning. As an example, SW(*tp3,tp4*)=3*(+3)=+9 and the actual aligned sequences are: <A,B,C> and <A,B,C>.

4.3 Minimizing similarities

In the last step of STCS, the similarity matrix and the desired number of selected test cases (test selection size) is given to an algorithm which minimizes the average pair-wise similarity between all pairs of ATCs in the selected set. Note that this problem is, in general, an NP-hard problem (traditional set cover) [30]. Therefore, using an exhaustive search in most realistic problems 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. For example, in one of our case studies, the search space size for *n*=28 (~10% of the test suite with size 281) is $\binom{281}{28}$ ≅2.9*10^38. Given a similarity matrix, we have analyzed four strategies to select the most diverse test cases: (1) Greedy-based, (2) Clustering-based, (3) Adaptive Random Testing, and (4) Search-based.

4.3.1 Greedy-based minimization

In this paper, what we call a similarity-based Greedy algorithm (*SimGrd*) is an exact implementation of the selection technique which is used in the only published related (STCS for MBT) work [21] (that we are aware of), and we will use it as an STCS baseline. Assume we want to select *n* ATCs (s_n) out of a test suite (*TS*). In each step, a pair of ATCs that has the maximum similarity in the similarity matrix (maximum *SimFunc*(tp_i , tp_j)) is chosen. If there is more than one pair with the same similarity (maximum similarity) all are chosen. Then, among all ATCs in all selected pairs the one with shortest length is selected and removed from the original *TS*. The algorithm is stopped when there are *n* ATCs remaining from the *TS*. Selecting the shortest ATC is done to avoid purely random

elimination assuming that longer ATCs can detect more faults [31]. However, some degree of randomness might still affect the results if more than one ATC in the set of selected pairs have the shortest length. There are potential improvements to this algorithm, but we keep this as the original proposal in [21] to have a valid baseline of comparison.

4.3.2 Clustering-based minimization

Clustering algorithms partition objects into groups, using a similarity/distance measure between pairs of objects and pairs of clusters, so that objects belonging to the same groups are similar and those belonging to different groups are dissimilar. Though clustering techniques are not minimization techniques, the fact that clusters are formed based on the similarities/distances among inputs makes these algorithms a potential solution for our selection problem. To select *n* ATCs, a clustering-based technique partitions the ATCs in the original test suite into *n* non-empty clusters so that (dis)similar ATCs (do not) fall in the same cluster. Then a one-per-cluster sampling method (in this study we randomly select one ATC per cluster) is applied to provide the final *n* diverse ATCs. In this paper, we have tried two of the most used clustering techniques in software engineering, which will be introduced next.

K-Means clustering. The first clustering algorithm used in this paper (KMC) is inspired from the most popular clustering algorithm, K-Means clustering [32]. Though K-Means was proposed over 50 years ago and thousands of clustering algorithms have been designed since then, K-Means and its extensions are still widely used [32]. K-Means objective is to minimize the average squared Euclidean distance of objects from their cluster means [33].

$$\text{Squared Euclidean distance} = \sum_{\vec{x} \in C_k} |\vec{x} - \vec{\mu}(C_k)|^2$$

Where a cluster mean is the centroid $\vec{\mu}$ of a cluster (C)

$$\vec{\mu}(\omega) = \frac{1}{|C|} \sum_{\vec{x} \in C} \vec{x}$$

In our context, where we do not use Euclidean distance but our defined similarity functions, we do not have a geometrical centroid. One alternative could be to define one of the cluster members as the representative of the cluster, but it is not always easy to devise a rationale for such a representative in our context. Because for example, most of the similarity measures, unlike Euclidian distance, are not transitive and violate the *triangle inequality* property [26], which can results in an ATC being similar to a cluster representative (which is similar to all ATCs in the cluster) but not similar at all to any of the ATCs in the cluster.

Our version of K-Means clustering (denoted as KMC), instead of comparing one single cluster representative, uses intra/inter-cluster similarity measures based on Average Linkage. The Average Linkage intra-cluster similarity between an ATC (tp_i) and a cluster (C_x) is defined as the average similarities between the tp_i and all C_x members (tp_x).

$$IntraClusterSim(C_x, tp_i) = \frac{\sum_{tp_x \in C_x and\ tp_x \neq tp_i} SimFunc(tp_x, tp_i)}{|C_x|}$$

Each iteration of KMC assigns an ATC to the cluster with maximum intra-cluster similarity for that ATC. Using intra-cluster similarities, we no longer can use the original stopping criterion of K-Means clustering: "stopping when the average squared Euclidean distance between objects and their cluster centroids does not decrease from iteration *m* to iteration *m*+1", since there is no centroid anymore. Instead KMC uses inter-cluster similarity measure (the average similarity between all possible pairs of ATCs from two clusters) and stop iterating when inter-cluster similarity does not decrease from iteration *m* to iteration *m*+1.

$$InterClusterSim(C_i, C_j) = \frac{\sum_{tp_j \in C_j} IntraClusterSim(C_i, tp_j)}{|C_j|}$$

Agglomerative Hierarchical Clustering. One of the clustering algorithms which has been frequently used in software engineering, including software testing [18, 34, 35], is Agglomerative Hierarchical Clustering (AHC) [8]. AHC starts with forming clusters each containing exactly one object (an ATC in this study). A sequence of merge operations is then performed until the desired number of clusters is achieved. At each step, the two most similar clusters will be joined together. The measure that we use for assessing similarity

between two clusters, inter-cluster similarity, is the Average Linkage. The pseudo-code of the employed AHC follows:

(1) Make one cluster (C_k) per ATC (tp_i).
(2) While the number of clusters is more than *sampleSize* (*n*)
(3) Find the two most similar clusters C_x and C_y (with the maximum *InterClusterSim*(C_x, C_y)).
(4) Merge the two clusters.

4.3.3 Adaptive Random Testing

Adaptive Random Testing (ART) has been proposed as an extension to Random Testing [23]. Its main idea is that diversity among test cases should be rewarded, because failing test cases tend to be clustered in contiguous regions of the input domain. This has been shown to be true in empirical analyses regarding applications whose input data are of numerical type [36]. Therefore, ART is a candidate similarity minimization strategy in our context as well. In this paper, we use the basic ART algorithm described in [23], but we ensure that no replicated ATC is given in output. The pseudo-code for ART is:

(1) Z={}
(2) Add a random ATC to Z
(3) Repeat until |Z|= *sampleSize* (*n*)
(4) Sample K random ATCs that are different from Z
(5) For each of these ATCs k
(6) k.maxSim = max(SimFunc(k , z ∈ Z))
(7) Add the k with minimum maxSim to Z

4.3.4 Search-based minimization techniques

Many software engineering problems can be re-formulated as *search problems*, for which *search algorithms* can be applied to solve them [37]. This has led to the development of a research area often referred to as *Search-Based Software Engineering*, for which several successful applications can be found in the literature [38], with a large representation from software testing [39]. Therefore, in this paper we also analyze the use of search algorithms for STCS.

Given a set of n encoded ATCs (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 ATCs in s_n represented by tp_i and tp_j. The space of all possible sub-sets of size n represents the *search space.* The sets with the minimum fitness values are called *global optima. SimMsr* is used as the *fitness function* to guide the search algorithms to find (near-)optimal sets of ATCs.

A search algorithm can be run for an arbitrarily amount of time. The more time is used, the more elements of the search space can be evaluated. This would lead to better results on average. Unfortunately, in general we cannot know whether an element of the search space is a global optimum, because such knowledge would require an evaluation of the entire search space. Therefore, *stopping criteria* need to be defined, as for example timeouts or fixed number of fitness evaluations.

The minimization problem we address in this paper must address *constraints* on the elements of the search space. In particular, each element is a *set* of ATCs, and therefore no duplicate ATC is allowed. Running a test case twice would not lead to find more faults (as long as the execution of each test case is independent, as it is the case in our industrial case studies). There are several ways to handle constraints [40], and in this paper we simply enforce each search operator to always sample valid sets, because it is the simplest feasible solution in our context (see next section on Random Search, for more information on unique ATC selection).

There are several types of search algorithms that one can choose. On average, on all possible problems, all search algorithms perform equally, and this is theoretically proven in the famous *No Free Lunch* theorem [41]. Nevertheless, for specific classes of problems (e.g., software engineering problems) there can be significant differences among the performance of different search algorithms. Therefore, it is important to study and evaluate different search algorithms when there is a specific class of problems we want to solve, as for example software testing and its sub-problems. This type of comparisons in software testing can be found for example in [42-44]. In this study, we have applied and evaluated six widely used search techniques to minimize *SimMsr*(S_n). In the following sections, we describe each of them in turn.

Random Search. Random Search (RS) is the simplest search algorithm. It samples search elements at random (i.e., sets of n ATCs), and then, once the algorithm is stopped (e.g., due to a timeout), the element with best fitness value is given as output. RS does not exploit any information about previously visited elements when choosing the next elements to sample. Often, RS is used as a baseline for evaluating the performance of other more sophisticated meta-heuristics [39]. Note the difference between RS, which is a search algorithm, and RT which simply selects ATCs at random without any iteration.

What distinguishes alternative RS algorithms is the probability distribution used for sampling the new solutions. In general, a uniform distribution is employed. However, for the problem we address in this paper, we need to guarantee that no duplicate ATC is present in a selected test set. To sample a subset s_n of size n of unique elements from an original set of size k, we use the following procedure to generate s_n. We start from an empty s, and we add one ATC at a time, until n ATCs are inserted. When we add a new ATC, we choose it randomly from the k ATCs. If the chosen ATC is already present in s, we choose another one at random. Because this ATC could be already in s, we repeat this process until we find one ATC that is not present in s. How long is this process going to take? On average, it is really fast. The probability of sampling an ATC that is not in s is equal to $p=(k-|s|)/k$. The goal of STCS is to produce small subsets of effective test cases and, therefore, in general we would have $n<<k$. We can realistically consider the case $n \leq k/2$ (i.e., we consider the cases in which the selected test suites are not larger than 50% of the original suite). In this case, $p=(k-|s|)/k \leq (k-k/2)/k=0.5$. Because we can describe the process of sampling a unique ATC as a geometric distribution with probability p [45], then the expectation E of this process would be $E=1/p \leq 2$. Therefore, to generate a set of n unique ATCs, on average we just need to sample at most $2n$ ATCs.

Hill Climbing. Hill Climbing (HC) belongs to the class of local search algorithms [46]. It starts from a search element, and then it looks at neighbor solutions. A neighbor solution is structurally close, but the notion of distance among solutions is problem dependent. If at least one neighbor solution has better fitness value, then HC "moves" to it and it recursively looks at the new neighborhood. If no better neighbor is found (i.e., the current element represents a *local optimum*), then HC re-starts from a new element in the search space. HC algorithms differ on how the starting points are chosen, on how the neighborhood is defined and on how the next element is chosen among better ones in the neighborhood.

Often, the starting elements are chosen at random, and this is what we employ for the HC used in this paper. We use the common strategy to visit the neighborhood that makes HC move to the first found neighbor solution with better fitness. Another common strategy would be to evaluate first all the elements in the neighborhood, and then moving to the best one (i.e., the so called *steepest ascent*). In this paper, the neighborhood of a set s_n is defined as follows: for each of the n ATCs, consider its replacement with a random ATC that is not already in s_n. The size of the neighborhood is hence n. Notice that considering all possible ATCs, instead of just one at random, would lead to a far too large neighborhood of size $n*(k-n)$, since there is $k-n$ possible neighbors per ATC.

Steady State Genetic Algorithms. Genetic Algorithms (GAs) [47] are inspired from evolutionary theory, and they are the most used search algorithm in search-based software engineering [37-39]. 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, selected 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.

In this paper, we use a steady state GA (SSGA), in which only the offspring that are not worse than their parents are added to the next generations. Parents are chosen using *rank* selection [48]. We use a single point crossover with probability P_{xover} to combine two different parents s_n^x and s_n^y. Each ATC in an offspring is mutated with probability $1/n$. A mutated ATC is replaced by an ATC that is selected at random from the set of all possible ATCs. The crossover and mutation operators could generate invalid elements (i.e., sets with non-unique ATCs). To cope with this problem, the offspring go through a *repair* phase in which all repeated ATCs are randomly replaced with new ones until all are unique (in a similar way in which random sets are sampled, see Random Search in Section 4.3.4).

(1) Sample a population G of m sets of ATCs uniformly from the search space (i.e., the set of all possible valid sets with a given size n)

(2) Repeat until the stopping criterion is met

(3) Choose s_n^x and s_n^y from G

(4) $(\acute{s}_n^x, \acute{s}_n^y) :=$ crossover $(s_n^x, s_n^y, P_{xover})$

(5) Mutate $(\acute{s}_n^x, \acute{s}_n^y)$

(6) if $((\acute{s}_n^x, \acute{s}_n^y)$ is invalid)

(7) Then Repair $(\acute{s}_n^x, \acute{s}_n^y)$

(8) If min $(SimMsr(\acute{s}_n^x), SimMsr(\acute{s}_n^y)) \leq$ min $(SimMsr(s_n^x), SimMsr(s_n^y))$

(9) Then $s_n^x := \acute{s}_n^x$ and $s_n^y := \acute{s}_n^y$

(1+1) Evolutionary Algorithm. (1+1) Evolutionary Algorithm (EA) [9] is a single individual evolutionary algorithm. It starts from a single individual (i.e., an element of the search space) that is in general chosen at random. Then, a single offspring is generated at each generation by mutating the parent. The offspring never replace their parents if they have worse fitness value. In our context, we can see (1+1) EA as being an instance of the SSGA described in the previous section when the population size is set to one single individual.

Memetic Algorithms. Memetic Algorithms (MAs) [49] are a meta-heuristic that uses both global and local search (e.g., a GA with a HC). It is inspired by Cultural Evolution. A meme is a unit of imitation in cultural transmission. The idea is to mimic the process of the evolution of these memes. From an optimization point of view, we can approximately describe a MA as a population-based meta-heuristic in which, whenever an offspring is generated, a local search is applied to it until it reaches a local optimum. A simple way to implement a MA is to use a GA, with the only difference that, at each generation, on each offspring a HC is applied until a local optimum is reached. The cost of applying those local searches is high, hence the population size and the total number of generations are usually lower than in GAs.

Simulated Annealing. Simulated Annealing (SA) [50] is a search algorithm that is inspired by a physical property of some materials used in metallurgy. Heating and then cooling the temperature in a controlled way often brings to a better atomic structure. In

fact, at high temperature the atoms can move freely, and a slow cooling rate gets them fixed in suitable positions. In a similar way, a temperature is properly decreased in SA to control the probability of moving to a worse solution to escape from local optima in the search space.

From an algorithmic point of view, SA is similar to HC. SA stores one element at a time and, at each step of the algorithm, it samples a new neighbor. If this neighbor has better fitness, then SA moves to it and discards the previous element. Otherwise, SA moves to this new neighbor according to a probability function that is based on the current temperature. In contrast to HC, SA does not restart from a random element in the search space in case of local optima. Given a starting temperature T, one common way to reduce it is to update it every x steps, using for example $T'=\lambda T$, where $\lambda<1$.

5 Empirical Study

In this section, we report the design and results of our empirical analysis. The section starts with description of the case studies and research questions, follows by explaining the experiments settings, the study design, and results. The section ends with discussion on scalability and threats to validity of the results.

5.1 Test suites description

There are two different SUTs used in this study, which are subsystems of two industrial software systems. There is also a set of similarity matrices, which are built based on one of the industrial case studies to simulate different SUTs with specific characteristics. The simulated matrices will be explained in Section 5.3.3. The remainder of this section introduces the two industrial case studies.

5.1.1 Case study A

The SUT in case study A is the core subsystem of a video-conference system at Tandberg AS (now part of Cisco), which manages sending and receiving of multimedia streams implemented in C. Audio and video signals are sent through separate channels and there is also a possibility of transmitting presentations in parallel with audio and video. Presentations can be sent by only one conference participant at a time and all others receive it. A three-level hierarchical state machine describes A's behavior and consists of four submachine states. The first submachine state hides three simple states, whereas the second contains two additional submachine states, each having three simple states. This

state machine was modeled by the authors and verified by the company's experts. The flattened version of the state machine (automatically generated by our USBT tool, TRUST [6]) consists of 11 states and 70 transitions. Constraints specifying state invariants and guards are expressed in the Object Constraint Language (OCL) [51] and are used to derive automated test oracles. Applying TRUST, 59 ATCs, covering all transitions in the state machine, are generated as the original test suite. Ten concrete test cases (each concrete test case is an instantiation of one ATC with a given value for each trigger's input parameter) are randomly generated per ATC (with equal probability for each input data value). Running these 590 concrete test cases on four releases of the SUT resulted in detecting four distinct faults. These faults are all reported in bug reports of the releases and are detected by ATCs either by visiting a specific state, taking a specific path, or using a specific input data for the triggers. Note that there are reported faults which are not detected by our test suite because they are either related to functionalities which are not modeled, e.g., user interface, or they are related to non-functional requirements, e.g., robustness behaviors, which are not accounted for in the current model of the SUT.

Since each ATC corresponds to several concrete test cases, which may or may not detect a fault, in our experiments we need to estimate the FDR of a set of ATCs (s_n). Given the 10 concrete test cases per ATC, the FDR of s_n is equal to the average probability of finding the existing faults. Probability P_f of finding a specific fault f with the selected subset of ATCs is equal to one minus the probability of not finding the fault with any of the ATCs in the chosen set: $P_f = (1 - \prod_{i=1}^{n}(1 - p_i))$ where n is the size of the subset and p_i is the estimated probability of detecting fault f with ATC i in the subset: number of times the fault is detected by the 10 test cases generated for that ATC divided by 10. The FDR is hence computed by averaging these probabilities: $\sum P_f / |F|$, where $|F|$ is the number of faults.

5.1.2 Case study B

The SUT in case study B (information about this case study is sanitized due to confidentiality restrictions) is a safety monitoring component in a safety-critical control system implemented in C++. 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 the company experts and our research team. A total of 26 faults 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. The correct, most recent UML state machine was given to TRUST as an input model. Using all-transitions coverage, 281 ATCs and the corresponding executable test cases along with their test oracles were automatically generated. In this case study, if an ATC has the ability to detect a fault, it can be detected by any valid test data for that ATC. Therefore, unlike case study A, we only need one concrete test case per ATC to compute its FDR and the test suite FDR is simply the number of faults detected by the concrete test cases corresponding to the set of selected ATCs (s_n), divided by the total number of detectable faults in the system.

Among the 26 faults, 11 of them were sneak paths (illegal transitions in the modified model) [52]. 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 detected by any test case generated from the model. The remaining 15 faults (detectable by the test cases generated from the model) are used to produce 15 faulty versions of the code by introducing one fault per program. The faults are due to both code and design level faults.

There are four main differences between these two case studies: (1) Case study A is smaller both in terms of the model size (number of states and transitions) and the test suite size (number of ATCs generated for the input model); (2) The number of faults detected by the test suite is much higher for case study B. Recall that we do not account for faults that relate to robustness behavior, which is not modeled in the current state machines, as this case studies focus exclusively on the nominal behavior of the SUTs; (3) The fault detection in case study A depends on the input data, whereas ATCs in case study B either detect or not a fault regardless of input data; and (4) The failure rate (θ)—i.e. the probability that a test case chosen at random from the original test suite triggers any failure (we assume a

uniform probability and not a usage profile)—is much higher for case study A. Notice that, in these case studies, the θ is expected to be much higher than testing scenarios in which all possible test cases are considered. This is because MBT is effective in generating test suites with high fault detection rates. In case study B, 74 out of 281 ATCs detect at least one fault thus yielding θ = 74/281≅0.26. However, θ in case study A is higher. Because in this case study each ATC has a probability of detecting a fault that depend on input data, we calculate the probability for an ATC i to detect at least one fault as $P_i = 1 - \prod_{f=1}^{4}(1 - p_{if})$, where p_{if} is the probability for ATC i to detect fault f. Therefore, the θ is $(\sum_{i=1}^{59} P_i)/59$ = 43.28/59≅0.73. This high θ is a direct consequence of the faults types in case study A. Since the SUT in case study A was rather stable, we had to look back into the earliest releases to be able to detect faults, and therefore, since the SUT was of poor quality (was not well tested at that stage), it contained faults which were relatively easy to detect with MBT.

Case study A is smaller but it has interesting characteristics such as a high failure rate and a dependency between FDR and input data. Therefore, we report the results from both case studies separately to show potential differences due to differences in SUT characteristics. In a subsequent step, we also summarize the results in a more general way by looking at the two cases together along with the results from a simulation study. Table 2 summarizes the SUT features.

5.2 Research questions

The main goal of this study is to propose a model-based test case selection technique that is adjustable based on available testing budget and resources. This is expected to make MBT more scalable in situations where, for a number of possible reasons, test case execution is

Table 2 Summary of case study features

Feature	Case study A	Case study B
Domain	Multimedia systems	Safety control systems
Number of states in the flattened state machine	11	70
Number of transitions in the flattened state machine	70	349
Number of ATCs covering all transition of the state machine	59	281
Number of detectable faults	4	15
Failure rate	73%	26%
Are input data important in fault detection?	YES	No

expensive. To achieve this goal, we perform a series of experiments to (1) investigate many alternative STCS techniques, (2) assess how effective STCS is compared to other techniques, (3) determine in which situations it can be expected to be more effective, and (4) what benefits can be expected in practice. The experiments are designed to answer the following five research questions:

RQ1: How influential are STCS parameters on its effectiveness?

STCS consists of three phases (Section 4) within which many variants of the technique can be formed. Setting parameters of the phases leads to a completely specified STCS technique (one variant). An important question is to assess how the choice of techniques—for encoding, similarity function, and minimization algorithm—affects STCS effectiveness. In other words, we want to understand whether all three parameters matter, which ones have a significant effect, and why it is so.

RQ2: What is the most cost-effective STCS variant?

Since choosing a specific STCS strategy requires to set three parameters out of a large possible number of combinations, one needs clear guidance to choose the best combination possible in a given context. Therefore we aim at identifying the most cost-effective variants of these parameters in the case studies, explain the results, and attempt to generalize them to provide guidelines.

RQ3: What is the practical benefit of using STCS?

Once the best STCS strategy has been identified (RQ2), we want to assess the improvement we gained over the state of the art. The non-STCS baselines of comparison are Random Testing (RT) [17]—randomly selecting *n* ATCs from the original test suite—and coverage-based techniques (described in details in Section 5.3.2). A coverage-based technique maximizes a model coverage measure (e.g., number of covered transitions in USBT) in the selected test cases. We will describe them in more details in the design section (Section 5.3.2). Improvement is achieved either with higher FDR for the same number of test cases or the same level of FDR but with fewer test cases.

RQ4: What is the effect of the failure rate on the effectiveness of STCS?

Since θ in both of our case studies is quite high, it is important to apply STCS in SUTs with lower θ before generalizing the results. Therefore, in this question we specifically address the effect of θ on STCS effectiveness.

RQ5: How one can estimate the minimum number of test cases required for achieving (near) maximum FDR?

In practice choosing a test budget is a tradeoff within constrained limits. For a software tester, it may be possible to argue for a larger budget and obtain it if justified. Therefore, we investigate the relationship between average similarity within a test set and its FDR to assess whether this can help us decide when enough test cases have been selected. Can we conclude that, if after increasing test suite size average pair-wise similarity reaches a plateau, then there is little chance to expect any further improvement in FDR when augmenting further the test suite?

5.3 Experiment design and results

We designed four experiments to answer the five research questions described in Section 5.2. Two different analyses on the results of the first experiment help answer RQ1 and RQ2 and there are three other experiments to answer the three remaining questions, respectively. Each experiment consists of running several STCS variants on the SUT similarity matrices, either actual or manipulated for the sake of simulation. The techniques are implemented by the authors in Java and our large empirical study is concurrently executed on an IBM multicore-based cluster of 84 computer nodes, each with eight cores (each computing node has dual Intel quad-core 2.5GHz processors) with eight GB shared memory and Linux Ubuntu operating system.

5.3.1 Experiment 1: answering RQ1 and RQ2

In this experiment, we look at all the STCS variants discussed in Section 4 and perform two different analyses to answer RQ1 and RQ2. We investigate a total of 320 STCS variants (four encodings: SB, TB, TGB, STGB; eight similarity functions: LEV, NW, SW, CNT, HAM, JAC, SOK, GOW; and 10 minimization algorithms: SimGrd, KMC, AHC, ART, RS, HC, SSGA, (1+1)EA, MA, SA).

There is no specific parameter setting for encoding definitions and similarity matrix generations in the experiments other than what is defined in Section 4. However, certain minimization algorithms need parameter settings. SimGrd, KMC, and AHC involve no parameters. For ART, we used k=10 (as suggested in [23]). For search-based techniques, we made choices based on what is suggested in the literature [47] and our previous experience (e.g., [2, 3, 7, 24]). We use a high crossover probability (0.75) for GA and MA. Mutation probability is the standard one: $1/n$, for SSGA, MA, and (1+1)EA. Population size is set to 50 for SSGA and 10 for MA. The rank selection for SSGA and MA uses a 1.5 bias. For SA, the initial temperature is set to 0.9, and it is reduced by 5% (i.e., $\lambda=0.95$)

every 10*n steps of the algorithm. We will discuss about potential threats to validity of the results due to these parameter settings in Section 5.5.

Each STCS variant is applied on both case studies A and B for different test selection sizes (3 to 15 by intervals of one for case study A and 10 to 140 by intervals of 10 for case study B). The reason that we do not continue after 15 (case A) and 140 (case B) ATCs, respectively, is that most techniques converge to the maximum FDR over these sizes. Comparisons are more important for smaller test selection sizes since they represent more realistic scenarios in practice. For each size, each STCS variant is executed 100 times and FDR is computed for the selected ATCs. In addition, to take into account the random nature of these STCS algorithms, we followed a rigorous procedure to assess whether there is any statistically significant difference among the performances of these STCS variants [53]. To rank the variants, we assign a score to each variant called *variant score*, which is initialized to zero. For each variant and test selection size, we perform 319 non-parametric Mann-Whitney U-tests to assess the statistical significance of FDR differences (if any) between the considered variant and all the others over 100 runs. This resulted in 13*320*319/2=663,520 and 14*320*319/2=714,560 statistical tests for case studies A and B, respectively. If the results of the tests show a significant difference (at level α =0.05) between the variants' results, the effect size is calculated using the A statistic [53, 54] for the FDR of the two variants and the *variant score* of the better variant (with higher A statistic) increments while the score of the worse variant decrements. There will be no change if the test result shows no significant difference (i.e., if the p-values are higher than 0.05). Note that Vargha-Delaney's A statistic, which is an effect size measure, is used instead of comparing the medians of the distributions since we cannot state that the compared distributions have same shape [53]. This statistic estimates the probability that a data point randomly taken from a set (i.e., a probability distribution) will have higher value than another point randomly taken from a second data set. When the two distributions are the same, we would have A=0.5.

After performing all statistical tests, each test selection size of a variant has a score in the range [-319, +319]. The scores are replaced by ranks from 1 to 320 (lower rank represents higher FDR). The average of the ranks over test selection sizes makes a one rank value per pair (variant, case study). We also combined the results from the two case studies in one rank value by averaging the two ranks of each variant. Note that we cannot take both case study results together while performing the ranking, since the number of samples is different in the two cases. Therefore, we first rank variants in each case study

Table 3 The Kruskal-Wallis test results on the effect of different variant parameters on FDR

Parameters	Case study A		Case study B		Both case studies together	
	p-value	Chi-squared	p-value	Chi-squared	p-value	Chi-squared
Encoding	1.344e-14	67.67	< 2.2e-16	170.64	1.874e-5	24.60
Similarity function	< 2.2e-16	161.62	0.8349	3.50	< 2.2e-16	159.64
Minimization algorithm	0.0028	25.11	5.836e-9	56.67	6.886e-6	40.23

and then average them. In the next step, given the rank values for each variant, a Kruskal-Wallis test, which is a non-parametric analysis of variance test, is applied per each of the three variant parameters separately. Table 3 reports the *p*-value and Chi-squared results, which show how much of variance is explained by each parameter from the Kruskal-Wallis tests.

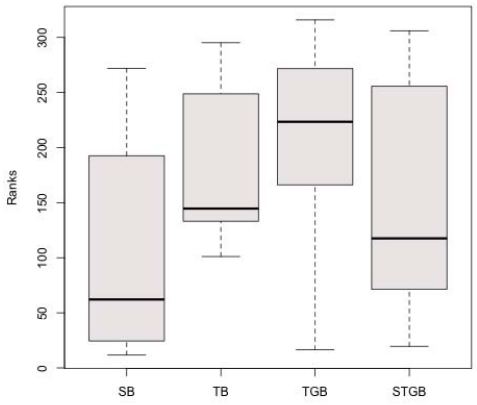
In addition, Figure 3 to Figure 5 show the boxplots of the variant ranks when considering one parameter at a time. For example, Figure 3.a shows the distribution of ranks when using any of the four encodings in case study A as boxplots (the middle line is the median). There are 80 observations per category (eight similarity functions times 10 minimization algorithms). Figure 3.b and Figure 3.c show the same boxplots for case study B and both studies together, respectively. As visible from the boxplots and confirmed by the statistical tests, all parameters have a significant effect on ranks (and consequently the FDR). The only non-significant result is for similarity functions on case study B (*p*-value=0.83). However, by considering both studies together, similarity functions also show significant differences. To answer RQ1, we account for both case studies, and results suggest that all parameters, when taken individually, have a significant effect on the FDR. However, we cannot say which parameter is more important than the other, since the Chi-squared statistic values vary greatly for the three parameters across case studies.

RQ2 can also be answered based on the boxplots by choosing the best parameter value per parameter. However, that might be misleading since there is no consistent, dominant parameter across case studies (RQ1) and interaction effects might take place between parameters, e.g., the best encoding and the best similarity function may not form the best combination. Therefore, the most accurate way of finding the best STCS variant is to analyze the ranks of all 320 variants. To do this, we built a rank table per case study and

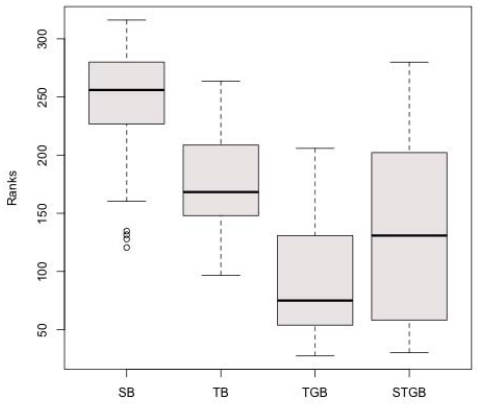
for both case studies together. Table 4 to Table 6 show the first top 20 rows out of 320 rows of each rank table.

Looking at each case study individually shows that JAC, SOK and GOW are the best similarity functions and that (1+1)EA, MA, and SA are the best candidates for minimization algorithm in both case studies. The fact that set-based similarity measures perform better than sequence-based ones shows that the difference between the orders of elements in ATCs does not play an important role in their FDR. This may be, however, a characteristic of the faults in our case studies. Search-based techniques definitely overcome Greedy, ART, and clustering-based techniques— this may be expected since they are meant to be optimization techniques, as opposed to the other alternatives. SB is the best encoding for case study A followed by STGB whereas TGB followed by STGB are more effective on case study B. The difference in best encoding between case studies A and B can be explained by differences in the number of faults, fault rate, and types of faults in the two case studies. In case study A, faults are mostly state faults and are easy to find. Therefore, a simple state-based encoding is enough to differentiate ATCs according to the faults they detect, whereas in case study B we need to account for more details in the test to differentiate ATCs revealing different faults.

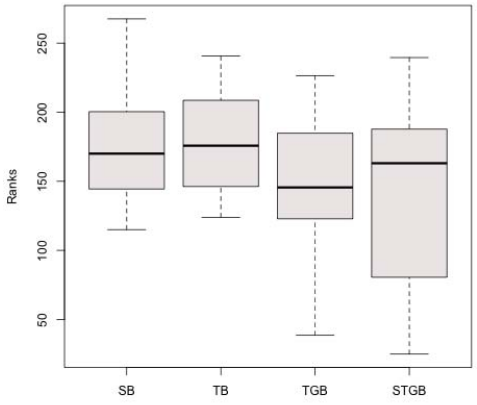
To draw a conclusion based on both case studies together, we need to look at the rank of the best STCS variants in Table 4, where the average ranks over the two case studies are computed. STGB appears to be the best encoding—most probably because it contains all necessary information from the path—but it is also slightly suboptimal in each study individually, as it introduces irrelevant information in the similarity computations. Unless we know exactly the type of faults we can expect in a case study, we should select as a default STGB to ensure we do not miss any useful information from the ATCs, though it might be somewhat suboptimal. Among JAC, SOK and GOW, which are the best similarity functions in the case studies, GOW shows the best results when averaging the ranks. That implies that assigning more weight to similarities than differences—that is what differentiates GOW compared to JAC and SOK—seems more effective, though differences are relatively minor. Regarding minimization algorithms, we already knew that search-based techniques like SSGA are more effective than ART and clustering-based techniques, such as AHC [2]. Looking at the results of each study individually confirms that finding. However, a more interesting result is that (1+1)EA, which is a simplified GA (a non-population-based GA), is more effective in both case studies than SSGA and actually is the best when considering both case studies together. In general, whether this



(a) Case study A

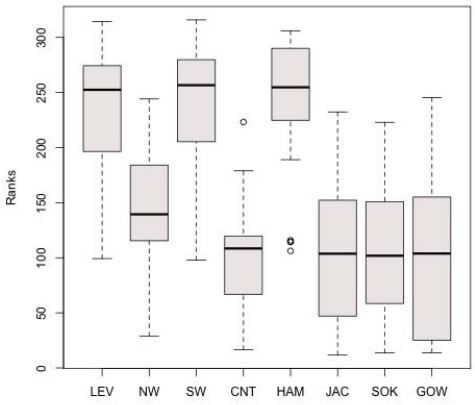


(b) Case study B

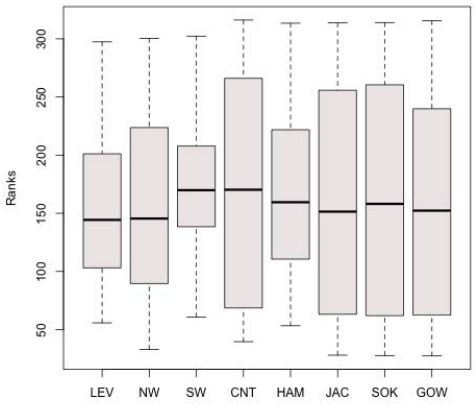


(c) Both case studies together

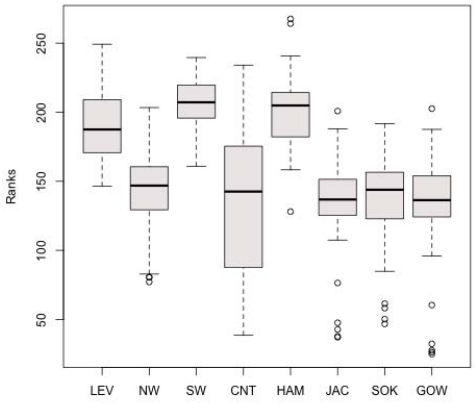
Figure 3 The effect of encoding on FDR of STCS



(a) Case study A

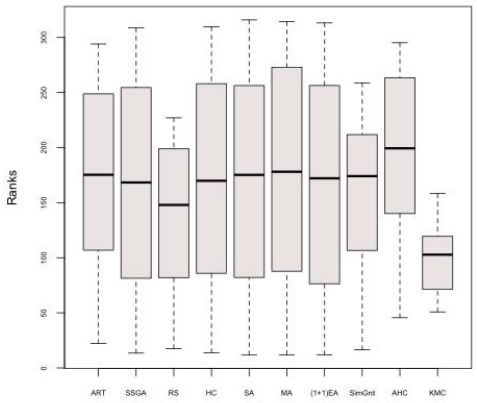


(b) Case study B

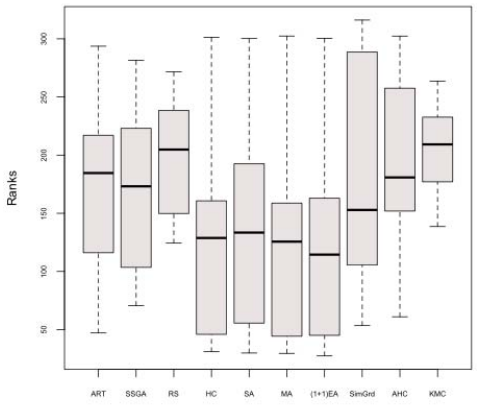


(c) Both case studies together

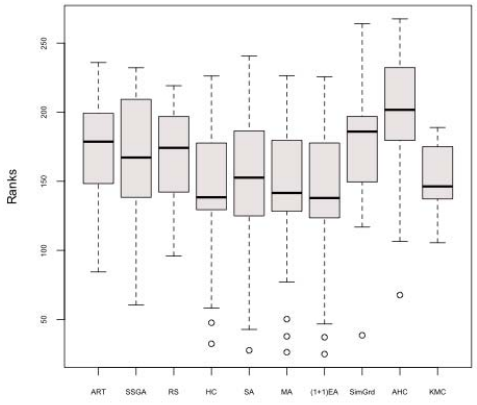
Figure 4 The effect of similarity function on FDR of STCS



(a) Case study A



(b) Case study B



(c) Both case studies together

Figure 5 The effect of minimization algorithm on FDR of STCS

Table 4 Top 20 STCS variants in the rank table for both case studies together

Encoding			Similarity Function												Minimization Algorithm						Rank
SB	TB	TGB	STGB	LEV	NW	SW	CNT	HAM	JAC	SOK	GOW	ART	SSGA	RT	HC	SA	MA	(1+1)EA	SimGrd	AHC	KMC	
			STGB								GOW							(1+1)EA				24.898
			STGB								GOW						MA					26.251
			STGB								GOW					SA						27.596
			STGB								GOW				HC							32.352
			STGB						JAC									(1+1)EA				37.117
			STGB						JAC								MA					37.717
		TGB					CNT												SimGrd			38.497
			STGB						JAC							SA						42.745
			STGB							SOK								(1+1)EA				46.699
			STGB						JAC						HC							47.565
			STGB							SOK							MA					50.229
			STGB							SOK					HC							58.166
			STGB								GOW		SSGA									60.375
			STGB							SOK						SA						61.408
		TGB					CNT													AHC		67.663
			STGB				CNT								HC							72.205
			STGB						JAC				SSGA									76.401
			STGB				CNT											(1+1)EA				76.554
			STGB		NW												MA					77.107
			STGB				CNT									SA						78.923

Table 5 Top 20 STCS variants in the rank table for case study A

Encoding			Similarity Function												Minimization Algorithm						Rank
SB	TB	TGB	STGB	LEV	NW	SW	CNT	HAM	JAC	SOK	GOW	ART	SSGA	RT	HC	SA	MA	(1+1)EA	SimGrd	AHC	KMC	
SB									JAC							SA						11.654
SB									JAC								MA					11.962
SB									JAC									(1+1)EA				12.000
SB									JAC				SSGA									13.462
SB										SOK						SA						13.577
SB										SOK								(1+1)EA				13.654
SB											GOW				HC							13.731
SB										SOK					HC							13.962
SB											GOW					SA						14.500
SB											GOW						MA					14.962
SB											GOW		SSGA									15.538
SB											GOW							(1+1)EA				16.038
SB									JAC						HC							16.192
SB										SOK			SSGA									16.423
		TGB					CNT												SimGrd			16.423
SB										SOK							MA					16.462
SB											GOW			RS								17.423
			STGB								GOW							(1+1)EA				19.654
			STGB								GOW					SA						19.692
			STGB								GOW						MA					20.038

Table 6 Top 20 STCS variants in the rank table for case study B

Encoding			Similarity Function												Minimization Algorithm						Rank
SB	TB	TGB	STGB	LEV	NW	SW	CNT	HAM	JAC	SOK	GOW	ART	SSGA	RT	HC	SA	MA	(1+1)EA	SimGrd	AHC	KMC	
		TGB									GOW							(1+1)EA				27.357
		TGB								SOK								(1+1)EA				27.679
		TGB							JAC									(1+1)EA				28.000
		TGB							JAC								MA					29.536
		TGB								SOK						SA						29.857
			STGB								GOW							(1+1)EA				30.143
		TGB								SOK							MA					30.214
		TGB									GOW					SA						30.286
		TGB									GOW						MA					30.286
		TGB							JAC							SA						30.536
		TGB							JAC						HC							30.964
		TGB									GOW				HC							31.679
			STGB								GOW				HC							32.357
			STGB						JAC									(1+1)EA				32.464
			STGB								GOW						MA					32.464
			STGB		NW											SA						32.929
			STGB		NW													(1+1)EA				33.500
		TGB								SOK					HC							34.107
			STGB								GOW					SA						35.500
			STGB							SOK								(1+1)EA				35.821

result stands depends on the search landscape, but considering the simplicity of the (1+1)EA, for example in terms of parameters, together with the results from the two case studies, we suggest to adopt it as the best minimization algorithm for STCS.

When we analyzed the cost of different techniques, we saw no significant difference between the execution time when generating encoded ATCs. There is a significant difference in execution time between sequence and set-based similarity matrix generations. But the more effective techniques (set-based ones) are actually the cheaper ones. Among set-based techniques there is no significant difference. For example, for the bigger case study (case study B) the whole matrix generation time is in the range of 400 to 800 milliseconds (ms) using the set-based functions depending on the encoding and the function, but sequence-based techniques need more than three seconds to generate the same size matrices. These differences are not practically significant for these case studies, but for larger SUTs the difference may matter. In Section 5.4, where we investigate scalability issues, more details about the cost of sequence and set-based techniques will be provided.

To perform a fair comparison of minimization algorithms, we evaluate the FDRs of different STCSs while keeping their execution cost equal. We force algorithms to have exactly the same cost by using the same stopping criterion. When comparing search-based minimization algorithms and ART, we can force them to have the same number of similarity comparisons and thus obtain a platform independent cost measure. Given a test selection size n, search-based techniques stop after 10,000 fitness evaluations (each consisting of $n*(n-1)/2$ similarity comparisons). For ART, the candidate size is 10 (resulting in $10*n*(n-1)/2$ similarity comparisons). Therefore, we run ART 100 times and select the best among them to have the same number of total similarity comparisons as the search-based techniques. However, it is not possible to use the same cost measure for SimGrd, KMC, and AHC. Therefore, we have to rely on execution time even though this is an imperfect measure of cost [39]. What makes comparisons simpler however is that AHC and KMC, which are worse in terms of FDR, are also more expensive, thus dismissing them as valid alternatives. SimGrd is the only algorithm that is quicker but less effective. We borrowed this algorithm from the only related work on STSC [21] and treat it as a baseline of comparison for our work. Therefore, we do not change its design and the current algorithm cannot be run for a longer time to achieve better results. Thus, we cannot

fix the selection time of SimGrd in the same way we do with search-based algorithms in order to perform a fair comparison. However, SimGrd does not appear to be an interesting alternative since: (a) It is one of the worst algorithms in terms of rank based on Figure 5 (its median is very low and there is a high variation in its results, especially visible in case study B) and it only appears twice out of 60 among the best variants in Table 4 to Table 6; (b) its FDR cannot be improved by assigning more time to its execution; (c) the algorithm is more expensive for smaller test selection sizes, which are more of interest in practice, since instead of selecting test cases, it removes them from the test suite to reach the preferred size. Therefore smaller test selection sizes require more eliminations that makes them more expensive. The cost of selection by SimGrd for small test selection sizes is even more expensive than search-based techniques. For example, for case study B and test selection size 10, SimGrd takes 175ms on average whereas (1+1)EA only needs 17ms.

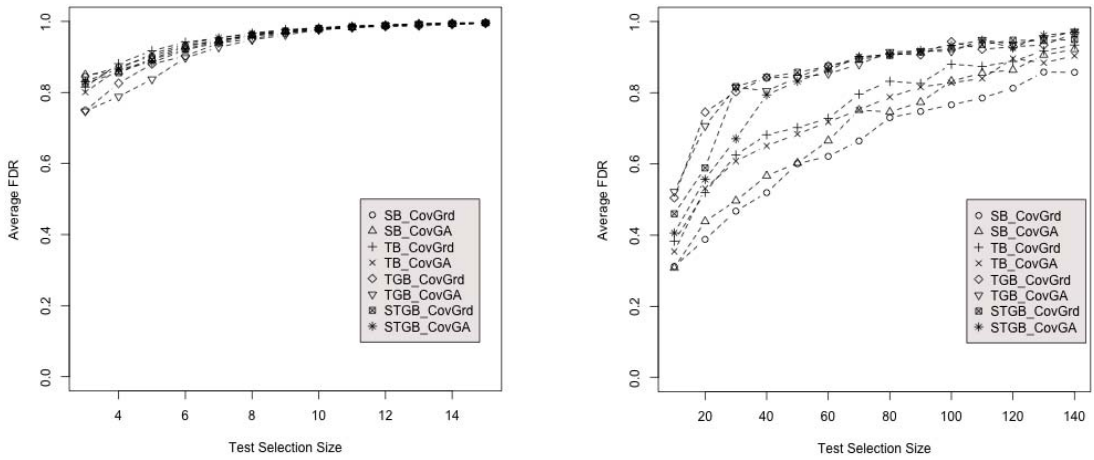
As a result, we can summarize Experiment 1 as follows: All STCS parameters are potentially influential on FDR and the most cost-effective variant is STGB, GOW, and (1+1)EA for encoding, similarity function, and minimization algorithm, respectively. In the rest of this paper we denote this variant as Best_STCS.

5.3.2 Experiment 2: answering RQ3

In Experiment 2, we are interested in evaluating STSC cost and effectiveness compared to state of the art, non-STCS test case selection strategies. The goal is to assess the magnitude of improvement that STSC provides, which can be measured both in terms of higher FDR with the same number of test cases but also by achieving the same FDR with fewer test cases. In this experiment, based on the results of Experiment 1, we take Best_STCS as the best representative of STCS selection techniques and compare it with other possible non-STCS selection strategies, which are either coverage-based test case selection or random testing (RT). RT is simply randomly selecting *n* ATCs with uniform probability from the original test suite. Coverage-based techniques vary in two dimensions: what should be covered and how this coverage should be maximized. Therefore, we first need to determine the best coverage-based technique as a representative of this category. In a model-based selection context, coverage can only be defined based on the ATCs, since no execution or source code information is available. The most well-known USBT coverage criterion, which is used for test case selection in the literature, is transition coverage [19]. Based on our encoding for STCS, we also define state, trigger-guard, and state-trigger-guard based coverage criteria. Regarding maximization of coverage, we apply two of the most used

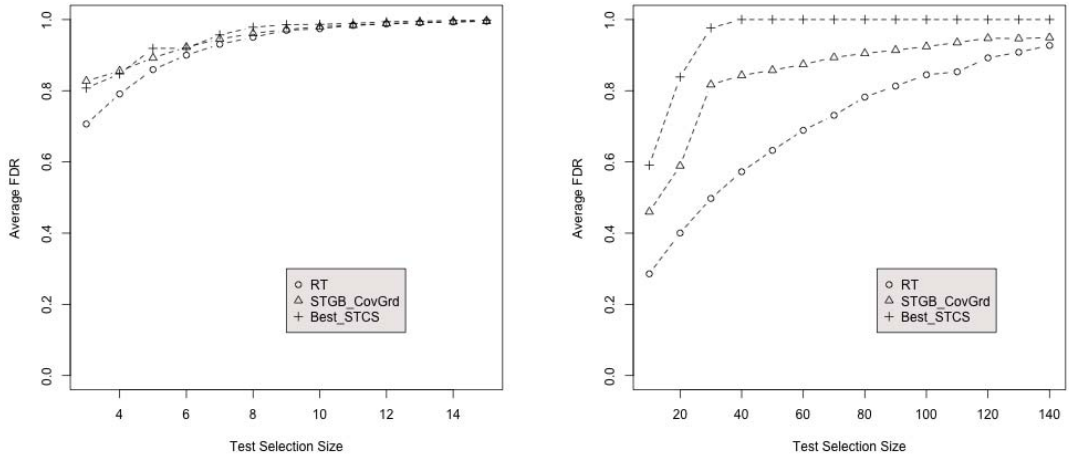
techniques in the literature: Greedy [19, 55] and GA [20, 56]. In the Greedy-based technique (CovGrd), which is inspired by the additional coverage technique [10], a greedy algorithm selects in a stepwise manner one additional ATC which covers the most uncovered elements, based on the selected coverage criterion. In our context, an element is one of the following: state, transition, and trigger-guard, based on the encoding. In case of the STGB encoding, the goal is covering both all states and all trigger-guards. The second approach (CovGA) uses a SSGA (with the same settings and stopping criterion as the SSGA used in STCS) to maximize the total coverage of the selected test set. Applying these two techniques (CovGrd, CovGA) with the four coverage criteria (corresponding to the four encodings), we evaluated the FDR of the selected ATCs on the two case studies. Figure 6 shows FDR means, over a range of test selection sizes, for the eight variants of coverage-based techniques labeled with the name convention: *Encoding_MaximizationAlgorithm*. The first observation (more visible on Figure 6.b, case study B) is that the effectiveness trend among coverage criteria is the same as the effectiveness trend among encodings in STCS techniques. For case study A the ordering of FDRs is SB>STGB>TB>TGB and for case study B it is TGB>STGB>TB>SB. However, the differences in case study A (Figure 6.a) are practically negligible because of the high failure rate (θ). Since there are a few easy-to-detect faults in the SUT of case study A, regardless of the coverage criterion, both Greedy and GA techniques are able to catch the faults. Thus, the differences in terms of FDR are very small. Therefore, taking the average FDR from both case studies together, we choose STGB as the best coverage criterion. Comparing CovGrd and CovGA, we could not find any practical difference in any of the case studies and, considering the high cost of SSGA compared to a Greedy algorithm, we suggest STGB_CovGrd as the best representative of coverage-based category.

Comparing Best_STCS, STGB_CovGrd, and RT, Figure 7 reports their mean FDRs over the range of test selection sizes in case study A (Figure 7.a) and case study B (Figure 7.b). We also reported the *A* statistic (effect size measure) for comparing Best_STCS vs STGB_CovGrd and Best_STCS vs. RT, respectively, in Figure 8.a (for case study A) and Figure 8.b (for case study B). The *A* statistics show the probability that a selected test set by Best_STCS (randomly taken from the 100 runs) will have higher FDR than a selected test set by STGB_CovGrd and RT(randomly taken from the 100 runs), respectively.



(a) Case study A (b) Case study B

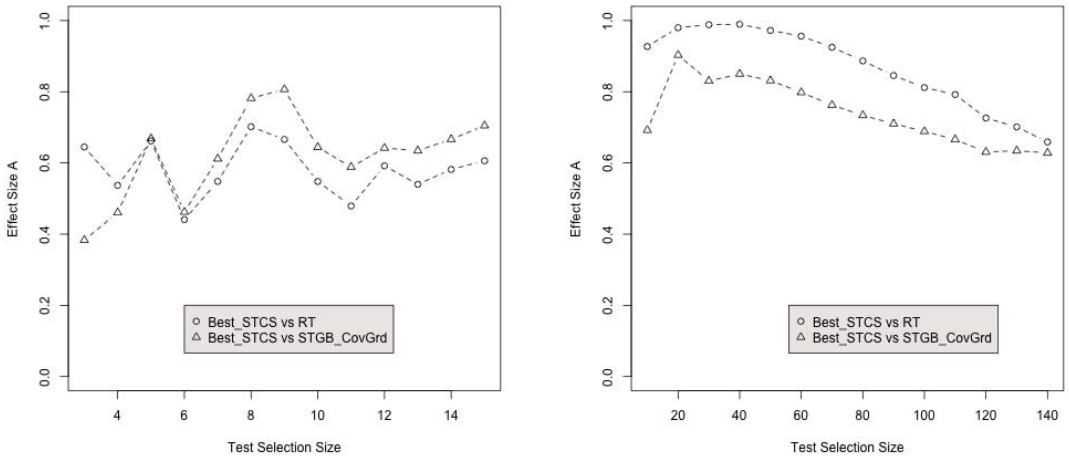
Figure 6 The FDR comparison of different coverage-based test case selection techniques for different test selection sizes



(a) Case study A (b) Case study B

Figure 7 The FDR comparison between the best STCS (Best_STCS) and non-STCS techniques (STGB_CovGrd and RT) for different test selection sizes

Based on these results, we clearly see the improvement in FDR in case study B. The mean FDR of Best_STCS reaches 100% with only 40 ATCs (around 14% of the original test suite) while the two alternatives cannot reach 100% even with 140 ATCs (half of the original test suite). Using Best_STCS, 50% to 80% fewer test cases are required to achieve the same FDR as STGB_CovGrd and RT. Looking at the percentage of mean FDR improvement, especially for smaller test selection sizes, Best_STCS provides between 15%



(a) Case study A (b) Case study B

Figure 8 The effect size measure (A statistic) when comparing FDR of Best_STCS against STGB_CovGrd and RT for different test selection sizes

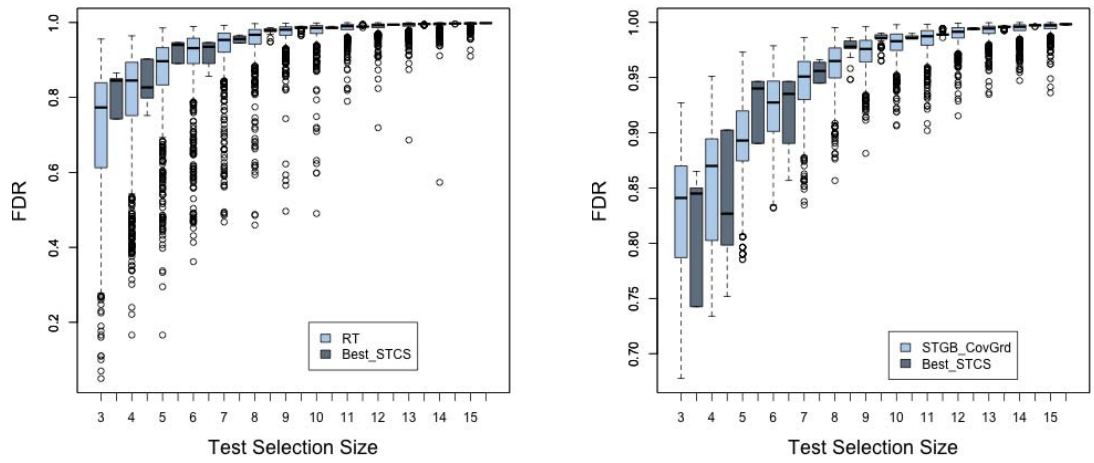
to 45% improvement over STGB_CovGrd and 80% to 110% over RT. The maximum improvement is obtained around test selection sizes 20 to 30 where Best_STCS has already reached a very high FDR though STGB_CovGrd lags far behind. We also have applied a Mann-Whitney U-test on the FDRs of different test selection sizes, comparing Best_STCS with STGB_CovGrd and RT. All *p*-values are very low (zero or close to zero). This phenomenon is also visible in Figure 8.b, where the effect size around test selection size 20 is close to 1.0, thus showing that Best_STCS is nearly always a better option in terms of effectiveness for case study B.

However, the improvements are not practically significant in case study A. The differences in mean FDRs (from Figure 7.a) are small. In some cases there is not even a statistically significant difference between them and the *A* statistic also does not show very high values. The small differences among techniques have been also observed in Figure 6 when comparing coverage-based techniques. The most plausible explanation is the high failure rate (θ) since most selected test sets, even if they are chosen by a simple selection technique, are good enough to detect most of the easy-to-detect faults. Therefore, a more complex technique such as Best_STCS may not be of practical help in this case.

Running a deeper analysis of this case study, we plot the FDR distributions as boxplots in Figure 9.a (Best_STCS vs. RT) and Figure 9.b (Best_STCS vs. STGB_CovGrd). It is easy to see two trends: (1) In the interval from 7 to 9 Best_STCS performs better in terms of all the statistics discussed above. For test selection sizes above 10 all techniques

perform the same (almost reaching maximum possible FDR) and for very small test selection sizes (3 to 6) none of the techniques can dominate the other. (2) Best_STCS and RT have the least and most variance in results, respectively. Note that, in practice, selecting a subset of test cases with a high variance technique can, in the worst case, lead to a very low FDR. Therefore, even for case study A, we would prefer using Best_STCS than STGB_CovGrd or RT.

Cost is also an important factor in selecting the best technique. The simple algorithms used in CovGrd and RT definitely result in very low execution time. RT execution time is extremely low (e.g., in case study B, execution time is less than 0.1ms for any test selection sizes). STGB_CovGrd's execution time is around 11ms for any test selection size. However, Best_STCS takes, for example, 17ms, 51ms, 108ms, and 189ms for test selection sizes 10, 20, 30, and 40 respectively. Though these differences are not practically significant (especially for smaller test selection sizes which are more of interest in test case selection), they may become so in the case of larger systems. In practice, if considering each test case execution time (e.g., in our case studies, it is in the range of minutes), an additional 200 milliseconds for test case selection is negligible, as long as the more expensive technique can yield the same or higher FDR than alternatives with fewer test cases. In other words, given that the total cost of a solution is the selection time plus the execution time of the selected test cases, when each test case execution time is in the range of minutes and total selection time in the range of hundreds of milliseconds, any reduction



(a) Best_STCS vs. RT

(b) Best_STCS vs. STGB_CovGrd

Figure 9 Distribution of FDRs over 100 runs for different test selection sizes of Best_STCS, STGB_CovGrd, and RT as boxplots on case study A

in the number of test cases is much more effective in reducing the total cost than saving milliseconds during selection. We will discuss more about the cost of the techniques for larger systems in Section 5.4.

In summary, the results of Experiment 2 answers RQ3 by showing that, most of the time, Best_STCS results in equal or higher FDR with fewer ATCs when compared with state of the art alternatives (coverage-based selections and RT). In a few cases, Best_STCS is not more effective than these baselines, but because it shows less variance, it is still a less risky technique to use. In addition, in most cases the FDR improvement is very significant (e.g., for some test selection sizes and case studies, 40% and 110% improvement is achieved when compared to STGB_CovGrd and RT, respectively). Even for case study A, where the failure rate (θ) was high, the Best_STCS never performed worse than the baselines (in terms of *A* statistics), except for very small test selection sizes of three, four, and six ATCs (Figure 8.a), where in those cases Best_STCS shows less variance in results (Figure 9). Therefore, we suggest using Best_STCS as a model-based test case selection technique, even for small test suites, since there is essentially no harm using Best_STCS. The extra cost for small test suites is negligible, Best_STCS has the potential to result in much better FDR and a reduced MBT cost. If the test suite execution cost is negligible, then there is no need for any kind of selection, since the entire test suite can probably be executed within the project time constraints.

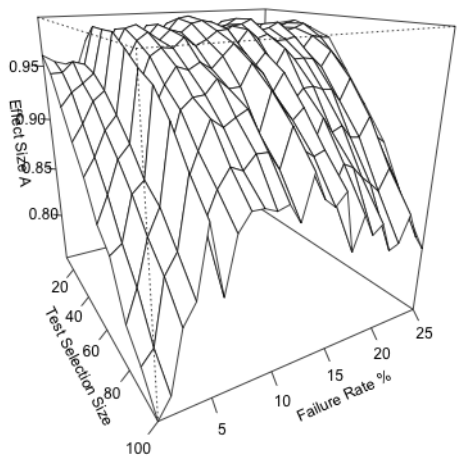
5.3.3 Experiment 3: answering RQ4

In Experiment 3, we simulate test suites with different θs to study its effect on the effectiveness of STCS. We are specifically interested in investigating whether STCS still provides higher FDR than coverage-based and random selection when the original test suite θ is low. To run such experiment, we need test suites with lower θs than that of our current case studies. Therefore, we take case study B, which has the lowest θ (26%) to start with, and use it to form 25 types of test suites with equal size (200), but with different θs ranging from 25% down to 1%. These test suites are formed by removing a different subset of 81 ATCs from the original test suite of 281 ATCs. Because we do not generate artificial ATCs or change the fault detection pattern of ATCs (which faults are detected by which ATCs), this design preserves most features of the original industrial case study. Notice that, regardless of the matrix generation procedure, we cannot build a fault detecting matrix with θ lower than 1/(281-73)=0.48% from case study B. There can be many test suites with size 200 and a given θ based on case study B. Therefore, we generate 30

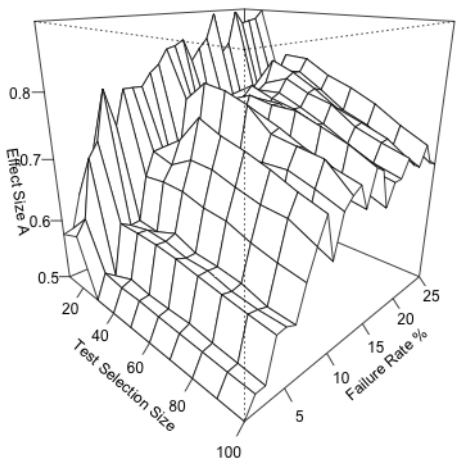
different test suites per θ and apply Best_STCS, STGB_CovGrd, and RT 100 times for different test selection sizes on each test suite.

Figure 10.a and Figure 10.b show three-dimension graphs to visualize variations in effect size measure (*A* statistic) when comparing the FDR of ATCs selected by Best_STCS vs. RT and Best_STCS vs. STGB_CovGrd, respectively. Two parameters are varied to explain variations in *A* statistic values: θ from 1% to 25%, test selection size from 10 (5% of the test suite) to 100 (50% of the test suite). Results from the graphs, reporting the *A* statistic of the FDRs for each value of θ and test selection size (there are 3000 data points for each *A* statistic value: 100 runs and 30 test suites), show that Best_STCS is never worse than its alternatives (*A* statistic ≥ 0.5) regardless of test selection size and θ. This answers RQ4. If we carefully analyze the *A* statistic change over different test selection sizes for different θs, we can see that the trend is the same as what has been seen in Figure 8.b (the original case study B with θ=26%). There is a test selection size value where *A* is maximum (e.g., test selection size 20 in Figure 8.b and Figure 10.b) and from there on, increasing the test selection size decreases *A* since any technique can detect faults with a large enough test selection size.

Another interesting observation is that, for a given moderate test selection size (so that not all techniques are effective), *A* is higher when θ is large. A higher θ provides the better



(a) Best_STCS vs. RT



(b) Best_STCS vs. STGB_CovGrd

Figure 10 The effect size measure (A statistic) when comparing FDR of Best_STCS against STGB_CovGrd and RT for different test selection sizes (10 to 100) and failure rates (1% to 25%)

techniques with more opportunity to increase FDR by selecting the right ATCs. However, beyond a certain point, the differences between techniques start to reduce and we finally reach a point where all ATCs detect faults and technique will result in the same FDR as RT. In summary, the results of Experiment 3 suggest that Best_STCS dominates its alternatives and that it is not an artifact of the high failure rate (θ) of our case studies. It also confirms that Best_STCS may not be significantly better when the original test suite's θ is extremely low or high. This highlights the importance of applying STCS on systematically generated test cases resulting from a cost-effective MBT strategy, in order to guarantee a high enough θ. On the other side of the range, extremely high θs are unlikely on realistic systems.

5.3.4 Experiment 4: answering RQ5

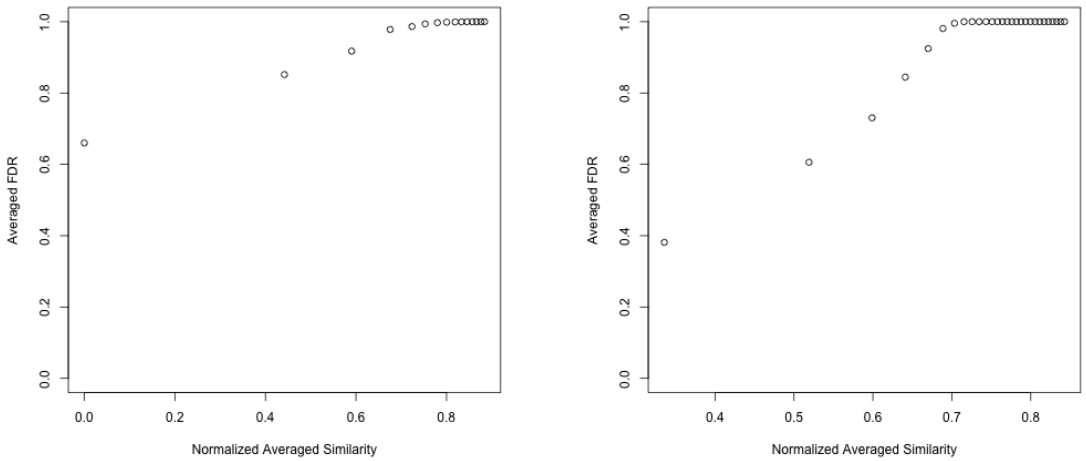
Though in practice the test budget may be imposed by external constraints, the tester may have some degree of freedom to increase it if it is believed to yield significant benefits in terms of fault detection. We are therefore interested in finding a heuristic that helps the tester choose an optimal test selection size for STCS. Our heuristic is based on the average similarity of ATC pairs in the set ($SimMsr(s_n)$), which is the same as the fitness function of (1+1)EA in Best_STCS. If there is a linear correlation between $SimMsr(s_n)$ and the FDR of s_n over different test selection sizes, we can get a fairly good estimate of the FDR of the selected ATCs based on their $SimMsr(s_n)$.

In Experiment 4, we re-apply Best_STCS on both case studies and calculate the normalized $SimMsr(s_n)$ per output selected test set. We normalize the values between zero and one, so that we can plot them with FDR values in one overlay graph using the same scale. To do so, we need to know the maximum and minimum possible $SimMsr(s_n)$. The minimum similarity measure corresponds to the set of two ATCs with minimum pair-wise similarity ($Minimum(SimFunc(tp_i, tp_j))$). That is because adding any ATC to the set of two ATCs with minimum similarity will increase (or not change) the average of the similarities among ATCs in the set. Therefore, maximum $SimMsr(s_n)$ is equal to $SimMsr(TS)$, where *TS* is the entire test suite, and the normalized similarity measure is:

$$Norm(s_n) = \frac{SimMsr(s_n) - Minimum(SimFunc(tp_i, tp_j))}{SimMsr(TS) - Minimum(SimFunc(tp_i, tp_j))}$$

Figure 11 shows the average FDR(s_n) (y-axis) as a function of $Norm(s_n)$ (x-axis) for case study A (Figure 11.a) and case study B (Figure 11.b), over 100 runs. The test selection size *n* was varied between 2 to 30 (50% of the test suite) by intervals of two and 5 to 140 (50% of the test suite) by intervals of five for case studies A and B, respectively. It is clearly visible in these two case studies that there is a monotonic increase in average FDR as test selection size and *Norm* increase. After a certain *Norm* threshold, the average FDR gets close to the maximum and plateaus. This was expected since increasing *Norm* results from increasing the test selection size and, of course, the average FDR naturally converges towards 1.0.

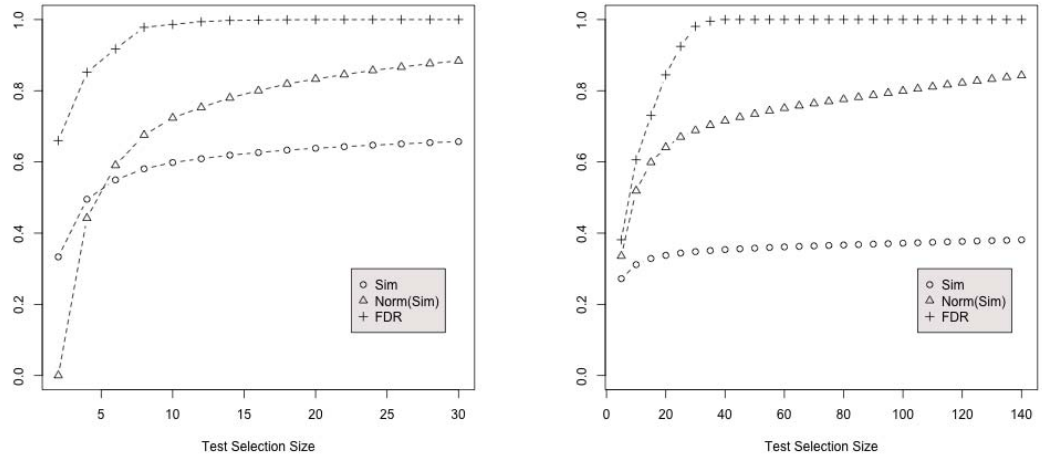
However, what is more interesting is the near-linear relationship between *Norm* and FDR, before reaching the plateau. In practice, while considering increases in test selection size, one can look at the trend of *Norm* (or $SimMsr(s_n)$)) and choose to increase the test selection size only when it triggers a significant increase in $SimMsr(s_n)$ afterwards. Because of the observed relationship between *Norm* and FDR, we know that if the former does not significantly increase, we are unlikely to obtain significant increases in FDR. However, a significant increase in *Norm* may not guarantee an increase in FDR if its maximum value has already been reached, which we cannot know in practice. The relationship between *Norm* and FDR is expected to vary significantly as, depending on the



(a) Case study A with test selection sizes between 2 to 30 (50% of the test suite), intervals of two.

(b) Case study B with test selection sizes between 5 to 140 (50% of the test suite), intervals of five.

Figure 11 Scattered plot of the averaged FDR (y-axis) and the normalized similarity measure (x-axis) of selected test cases over 100 runs using Best_STCS



(a) Case study A (b) Case study B

Figure 12 Average FDR, similarity measure, and the normalized similarity measure for different test selection sizes using Best_STCS

failure rate (θ) of the test suite, maximum FDR can be achieved with different test selection sizes. This makes it impossible to know beforehand the value of *Norm* by which maximum FDR is reached. This implies that guiding the choice of test selection size based on increases in *Norm* may lead to a conservative choice that will guarantee that an increase in FDR is possible but not certain if it has already reached its maximum, thus leading to an unnecessarily large test selection size.

Figure 12 shows the average FDRs along with *SimMsr* and *Norm*, for different test selection sizes. We can see the same trend in both studies (Figure 12.a for case study A and Figure 12.b for case study B): the *elbow point* (when the last significant decrease in the slope of the tangent line appears) in the *Norm* or *SimMsr* curve happens in the same interval of test selection sizes as when FDR reaches its maximum. We can also match these test selection sizes with test selection sizes corresponding to the *elbow points* in Figure 11 a and b. Table 7 (case study A) and Table 8 (case study B) reports the average FDR and *Norm* for different test selection sizes close to the *elbow points* of the curves in Figure11 and Figure 12. The test selection sizes that correspond to the *elbow points* in the scattered plots are in bold. It is clear that the gain in average FDR and *Norm* from those test selection sizes onward is not significant. Therefore, in practice one can decide about the number of test cases to be executed based on the testing budget (maximum affordable) and the increase in *Norm* values (maximum necessary) when increasing test selection size. For

Table 7 Average FDR and Norm values for test selection sizes close to the elbow points in scattered plot of Average FDR vs. Norm for case study A

	***n*=4**	***n*=6**	***n*=8**	***n*=10**	***n*=12**
Norm	0.441931	0.590401	0.675518	**0.723759**	**0.752873**
Average FDR	0.851713	0.9176	0.978124	**0.986415**	**0.993835**

Table 8 Average FDR and Norm values for test selection sizes close to the elbow points in scattered plot of Average FDR vs. Norm case study B

	***n*=15**	***n*=20**	***n*=25**	***n*=30**	***n*=35**
Norm	0.598903	0.641039	0.670007	**0.688725**	**0.703399**
Average FDR	0.7304	0.844533	0.924667	**0.980867**	**0.995533**

example, in case study A, one would determine that beyond a test selection size of 10 (*elbow points*), the gain in FDR would likely be much smaller.

In summary, answering RQ5, we can say that by observing how average similarity among test cases increases as test selection size increases, we can identify the point above which similarity starts increasing at a much slower pace and FDR is not likely to increase significantly. This is made possible by the presence of a near linear relationship between similarity and FDR until the latter reaches its maximum.

5.4 Discussion on scalability of STCS

The main motivation for STCS is to make MBT scalable, but how scalable are STCS techniques themselves? In this section we discuss about how STCS scales up to larger inputs. Note that the input of an STCS is a set of ATCs (original test suite). A larger input means more ATCs and/or longer ATCs. Therefore, we look at the scalability issue from these two points of view. Scalability can be discussed for each step separately. Encoding is the cheapest phase, since ATCs are already generated by the MBT tool. The only extra cost is eliminating the unnecessary elements from the ATCs to produce the encoded sets/sequences. Therefore, encoding linearly scales up with increasing the ATC length and test suite size.

In general, the time complexity of set-based and sequence-based techniques for calculating the similarity of two ATCs (*tp1* and *tp2*) is O(|*tp1*|+|*tp2*|) and O(|*tp1*|*|*tp2*|) respectively, where |*tpi*| is the length of *tpi*. The matrix generation, in total, needs |*TS*|*(|*TS*|-1)/2 similarity calculations, where |*TS*| is the number of ATCs in the test suite (e.g., NW takes in average three seconds for a 281*281 similarity matrix—

(281*280)/2=39,340 similarity value calculations—in case study B, but GOW only requires half a second, on a PC with Intel Core2 Duo CPU 2.40 Hz). Therefore, similarity matrix generation in Best_STCS also scales up fairly well, linearly with ATC length and polynomial ($O(|TS|^2)$) with test suite size. However, the polynomial growth in similarity matrix size is a memory scalability problem. One potential solution is on-demand similarity calculation instead of storing all paired similarities. We can also use a hash table to save the similarity of the most used ATCs in the minimization process. We did not investigate these techniques since it was not necessary for our two industrial case studies.

The most time consuming phase of an STCS is the minimization of similarities, which is an iterative search in Best_STCS. Since we always can set up a time limit as stopping criterion of (1+1)EA, Best_STCS is always applicable for large test suites. However, reducing the search time (compared to the problem size) degrades its effectiveness. Then the question is finding the problem size threshold from where Best_STCS, given the same fixed time budget, is not more effective than baselines anymore. Precisely answering this question requires many more empirical studies on very large industrial SUTs. Unfortunately, obtaining these artifacts for research purposes is not always possible. To cope with this problem, we applied Best_STCS on simulated similarity matrices with different sizes (note that the scalability of this step only depends on the number of ATCs and not the ATCs' length, since in this step we already have a similarity matrix as an input for the search). Using similarity matrices with 600, 6000, and 12000 ATCs—generated by randomly assigning values[1] to each pair and keeping the failure rate (θ) and number of faults the same as the test suite in case study B—we realized that keeping the same number of fitness evaluations as in this study (10,000), the actual extra time required when we increase the number of ATCs is small and very negligible compared to the improvement of FDR that we potentially get using Best_STCS instead of non-STCS techniques. For example, selecting 20 ATCs out of 281 ATCs of case study B takes 51ms on a PC with Intel Core2 Duo CPU 2.40 Hz, whereas it takes 75ms, 130ms, and 139ms when the test suite size is 600, 6000, and 12000, respectively. However, if we also increase the test selection size with the same proportion of the test suite as before, the selection cost is much more. For example, if we select 850 out of 12000 ATCs in the test suite (almost the same proportion as selecting 20 out of 281 ATCs in case study B), we need 85 seconds.

[1] Note that the exact similarity values do not matter, since we are only interested on the selection cost in larger matrices and not the actual FDR of the resulting selected test sets.

But in practice, even 85 seconds is a very negligible cost for selecting from a large test suite of 12000 ATCs.

Given the fact that the total cost of test suite execution ($Cost_{TS}$) is $Cost_{TS} = Cost_{sel} + n * Cost_{tc}$, (where $Cost_{sel}$ is the selection overhead, n is the number of selected ATCs, and $Cost_{tc}$ is the cost of each test case execution) then, obviously, not executing the unnecessary test cases is worth spending some extra seconds for test case selection, especially when each test case execution is costly. Therefore, in summary, we found Best_STCS to be a scalable selection technique, which is applicable in very large systems with reasonable cost while keeping its high effectiveness.

5.5 Discussion on validity threats

In this subsection, we discuss the potential threats to the validity of the study using the framework introduced in [57] for empirical studies in software engineering.

Construct validity*:* In this study any comparison is based on the cost or effectiveness of selection techniques. To evaluate the effectiveness of a selection technique, we need a measure to assess how effective at detecting faults a selected test set is. We use the fault detection rate (FDR), which is based on real faults, as explained in Section 5.1. There are three cost measures considered in the experiments: (a) selection time (actual time spent by the selection technique), (2) number of similarity calculations (which is used in comparing search-based minimization algorithms), and (3) test case execution time. Each test case execution time is taken from the industrial case studies and does not depend on the experiment settings. The number of similarity calculations is also a platform independent measure, but using actual selection time comes with known problems [39] such as platform dependency. However, in this study, this measure is used only to compare different techniques with the same execution settings. Therefore, the relative differences are used, not the exact platform dependant values. The differences between test case selection overhead and test suite execution time is so large that our results would hold even by running test case selection on slower machines. We do not have a discussion on memory consumption of the selection techniques, since we do not use it as a cost measure when comparing different techniques in our case studies. However, it is briefly discussed when discussing about the scalability of STCS in Section 5.4.

Internal validity*:* All encoding techniques, similarity functions (except alignment algorithms), and minimization algorithms are implemented by the authors. Any potential defect in the implementation may be a threat to internal validity. In addition, parameter

settings of similarity functions and search techniques may have an effect on their effectiveness. Regarding similarity functions, there exist techniques that are known to have influential parameters (e.g., NW). This means that they could possibly work better with some fine tuning. However, that would compromise the applicability of the approach as tuning can be time consuming and difficult in practice. Regarding minimization algorithms, Greedy and clustering-based techniques do not have parameters to be set. While comparing the remaining techniques (ART and search-based techniques), to alleviate possible threats to internal validity, we used (wherever applicable) equal values for the techniques' parameters such as stopping criterion and mutation rate. However, it is again possible that one of the algorithms from the search-based category, with a specific tuning, would work better than (1+1)EA with any parameter settings. But again, that would affect the applicability of the technique, since the current parameters are taken from the literature and using any other parameter values would require careful and time-consuming tuning.

When we generated the simulated similarity matrices with different θs in Experiment 3 and different sizes in experiments on scalability, we minimized threats to internal validity by keeping most features of the original industrial case study untouched to avoid introducing confounding factors. The goal was also to reduce external validity threats by making the SUT as similar as possible to real systems. However, the fault detection pattern among ATCs might not be the same if we had an industrial case study with the same θ or size.

Conclusion validity*:* Because randomized algorithms are affected by chance, to reduce the threats to conclusion validity we followed a rigorous statistical procedure to analyze the collected data. One hundred independent runs of each algorithm were performed to account for random variation and to collect a sufficient number of observations on the FDR and cost of each selected test set generated by a selection technique. In Experiment 3, where we also had randomness in the similarity matrices, 30 matrices were generated per θ. All conclusions are supported by non-parametric statistical tests (Mann-Whitney U-test and Kruskal-Wallis test). In Experiment 1, the number of observations was different in the two case studies and, as a result, we did not apply statistical tests on the combined data. Rather, the conclusions are based on the average of the results from the two case studies. To compare selection techniques and assess the magnitude of the differences in cost and FDR, we used an effect size measure (*A* statistic) in addition to showing boxplots.

***External validity**:* Our results rely on two industrial case studies using real faults, complemented by a set of simulated test suites. The SUTs are from different domains with different characteristics (e.g., different sizes and number of faults) and the simulated matrices try to generalize the results in two dimensions (different θs and different sizes). However, replicating our studies in various domains as many times as possible is of course required to gain confidence in our results and better understand their limitations. Despite the fact that one can never be sure whether case study results generalize to other systems, we have carefully tried to qualitatively explain our results, thus contributing to understand how they might generalize.

6 Related Works

Though STCS is a new topic in MBT, similar ideas have already been applied in the context of regression testing. Similarity in that domain is mostly defined using some type of code-level coverage of the test cases. For example, in [13] the similarity function is based on all def-use pairs coverage and they use a classification algorithm as a minimization 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 [18, 34, 35, 58]. Greedy search, adaptive random selection, and clustering are used in these studies for selection/prioritization. In [59] 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). Feldt *et al.* [60] has taken a different approach by proposing a diversification technique which is driven by execution related information of the test cases such as the test setup, arguments, control flow, outcome of evaluation, etc. They have applied an information distance function on the description of the test cases. The problem with all these approaches is that they need source code coverage and/or previous execution information which are not available in our context when we do system level, black-box testing and select test cases to minimize test execution.

Ledru *et al.* [22] have introduced a similarity-based prioritization 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 an edit-

distance function. This approach is missing the encoding phase and results in using noisy data (platform dependant information in the test scripts) when applying a similarity/distance function.

There is also a category of test case diversification techniques which are based on data diversity. ART, as introduced in Section 4.3.3, for example, is one of the most well-known techniques in this group. Vega *et al.* [61] also applied test data variance as a test quality measure. In MBT, ATCs do not contain test data and concrete test cases are generated by adding specific test data to each ATC. Test data variance techniques may be useful when generating the concrete test suite from ATCs. However, we are interested in reducing the set of ATCs as much as possible before concretizing them. This means that these techniques can be complementary to STCS, if we employ a diversity-based test data generation technique to select input data for the selected ATCs.

To the best of our knowledge, the only STCS technique applied on the model level was recently introduced in [21], where sequences of transitions in a Labeled Transition System model of the SUT are used to represent test paths. The similarity function is our CNT, as defined as Counting Function in Section 4.2.1, and the selection technique is a Greedy search, that is SimGrd in Section 4.3.1. If we tailor their approach for USBT, the encoding is similar to our TB described in Section 4.1. Therefore, their technique can actually be considered as one of our 320 variants. If we consider their technique as an STCS baseline of comparison, Best_STCS is on average the best variant (18^{th} best variant for case study A and 6^{th} best for the case study B) and is a much better choice than their variant, which is ranked 127^{th} for case study A, 111^{th} for case study B, and 66^{th} on average.

This study is an extension of the work we reported in [2, 3, 24]. The general idea of STCS is introduced in [3] and SB, TB, and TGB encodings are compared while using a CNT similarity function and a SSGA/SimGrd algorithm on case study B. We also compared the variant identified by TGB, CNT, and SSGA with coverage-based selection and random testing. In [24] we have focused on comparison between six similarity functions (all similarity functions introduced in this paper except GOW and SOK) on the same case study. In [2], we further evaluated the approach proposed in [3] when replacing SSGA with AHC and ART. Also, we conducted an experiment to investigate why diversifying test cases improves FDR by showing that, given our similarity function, test cases which detect distinct faults are dissimilar and test cases that detect a common fault are similar. In [7], we conducted a series of experiments investigating the properties of test suites with respect to similarities among fault revealing test cases and identified the ideal

situation for STCS. We also proposed a rank scaling technique for modifying similarity values to address outliers problem in STCS (i.e., a small group of very different test cases).

7 Conclusion and Future Work

In practice, system level testing on real hardware platforms or test networks may be a highly expensive task and very constrained. Therefore, an ideal automated testing approach should be adjustable to the time and resource constraints of the project. This is especially essential for large systems where automated systematic testing, such as model-based testing, typically results in very large test suites. In this paper, we introduced a family of model-based test case selection techniques, called similarity-based test case selections (STCS), which, given a test selection size, minimize the similarity among selected test cases to increase the chance of detecting more faults. We first investigated the different STCS parameters (namely encoding, similarity function, and minimization algorithm) and showed that all parameters are potentially influential on fault detection. Among 320 identified STCS variants, we found the technique (Best_STCS) with state-trigger-guard-based encoding, Gower-Legendre similarity function, and (1+1) Evolutionary Algorithm to be the most cost-effective technique on average on two industrial case studies. Using Best_STCS, much higher FDR is achieved for the same number of test cases compared to the baselines (e.g., for some test selection sizes and case studies, 40% and 110% improvement is achieved compared to the best coverage-based selection and random testing, respectively). This leads to very large savings in terms of number of test cases that do not need to be executed (up to 80% reduction in the number of test cases required for detecting the same number of faults). We also found Best_STCS more effective than other baseline selection techniques regardless of test selection size and failure rate. The scalability of different Best_STCS steps was investigated for larger test suites and test cases. In addition, we proposed a method, based on monitoring change in average similarity when test selection size increases, to help test manager in deciding about the best test selection size within their constraints.

A possible future work can be combining similarity-based and coverage-based selection techniques by applying a multi-objective search technique [62] that minimizes similarities while it maximizes coverage of the selected test cases. Another extension of this work is to assign weights to test cases based on estimates of their execution cost and modify the

selection technique to minimize the total execution cost. Analysis of the search space properties in this field of application is also an interesting further investigation.

Acknowledgements

The authors wish to thank Marius Liaaen, from Tandberg AS, now part of Cisco, for helping us in conducting some of the experiments.

References

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

[2] H. Hemmati, A. Arcuri, and L. Briand, "Reducing the Cost of Model-Based Testing through Test Case Diversity," in *22nd IFIP International conference on Testing Software and Systems (ICTSS), formerly TestCom/FATES*: LNCS 6435, 2010(b), pp. 63-78.

[3] 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, pp. 267-276.

[4] S. Dalal, A. Jain, and J. Poore, "Workshop on Advances in Model-Based Software Testing," in *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, 2005, p. 680.

[5] F. W. Vaandrager, "Does it Pay Off? Model-Based Verification and Validation of Embedded Systems!," Radboud University Nijmegen2006.

[6] 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.

[7] H. Hemmati, A. Arcuri, and L. Briand, "Empirical Investigation of the Effects of Test Suite Properties on Similarity-Based Test Case Selection, ," in *4th International Conference on Software Testing, Verification and Validation (ICST)*, 2011.

[8] R. Xu and D. Wunsch, "Survey of Clustering Algorithms," *IEEE Transactions on Neural Netwoks,* vol. 16, pp. 645-678, 2005.

[9] S. Droste, T. Jansen, and I. Wegener, "On the analysis of the (1+1) evolutionary algorithm," *Theoretical Computer Science,* vol. 276, pp. 51-81, 2002.

[10] 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.

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

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

[13] A. d. S. Simão, R. F. d. Mello, and L. J. Senger, "A Technique to Reduce the Test Case Suites for Regression Testing Based on a Self-Organizing Neural Network

Architecture," in *30th Annual International Computer Software and Applications Conference (COMPSAC)*, 2006, pp. 93 - 96.
[14] A. Orso, H. Do, G. Rothermel, M. J. Harrold, and D. S. Rosenblum, "Using component metadata to regression test component-based software," *Software Testing, Verification and Reliability,* vol. 17, pp. 61-94, 2007.
[15] S. McMaster and A. Memon, "Call-Stack Coverage for GUI Test Suite Reduction," *IEEE Transactions on Software Engineering,* vol. 34, pp. 99-115, 2008.
[16] G.-V. Jourdan, P. Ritthiruangdech, and H. Ural, "Test Suite Reduction Based on Dependence Analysis," in *Computer and Information Sciences – ISCIS 2006*. vol. 4263/2006: Springer Berlin / Heidelberg, 2006, pp. 1021-1030.
[17] R. Hamlet, "Random testing," in *Encyclopedia of Software Engineering*: Wiley, 1994, pp. 970-978.
[18] D. Leon and A. Podgurski, "A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases," in *14th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2003, pp. 442-456.
[19] B. Korel, G. Koutsogiannakis, and L. H. Tahat, "Model-Based Test Prioritization Heuristic Methods and Their Evaluation," in *3rd Workshop on Advances in Model Based Testing, A-MOST*, 2007, pp. 34-43.
[20] X. Y. Ma, B. K. Sheng, and C. Q. Ye, "Test-Suite Reduction Using Genetic Algorithm," in *Advanced Parallel Processing Technologies*. vol. 3756: Springer Berlin / Heidelberg, 2005, pp. 253-262.
[21] 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, in press,* 2009.
[22] 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, pp. 510-514.
[23] T. Y. Chen, F.-C. Kuoa, R. G. Merkela, and T. H. Tseb, "Adaptive Random Testing: The ART of test case diversity," *Journal of Systems and Software,* vol. 83, pp. 60-66, 2010.
[24] H. Hemmati and L. Briand, "An Industrial Investigation of Similarity Measures for Model-Based Test Case Selection," in *21st IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2010, pp. 141-150.
[25] P. N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*: Addison Wesley, 2006.
[26] G. Dong and J. Pei, *Sequence Data Mining*: springer, 2007.
[27] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*: Cambridge University Press, 1999.
[28] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*: Cambridge University Press, 1997.
[29] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2 ed.: The MIT Press, 2001.
[30] A. P. Mathur, *Foundations of Software Testing*, 1 ed.: Addison-Wesley Professional, 2008.
[31] A. Arcuri, "Longer is Better: On the Role of Test Sequence Length in Software Testing," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2010, pp. 469 - 478

[32] A. K. Jain, "Data Clustering: 50 Years Beyond K-means," *in press with Pattern Recognition Letters.,* 2009.
[33] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*: Cambridge University Press, 2008.
[34] 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, pp. 454-477, 2007.
[35] 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, pp. 201-212.
[36] L. J. White and E. I. Cohen, " A domain strategy for computer program testing," *IEEE Transactions on Software Engineering (TSE),* vol. 6, pp. 247-257, 1980.
[37] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology,* vol. 43, pp. 833-839, 2001.
[38] M. Harman, "The Current State and Future of Search Based Software Engineering," in *Future of Software Engineering*: IEEE Computer Society, 2007, pp. 342-357.
[39] 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,* vol. 36, pp. 742-762, 2010.
[40] Z. Michalewicz and M. Schoenauer, "Evolutionary Algorithms for Constrained Parameter Optimization Problems," *Evolutionary Computation* vol. 4, pp. 1-32, 1996.
[41] D. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Transactions Evolutionary Computation* vol. 1, pp. 67-82, 1997.
[42] A. Arcuri and X. Yao, "Search Based Software Testing of Object-Oriented Containers," *Information Sciences,* vol. 178, pp. 3075-3095, 2008.
[43] M. Xiao, M. El-Attar, M. Reformat, and J. Miller, "Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques," *Empirical Software Engineering,* vol. 12, pp. 183-239, 2007.
[44] M. Harman and P. McMinn, "A theoretical and empirical study of search based testing: Local, global and hybrid search," *IEEE Transactions on Software Engineering (TSE),* vol. 36, pp. 226-247, 2010.
[45] W. Feller, *An Introduction to Probability Theory and Its Applications*, 3rd ed. vol. 1: Wiley, 1968.
[46] E. H. L. Aarts and J. K. Lenstra, *Local search in combinatorial optimization*: Princeton University Press, 2003.
[47] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*: Addison-Wesley Professional, 2001.
[48] D. Whitley, "The GENITOR algorithm and selection pressure: why rank-based allocation of reproductive trials is best," in *the third International Conference on Genetic Algorithms* 1989, pp. 116-121.
[49] P. Moscato and C. Cotta, "A Modern Introduction to Memetic Algorithms " in *Handbook of Metaheuristics*. vol. 146: Springer, 2010, pp. 141-183.
[50] R. L. Haupt and S. E. Haupt, *Practical Genetic Algorithms*: Wiley-Interscience, 1998.
[51] T. Pender, *UML Bible*: Wiley, 2003.
[52] R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*: Addison-Wesley Professional, 1999.

[53] A. Arcuri and L. Briand, "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2011.

[54] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics,* vol. 25, pp. 101-132, 2000.

[55] 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.

[56] 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.

[57] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering: An Introduction*: Kluwer Academic Publishers, 2000.

[58] 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, pp. 233-244.

[59] 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, pp. 667-673.

[60] R. Feldt, R. Torkar, T. Gorschek, and W. Afzal, "Searching for Cognitively Diverse Tests: Towards Universal Test Diversity Metrics," in *Proceedings of the 1st Workshop on Search-Based Software Testing*, 2008, pp. 178-186.

[61] D. Vega, I. Schieferdecker, and G. Din, "Test Data Variance as a Test Quality Measure: Exemplified for TTCN-3," in *19th IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of SoftwareTestCom/FATES*, 2007, pp. 351-364.

[62] K. Lakhotia, M. Harman, and P. McMinn, "A multi-objective approach to search-based test data generation," in *The Genetic and Evolutionary Computation Conference (GECCO)*, 2007, pp. 1098-1105.