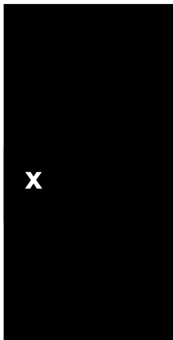


Achieving Scalable Model-Based Testing Through Test Case Diversity

H. HEMMATI[A,B], A ARCURI[A], AND L. BRIAND[A,B]
[A]Simula Research Laboratory, Lysaker, Norway
[B]Department of Informatics, Oslo, Norway

The increase in size and complexity of modern software systems requires 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 for test suites generated from state machines. 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, in the domain of embedded systems, 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.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms: Verification

Additional Key Words and Phrases: Test case selection, test case minimization, model-based testing, similarity function, search-based software engineering

Acronyms:
ATC: Abstract Test Case
CovGA: Coverage-based Genetic Algorithm
CovGrd: Coverage-based Greedy
EA: Evolutionary Algorithm
FDR: Fault Detection Rate
GA: Genetic Algorithm
MBT: Model-Based Testing
SB: State-Based
SimGrd: Similarity-based Greedy
SSGA: Steady State Genetic Algorithm
STCS: Similarity-based Test Case Selection
STGB: State-Trigger-Guard-Based
SUT: Software Under Test
TB: Transition-Based
TGB: Trigger-Guard-Based

1. INTRODUCTION

Model-based testing (MBT) [Utting and Legeard 2006] 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 [Dalal et al. 2005; Hemmati et al. 2010a; Hemmati et al. 2010b; Vaandrager 2006]. Scalability is an issue spanning all steps of the MBT procedure [Ali et

This work is a revised and extended version of papers presented at FSE 2010, ISSRE 2010, & ICTSS 2010.
Authors' addresses: Simula Research Laboratory, POBox. 134, 1325 Lysaker Norway.

al. 2010b; Utting and Legeard 2006], from handling large system models to generating and executing large test suites [Hemmati et al. 2010b]. In this paper, we focus on an important but neglected scalability aspect of MBT: Given software under test (SUT), how can one 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, especially in the context of embedded systems, 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 [Hemmati et al. 2010a] 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, which 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 [Hemmati et al. 2010b; Hemmati et al. 2011].

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 [Xu and Wunsch 2005], and applies an (1+1) Evolutionary Algorithm [Droste et al. 2002] 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 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 failure rates. The results show that the best STCS is never worse than non-STCS techniques regardless of the failure rate 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 heuristic that 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 larger very quickly 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 [Hemmati et al. 2010a] showed that such an approach does not solve the problem because (a) A coverage criterion is a means for systematically targeting specific types of faults, e.g. in UML state machine-based testing, 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 [Chen et al. 2009; Jones and Harrold 2003; Rothermel et al. 2002; Simão et al. 2006]. 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 [Orso et al. 2007], and execution traces (e.g., call stack [McMaster and Memon 2008]) 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 [Jourdan et al. 2006] and additional coverage [Jones and Harrold 2003]). 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 [Hamlet 1994], where there is no guidance to select test cases. Maximizing coverage has been a common practice over the years in selection and prioritization [Jones and Harrold 2003; Leon and Podgurski 2003]. 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 [Korel et al. 2007] 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 [Jones and Harrold 2003] uses a Greedy search to select, at every step, the test case that covers the most uncovered statements (additional coverage technique). Similarly, in [Ma et al. 2005] a Genetic Algorithm is used to

achieve maximum coverage in the selected subset of test cases. STCS is a newly introduced [Cartaxo et al. 2009; Hemmati et al. 2010a; Ledru et al. 2009] 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 underlying idea behind STCS techniques is borrowed from rewarding diversity among input data [Chen et al. 2010]. 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 [Hemmati et al. 2011], 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 [Hemmati et al. 2010b] 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 where each matrix cell represents the similarity value for a pair of encoded ATCs based on the given similarity function, and (3) minimizing similarities where an optimization algorithm selects a subset of the original ATCs with minimum sum of pair-wise similarities. Fig. 1 summarizes this process and the rest of this section explains the details of each activity in the process.

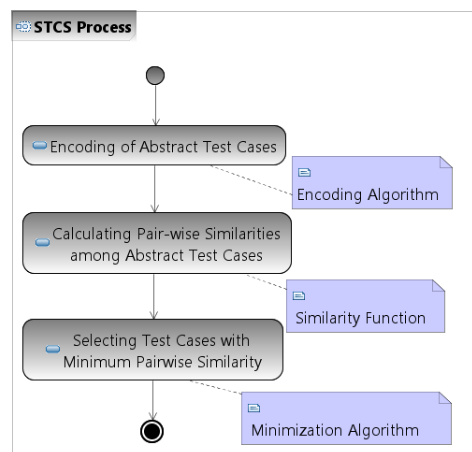


Fig 1. Similarity-based test cases selection process.

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. In UML state machine-based testing a test path represents an encoded ATC but 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 encoding focuses on state level faults by encoding the ATC using only a sequence of state ids, whereas TB can better extract relevant information to detect transition-based faults by encoding the ATCs using transition ids. In the case of TGB encoding, the focus is still on transition-based faults, but 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 TGB and TB are at a different level of abstraction since 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.

In summary, there are two main means to encode ATCs derived from a UML state machine: using state and transitions information. SB only encodes state information. TB and TGB encode only transition information, but at different levels of abstraction (low and high level of abstraction, respectively). Finally, STGB encodes both state and transition (high level of abstraction) information. In general, we want to select an encoding that will sufficiently distinguish ATC pairs but that, on the other hand, will not result in most pairs being assessed as entirely different, thus making STCS ineffective.

As an illustrative example, assume that the UML state machine in Fig. 2 represents the SUT. Applying an all-transition criterion (with a breadth-first search) on this model results in the transition tree of Fig. 3 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.

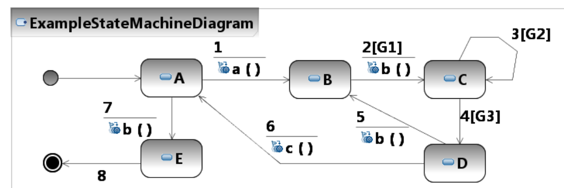


Fig 2. Example UML state machine.

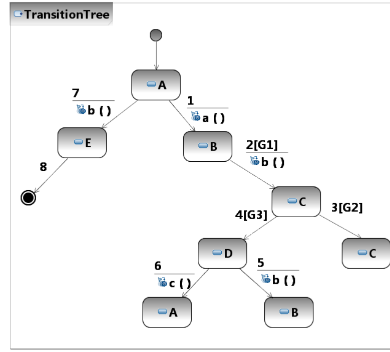


Fig 3. Example Transition Tree corresponding to the UML state machine of Fig 2.

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 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 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 [Hemmati and Briand 2010]. 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 [Hemmati and Briand 2010] 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.

4.2.1 Set-based Similarity Functions

Set-based similarity measures are widely used in data mining [Tan et al. 2006] to assess the closeness of two objects described as multidimensional feature vectors, where the set is composed of the features' values [Xu and Wunsch 2005]. 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.

4.2.1.1 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 [Dong and Pei 2007; Durbin et al. 1999; Gusfield 1997]. 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 [Dong and Pei 2007]. 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 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: Hamming(*tp3*,*tp4*)=4/5=0.8.

4.2.1.2 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, 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 Gower), and when *w*=2 this is called Sokal-Sneath or Anti-Dice (Sokal). 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 Gower/Sokal than Jaccard. 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,

Jaccard(*tp3*,*tp4*) = 3/4 = 0.75, Gower(*tp3*,*tp4*) = 6/7 = 0.86, and Sokal(*tp3*,*tp4*) = 3/5 = 0.6.

4.2.1.3 Counting Function

The Counting function is defined based on the similarity measure used in [Cartaxo et al. 2009] 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 *Counting*) as the number of identical elements in the input sets divided by the average length of inputs (in our case ATCs). *Counting* is equal to Gower in cases where all elements are unique in the input ATCs. Note that, to be precise, inputs of *Counting* 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, *Counting*(*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.

4.2.2.1 Levenshtein

One of the most well-known algorithms implementing edit-distance, and which is not limited to identical length sequences, is Levenshtein [Gusfield 1997]: 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) [Gusfield 1997]. Here we use a basic setting for the function 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 Levenshtein(*tp3,tp4*)=3.

4.2.2.2 Global alignment and Needleman-Wunsch similarity function

An alignment of two sequences is a mapping between positions of their elements [Durbin et al. 1999]. 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 [Durbin et al. 1999] (denoted as Needleman in this paper) 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 Needleman similarity function for global alignment.

Note that the only difference between Levenshtein and Needleman are the operation weights. In the case of Levenshtein, we assume the basic Levenshtein [Gusfield 1997] definition (with +1 for match and zero for mismatch and gap), and in the case of Needleman we use different operation weights as it is usual in Global alignment. The chosen weights are based on our context (UML state machine-based testing) 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 UML state machine-based testing, 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 Needleman's 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 Needleman 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, Needleman(*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 [Cormen et al. 2001] implementation of the algorithms, along with examples, can be found in [Durbin et al. 1999]. The scoring matrix F for Needleman 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.

4.2.2.3 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 (denoted as Smith in this paper) [Durbin et al. 1999], where the scoring matrix F is defined in a similar way as in the Needleman 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 Needleman for Smith with the same reasoning. As an example, Smith($tp3, tp4$)=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) [Mathur 2008]. 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 [Cartaxo et al. 2009] (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 [Arcuri 2010]. 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 [Cartaxo et al. 2009] 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.

4.3.2.1 K-Means Clustering

The first clustering algorithm used in this paper is inspired from the most popular clustering algorithm, K-Means clustering [Jain 2009]. 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 [Jain 2009]. K-Means objective is to minimize the average squared Euclidean distance of objects from their cluster means [Manning et al. 2008].

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

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. The reason is that most of the similarity measures, unlike Euclidian distance, are not transitive and violate the *triangle inequality* property [Dong and Pei 2007], 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, 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 K-Means clustering 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 K-Means clustering 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.

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

4.3.2.2 Agglomerative Hierarchical Clustering

One of the clustering algorithms which has been frequently used in software engineering, including software testing [Leon and Podgurski 2003; Masri et al. 2007; Yoo et al. 2009], is Agglomerative Hierarchical Clustering [Xu and Wunsch 2005]. This clustering algorithm 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 Agglomerative Hierarchical Clustering 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 has been proposed as an extension to Random Testing [Chen et al. 2010]. 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 [White and Cohen 1980]. Therefore, Adaptive Random Testing is a candidate similarity minimization strategy in our context as well. In this paper, we use the basic Adaptive Random Testing algorithm described in [Chen et al. 2010], but we ensure that no replicated ATC is given in output. The pseudo-code for Adaptive Random Testing 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 [Harman and Jones 2001]. 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 [Harman 2007], with a large representation from software testing [Ali et al. 2010a]. 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 [Michalewicz and Schoenauer 1996], 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 Section 4.3.4.1, 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 [Wolpert and Macready 1997]. 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 [Arcuri and Yao 2008; Harman and McMinn 2010; Xiao et al. 2007]. 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.

4.3.4.1 Random Search

Random Search 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. Random Search does not exploit any information about previously visited elements when choosing the next elements to sample. Often, Random Search is used as a baseline for evaluating the performance of other more sophisticated meta-heuristics [Ali et al. 2010a]. Note the difference between Random Search, which is a search algorithm, and Random Testing which simply selects ATCs at random without any iteration.

What distinguishes alternative Random Search 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 [Feller 1968], 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.

4.3.4.2 Hill Climbing

Hill Climbing belongs to the class of local search algorithms [Aarts and Lenstra 2003]. 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 the algorithm "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 the algorithm re-starts from a new element in the search space. Hill Climbing 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 Hill Climbing used in this paper. We use the common strategy to visit the neighborhood that makes the algorithm 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. Another more expensive strategy would be considering all possible ATCs, instead of just one at random. That would lead to a larger neighborhood of size $n*(k-n)$, since there is $k-n$ possible neighbors per ATC.

4.3.4.3 Steady State Genetic Algorithms

Genetic Algorithms (GAs) [Goldberg 2001] are inspired from evolutionary theory, and they are the most used search algorithm in search-based software engineering [Ali et al. 2010a; Harman 2007; Harman and Jones 2001]. 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 [Whitley 1989]. 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 Section 4.3.4.1).

(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 $s_n^y := \acute{s}_n^y$

4.3.4.4 (1+1) Evolutionary Algorithm

(1+1) Evolutionary Algorithm (EA) [Droste et al. 2002] 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.

4.3.4.5 Memetic Algorithms

Memetic Algorithms [Moscato and Cotta 2010] are a meta-heuristic that uses both global and local search (e.g., a GA with a Hill Climbing). 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 Memetic Algorithm 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 Memetic Algorithm is to use a GA, with the only difference that, at each generation, on each offspring a Hill Climbing 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.

4.3.4.6 Simulated Annealing

Simulated Annealing [Haupt and Haupt 1998] 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 the material 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 Simulated Annealing 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, Simulated Annealing is similar to Hill Climbing. Simulated Annealing 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 Simulated Annealing moves to it and discards the previous element. Otherwise, Simulated Annealing moves to

this new neighbor according to a probability function that is based on the current temperature. In contrast to Hill Climbing, Simulated Annealing 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. At the end of this section, Table 2 summarizes the characteristics of the 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. The SUT's implementation consists of more than three million lines of C code in 20 different subsystems. The core functionality of the system, which is modeled and tested in this case study, is one of these 20 subsystems. Unfortunately, we cannot report more details on the code of the SUT because, due to confidentiality restrictions, we do not have access to the code of the subsystems.

The system was modeled, for an already implemented and working system, through joint work between the company experts and the researchers. The researchers brought their modeling expertise, especially in the initial phases, and the domain experts provided domain specific knowledge. The researchers' involvement is just there to ensure that modeling is properly applied something that practitioners eventually get acquainted with. A three-level hierarchical state machine describes case study 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. The flattened version of the state machine (automatically generated by our UML state machine-based testing tool, TRUST [Ali et al. 2010b]) consists of 11 states and 70 transitions. Constraints specifying state invariants and guards are expressed in the Object Constraint Language (OCL) [Pender 2003] and are used to derive automated test oracles. When executing test cases these invariants should be evaluated at run-time to detect faults. 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. They are detected by ATCs either by visiting a specific state, taking a specific transition, or using a specific input data for the triggers. More specifically, these faults are detectable by traversing the state machine in the following manner:

- Fault1: It is detected if from any state a specific trigger is executed, regardless of the trigger's parameter values. The trigger corresponds to a faulty API that is called in several transitions, both in high and low level state machines.
- Fault2: It is detected if from any state a specific trigger with a specific parameter value is executed. Similar to Fault1, the faulty API exists in several transitions both in high and low level state machines, but is only detectable with one of the three possible values for an input parameter of the trigger.
- Fault3: It is detected only if in some specific states (three out of 11 states in the flattened state machine) a specific trigger is executed, regardless of the trigger's parameter values.
- Fault4: It is detected only if in some specific states (the same three states as Fault3 plus an extra three states, that is a total of six out of 11 states in the flattened state machine) a specific trigger with specific parameter values (only one third of the input data) is executed.

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.

In this case study, each test execution takes on average between one to five minutes to run depending on the number of *call*s in the test case and the network response time. For example, running 590 concrete test cases on one software version took around a day of test case execution (there was in total four software versions in this experiment). Each test case requires four video conferencing systems in a testing lab specifically assigned to this task. Though the total cost of executing 59 test cases is affordable for the company, they would like to decrease it so that the test suite size is comparable to their manually generated test suite (10 to 15 test cases). In addition, our main motivation is ultimately to apply MBT to larger scale subsystems of the company and to account for non-functional behavior, thus leading to larger models. Therefore, we have been specifically asked to provide scalable selection techniques.

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. Unlike, case study A, the model of the system was first constructed following the same approach as with case study A (joint work between our researchers and the company experts). The SUT was implemented based on the system model by the company developers using an extended version of the state design pattern [Gamma et al. 1995]. Subsequently, without intervention from researchers, developers derived new versions of the model and code to accommodate new requirements. Across these new versions, a total of 26 faults were detected. The resulting correct code for the new version of SUT consists of 26 classes and more than 3000 lines of code (without blank lines).

Instead of using the models originally devised by the developers, a correct UML state machine and class diagram, representing the latest version of the SUT, was constructed manually by the researchers to be the basis for test case generation. The SUT is described in a class diagram consisting of one control class, seven abstract classes, and 13 concrete classes. The UML state machine 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 model) [Binder 1999]. To detect such faults the model should account for the behavior of the SUT when receiving unexpected triggers. Since the focus of this work is not robustness testing, such robustness behavior is not currently modeled (though it could be by adding all scenarios involving unexpected input data and environment behavior) 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. Note that all faults are in the source code of the SUT. They can either be an implementation error or modeling error (misunderstanding of requirements), which has been reflected in the source code. However, in both cases if we model the faulty implemented SUT, we can relate each code-level fault to one ofthe following modelling errors:

- Incorrect guards (10 faults).
- Missing on-entry behavior (two faults).

- Missing transitions: An expected guarded transition, triggered by a completion event, was missing from the model (one fault).
- Erroneous on-entry behavior: An operation that handles the on-entry behavior of a super state was introduced with an error (one fault).
- Incorrect state invariants (one fault).

We do not report the exact timings in this case study due to confidentiality. In general, testing on the actual hardware of the company is quite expensive due to the set-up and access to sensors, actuators, and other hardware resources, a situation that is quite typical in embedded systems.

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 these 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— 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. In case study B, 74 out of 281 ATCs detect at least one fault thus yielding failure rate= 74/281≅0.26. In case study A, since 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, failure rate is $(\sum_{i=1}^{59} P_i)/59$ = 43.28/59≅0.73.

Notice that in these case studies failure rate is high. Since MBT is based on systematic test strategies it should be expected, as existing studies show [Utting and Legeard 2006], to lead to a higher failure rate than random testing. Though there is no proof for such a claim this is after all one main objective of MBT. In addition, in case study A, the very high failure rate is a direct consequence of the fault types. 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 characteristics of the case studies.

Note that in both case studies, after executing concrete test cases and detecting faults, we map each test case to detected faults in a fault table. Each row of the table represents an ATC and each column a detectable fault. Each entry in the table shows the probability of detecting the corresponding fault by the ATC. To evaluate the effectiveness of each test selection technique, this table is later used to calculate the FDR of a given subset of ATCs.

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 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:

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

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.

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

Since choosing a specific STCS strategy requires setting three parameters out of a large possible number of combinations, one needs clear guidance to choose the best combination possible in a given context.

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 [Hamlet 1994]—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 UML state machine-based testing) 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 the failure rate in both of our case studies is quite high, it is important to apply STCS in SUTs with lower failure rate before generalizing the results. Therefore, in this question we specifically address the effect of the failure rate 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 Experiment1: Answering RQ1(How influential are STCS parameters on its effectiveness?) and RQ2 (What is the most cost-effective STCS variant?)

With RQ1 we want to understand whether all three parameters matter, which ones have a significant effect, and why it is so. With RQ2 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. Therefore, 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: Levenshtein, Needleman, Smith, *Counting*, Hamming, Jaccard, Sokal, Gower; and 10 minimization algorithms: SimGrd, K-Means Clustering, Agglomerative Hierarchical Clustering, Adaptive Random Testing, Random Search, Hill Climbing, SSGA, (1+1)EA, Memetic Algorithm, Simulated Annealing).

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, K-Means clustering, and Agglomerative Hierarchical Clustering involve no parameters. For Adaptive Random Testing, we used k=10 (as suggested in [Chen et al. 2010]). For search-based techniques, we made choices based on what is suggested in the literature [Goldberg 2001] and our previous experience (e.g., [Hemmati et al. 2010b; Hemmati et al. 2011; Hemmati and Briand 2010; Hemmati, Briand, Arcuri and Ali 2010(a)]). We use a high crossover probability (0.75) for GA and Memetic Algorithm. Mutation probability is the standard one: $1/n$, for SSGA, Memetic Algorithm, and (1+1)EA. Population size is set to 50 for SSGA and 10 for Memetic Algorithm. The rank selection for SSGA and Memetic Algorithm uses a 1.5 bias. For Simulated Annealing, 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 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 [Arcuri and Briand 2011]. 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 [Arcuri and Briand 2011; Vargha and Delaney 2000] 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 [Arcuri and Briand 2011]. 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. In the rest of this paper we simply use the phrase "effect size" wherever we want to refer to the *A* statistic.

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 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, Fig. 4 to Fig. 6 show the boxplots of the variant ranks when considering one parameter at a time. For example, Fig. 4.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). Fig. 4.b and Fig. 4.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,

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

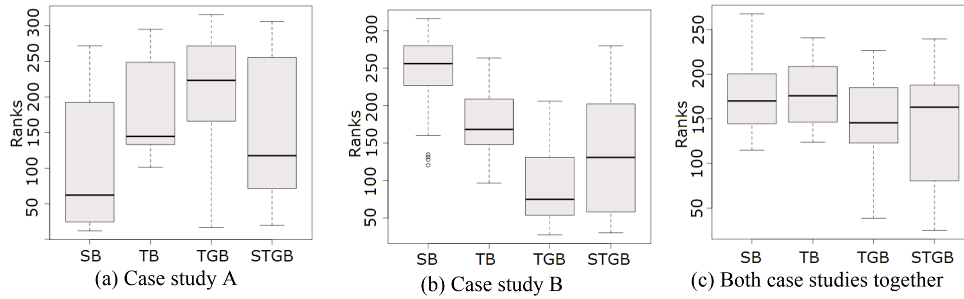
Parameters	**Case study A**		**Case study B**		**Both case studies together**	
	p-value	**Chi-squared**	**p-value**	**Chi-squared**	**p-value**	**Chi-squared**
Encoding	1.344e-14	67.67	< 2.2e-16	170.64	1.874e-5	24.60
Similarity function	< 2.2e-16	161.62	0.8349	3.50	< 2.2e-16	159.64
Minimization algorithm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

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 shows the first top 10 rows out of 320 rows of each rank table.

Looking at each case study individually shows that Jaccard, Sokal, and Gower are the best similarity functions and that (1+1)EA, Memetic Algorithm, and Simulated Annealing 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, Adaptive Random Testing, 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.

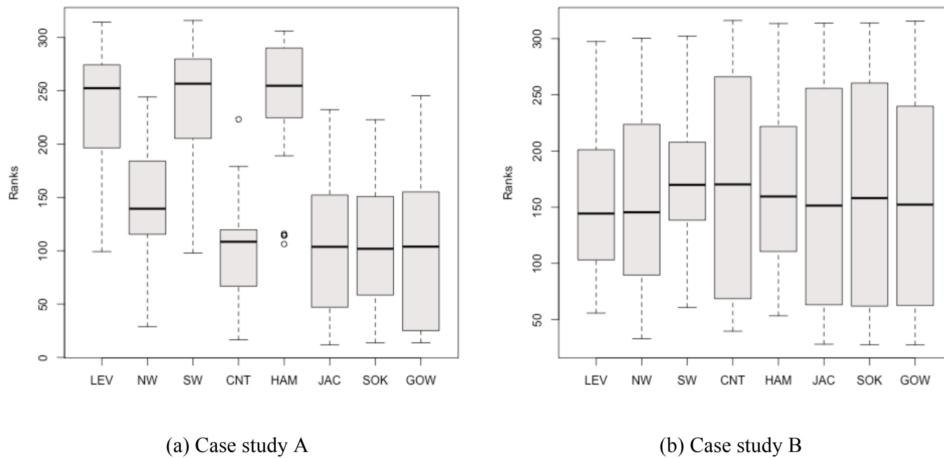
To draw a conclusion based on both case studies together, we need to look at the rank of the best STCS variants in the section on both case studies together of Table 4, where the average ranks over the two case studies are computed. STGB appears to be the best encoding—most probably because it contains 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 Jaccard, Sokal and Gower, which are the best similarity functions in the case studies, Gower shows the best results when averaging the ranks. That implies that assigning more weight to similarities than differences—that is what differentiates Gower compared to Jaccard and Sokal—seems more effective, though differences are relatively minor.

Regarding minimization algorithms, we already knew that search-based techniques like SSGA are more effective than Adaptive Random Testing and clustering-based techniques, such as Agglomerative Hierarchical Clustering [Hemmati et al. 2010b]. Looking at the results of each study individually confirms that finding. However, a more

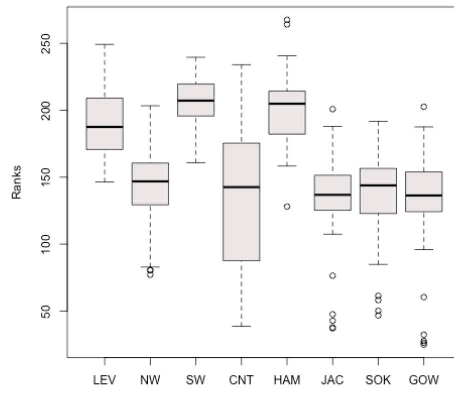


(a) Case study A (b) Case study B (c) Both case studies together

Fig 4. The effect of encoding on FDR of STCS. Each boxplot shows the rank (generated from statistical tests on FDRs) of 80 observations per encoding.

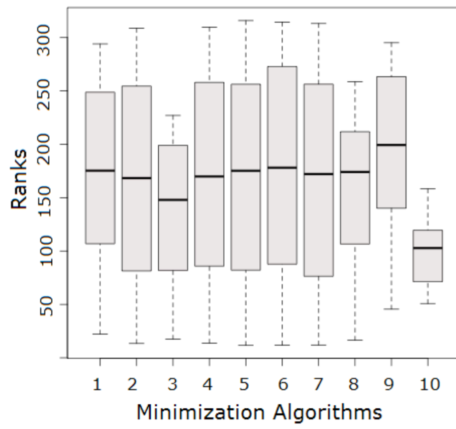


(a) Case study A (b) Case study B

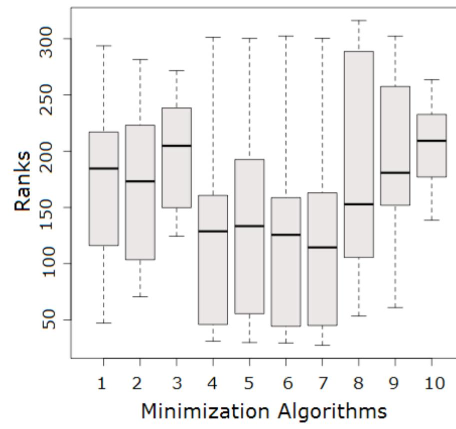


(c) Both case studies together

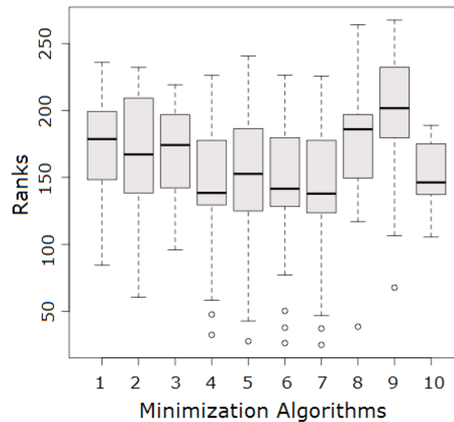
Fig 5. The effect of similarity function on FDR of STCS. Each boxplot shows the rank (generated from statistical tests on FDRs) of 40 observations per similarity function.



(a) Case study A



(b) Case study B



(c) Both case studies together

Fig 6. The effect of minimization algorithm (1: Adaptive Random Testing, 2:SSGA, 3: Random Search, 4: Hill Climbing, 5: Simulated Annealing, 6: Memetic Algorithm, 7:(1+1)EA, 8:Greedy, 9: Agglomerative Hierarchical Clustering, 10: K-Means Clustering) on FDR of STCS. Each boxplot shows the rank (generated from statistical tests on FDRs) of 32 observations per minimization algorithm.

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 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 adopting 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

Table 4. Top 10 STCS variants (the lower rank the better the technique), generated from statistical tests on FDRs, for case study A, B, and both case studies together.

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

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

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

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 Adaptive Random Testing, 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 Adaptive Random Testing, the candidate size is 10 (resulting in $10*n*(n-1)/2$ similarity comparisons). Therefore, we run Adaptive Random Testing 1000 times and select the best output set among those runs 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, K-Means clustering, and Agglomerative Hierarchical Clustering. Therefore, we have to rely on execution time even though this is an imperfect measure of cost [Ali et al. 2010a]. What makes comparisons simpler however is that Agglomerative Hierarchical Clustering and K-Means clustering, 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 [Cartaxo et al. 2009] 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 Fig. 6 (its median is very low and there is a high variation in its results, especially visible in case study B) and it only appears once out of 30 among the best variants in Table 4; (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, Gower, 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 Experiment2: Answering RQ3(What is the practical benefit of using STCS?)

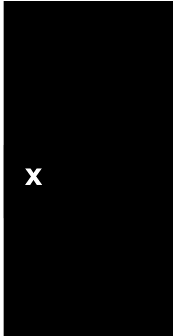
In Experiment 2, we are interested in evaluating STSC cost and effectiveness compared to state of the art, non-STCS test case selection strategies. The goal is to assess the magnitude of improvement that STSC provides, which can be measured both in terms of

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. Random Testing 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.

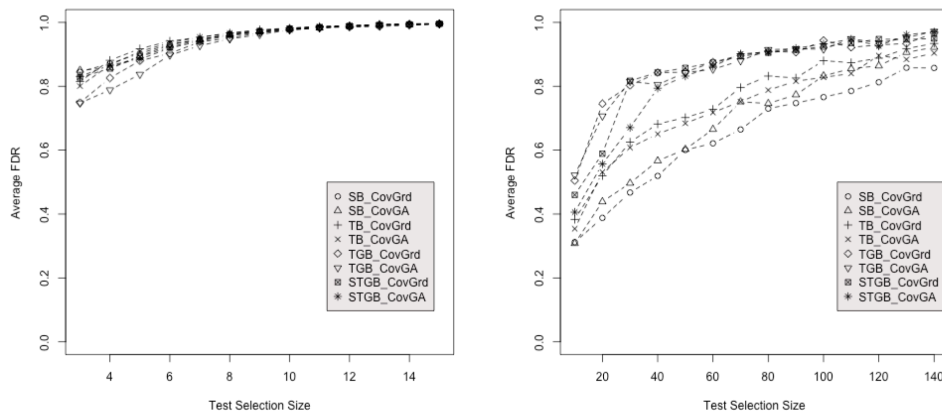
In a model-based selection context, coverage can only be defined based on the ATCs, since no execution or source code information is available. The most well-known UML state machine-based testing coverage criterion, which is used for test case selection in the literature, is transition coverage [Korel et al. 2007]. 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 [Elbaum et al. 2002; Korel et al. 2007] and GA [Li et al. 2007; Ma et al. 2005]. In the Greedy-based technique (CovGrd), which is inspired by the additional coverage technique [Jones and Harrold 2003], 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. Fig. 7 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 Fig. 7.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 (Fig. 7.a) are practically negligible because of the high failure rate. 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 Random Testing, Fig. 8 reports their mean FDRs over the range of test selection sizes in case study A (Fig. 8.a) and case study B (Fig. 8.b). We also reported the effect size for comparing Best_STCS vs STGB_CovGrd and Best_STCS vs. Random Testing, respectively, in Fig. 9.a (for case study A) and Fig. 9.b (for case study B). The effect size shows the probability that a selected test set by Best_STCS (randomly taken from the 100 runs) will have higher FDR

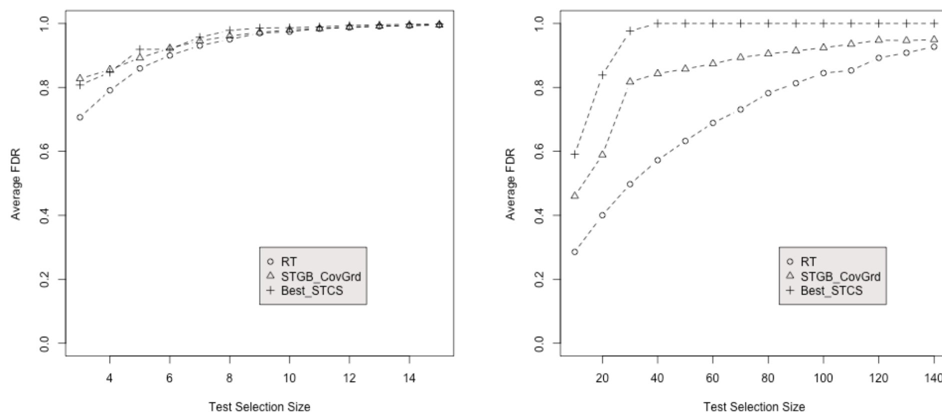


than a selected test set by STGB_CovGrd and Random Testing (randomly taken from the 100 runs), respectively.



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

Fig 7. The FDR comparison of different coverage-based test case selection techniques for different test selection sizes.



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

Fig 8. The FDR comparison between the best STCS (Best_STCS) and non-STCS technique, STGB_CovGrd and Random Testing (RT), for different test selection sizes.

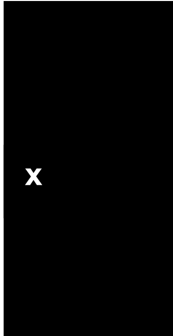
Comparing Best_STCS, STGB_CovGrd, and Random Testing, Fig. 8 reports their mean FDRs over the range of test selection sizes in case study A (Fig. 8.a) and case study B (Fig. 8.b). We also reported the effect size for comparing Best_STCS vs STGB_CovGrd and Best_STCS vs. Random Testing, respectively, in Fig. 9.a (for case study A) and Fig. 9.b (for case study B). The effect size shows 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 Random Testing (randomly taken from the 100 runs), respectively.

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 Random Testing. Looking at the percentage of mean FDR improvement, especially for smaller test selection sizes, Best_STCS provides between 15% to 45% improvement over STGB_CovGrd and 80% to 110% over Random Testing. 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 Random Testing. All *p*-values are very low (zero or close to zero). This phenomenon is also visible in Fig. 9.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.

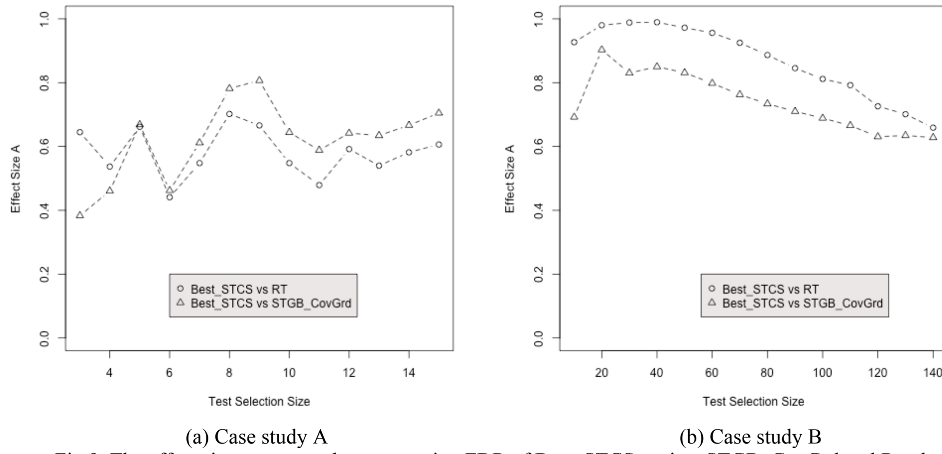
However, the improvements are not practically significant in case study A. The differences in mean FDRs (from Fig. 8.a) are small. In some cases there is not even a statistically significant difference between them and the effect size also does not show very high values. The small differences among techniques have been also observed in Fig. 7 when comparing coverage-based techniques. The most plausible explanation is the high failure rate 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 Fig. 10.a (Best_STCS vs. Random Testing) and Fig. 10.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 Random Testing 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 Random Testing.

Cost is also an important factor in selecting the best technique. The simple algorithms used in CovGrd and Random Testing definitely result in very low execution time. Random Testing 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

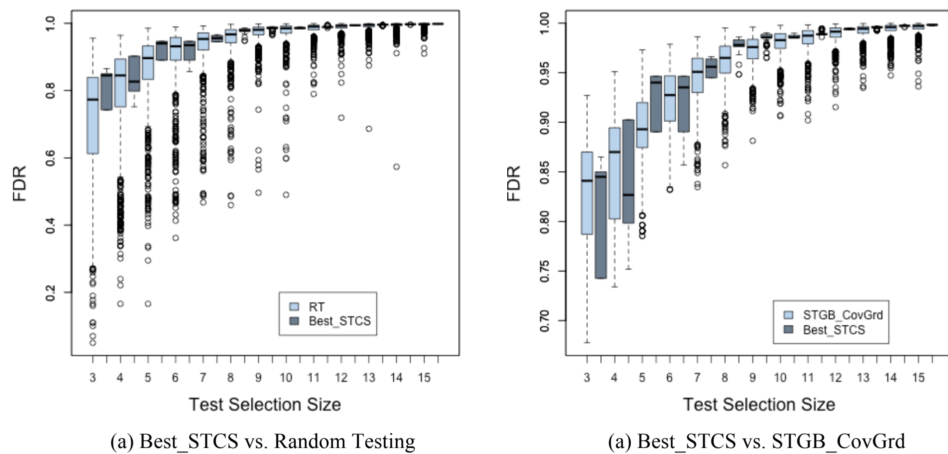


hundreds of milliseconds, any reduction in the number of test cases is much more effective in reducing the total cost than saving milliseconds during selection. We will discuss more about the cost of the techniques for larger systems in Section 5.4.



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

Fig 9. The effect size measure when comparing FDR of Best_STCS against STGB_CovGrd and Random Testing (RT) for different test selection sizes.



(a) Best_STCS vs. Random Testing (a) Best_STCS vs. STGB_CovGrd

Fig 10. Distribution of FDRs over 100 runs for different test selection sizes of Best_STCS, STGB_CovGrd, and Random Testing (RT) as boxplots on case study A.

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 Random Testing). 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 Random Testing, respectively). Even for case study A, where the failure rate was high, the

Best_STCS never performed worse than the baselines (in terms of effect size), except for very small test selection sizes of three, four, and six ATCs (Fig. 9.a), where in those cases Best_STCS shows less variance in results (Fig. 10). 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 Experiment3: Answering RQ4(What is the effect of the failure rate on the effectiveness of STCS?)

In Experiment 3, we simulate test suites with different failure rates 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 failure rate is low. To run such experiment, we need test suites with lower failure rates than that of our current case studies. Therefore, we take case study B, which has the lowest failure rate (26%) to start with, and use it to form 25 types of test suites with equal size (200), but with different failure rates 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. The rationale for this simulation strategy is to keep relying on actual ATCs and failures but to reduce the proportion of failing ATCs. As a result, we do not generate artificial ATCs or change the fault detection pattern of ATCs (which faults are detected by which ATCs). Notice that, regardless of the matrix generation procedure, we cannot build a fault detecting matrix with a failure rate lower than 1/(281-73)=0.48% from case study B. There can be many test suites with size 200 and a given failure rate based on case study B. Therefore, we generate 30 different test suites per failure rate and apply Best_STCS, STGB_CovGrd, and Random Testing 100 times for different test selection sizes on each test suite.

Fig. 11.a and 11.b show three-dimension graphs to visualize variations in effect size measure when comparing the FDR of ATCs selected by Best_STCS vs. Random Testing and Best_STCS vs. STGB_CovGrd, respectively. Two parameters are varied to explain variations in effect size values: failure rate 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 effect size of the FDRs for each value of failure rate and test selection size (there are 3000 data points for each effect size value: 100 runs and 30 test suites), show that Best_STCS is never worse than its alternatives (effect size ≥ 0.5) regardless of test selection size and failure rate. This answers RQ4. If we carefully analyze the effect size change over different test selection sizes for different failure rates, we can see that the trend is the same as what has been seen in Fig. 9.b (the original case study B with failure rate=26%). There is a test selection size value where A is maximum (e.g., test selection size 20 in Fig. 9.b and Fig. 11.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 the failure rate is large. A higher failure rate provides the better 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

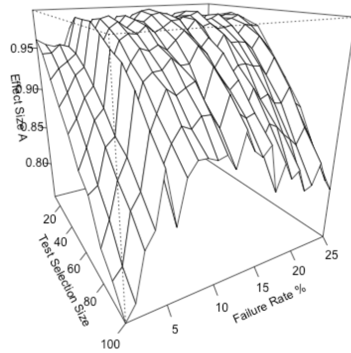
technique will result in the same FDR as Random Testing. In summary, the results of Experiment 3 suggest that Best_STCS dominates its alternatives and that it is not an artifact of the high failure rate of our case studies. It also confirms that Best_STCS may not be significantly better when the original test suite's failure rate 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 failure rate. On the other side of the range, extremely high failure rates are unlikely on realistic systems.

5.3.4 Experiment 4: Answering RQ5 (How one can estimate the minimum number of test cases required for achieving (near) maximum FDR?)

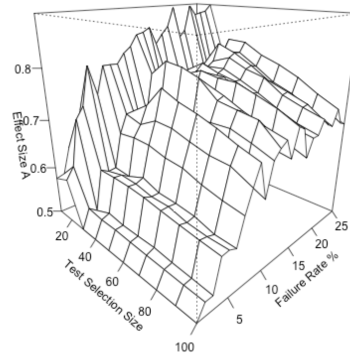
Though in practice the test budget may be imposed by external constraints, the tester may have some degree of freedom to increase it if it is believed to yield significant benefits in terms of fault detection. We are therefore interested in finding a heuristic that helps the tester choose an optimal test selection size for STCS. Our heuristic is based on the average similarity of ATC pairs in the set ($SimMsr(s_n)/n$), denoted as Sim. If there is a linear correlation between Sim 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 Sim..

In Experiment 4, we re-apply Best_STCS on both case studies and calculate the normalized Sim per 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 Sim.. 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 Sim is equal to average similarity of ATC pairs in the entire test suite (Sim_{TS}) and the normalized similarity measure is:

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



(a) Best_STCS vs. Random Testing



(b) Best_STCS vs. STGB_CovGrd

Fig 11. The effect size measure when comparing FDR of Best_STCS against STGB_CovGrd and Random Testing for different test selection sizes (10 to 100) and failure rates (1% to 25%).

Fig. 12 shows the average FDR(s_n) (y-axis) as a function of $Norm(s_n)$ (x-axis) for case study A (Fig. 12.a) and case study B (Fig. 12.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.

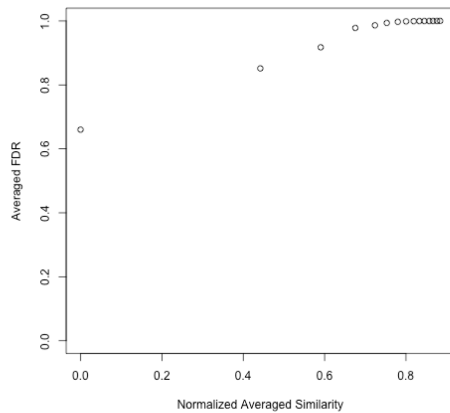
However, what is more interesting is the near-linear relationship between *Norm* and FDR, before reaching the plateau. We performed a least squares regression to linearly fit Averaged FDR to Normalized Averaged Similarity. Fig. 13.a and Fig. 13.b show the fitted lines and the 95% confidence intervals corresponding to the predicted line and the observations. Note that the regression fit is very high in both cases, thus suggesting a near linear relationship: R^2=0.94 for case study A and 0.91 for case study B.

In practice, while considering increases in test selection size, one can look at the trend of *Norm* (or *Sim*) and choose to increase the test selection size only when it triggers a significant increase in *Sim* afterwards. Because of the observed near-linear 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 failure rate 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.

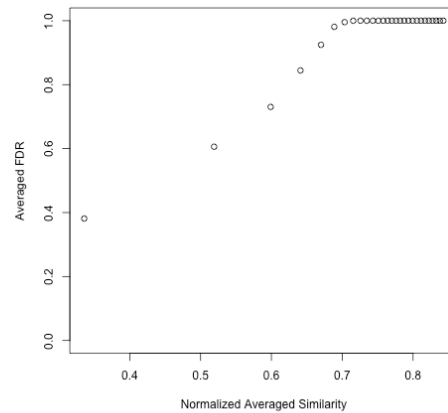
Fig. 14 shows the average FDRs along with *Sim* and *Norm*, for different test selection sizes. We can see the same trend in both studies (Fig. 14.a for case study A and Fig. 14.b for case study B): the *elbow point* (when the last significant decrease in the slope of the tangent line appears) in the *Norm* or *Sim* 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 Fig. 12.a and 12.b. Table 5 (case study A) and Table 6 (case study B) reports the average FDR and *Norm* for different test selection sizes close to the *elbow points* of the curves in Fig 14.

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 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

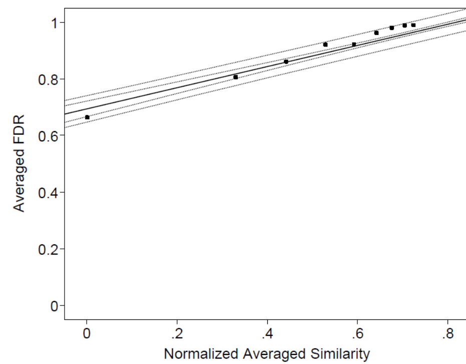


(a) Case study A with test selection sizes between 2 to 30 (50% of the test suite), intervals of two.

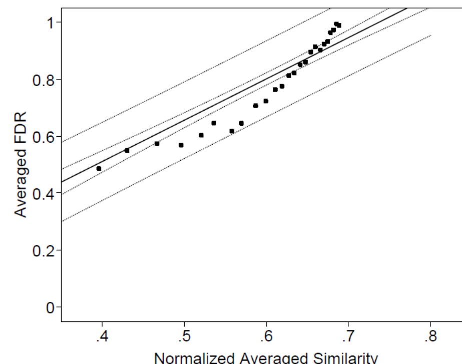


(b) Case study B with test selection sizes between 5 to 140 (50% of the test suite), intervals of five.

Fig 12. 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



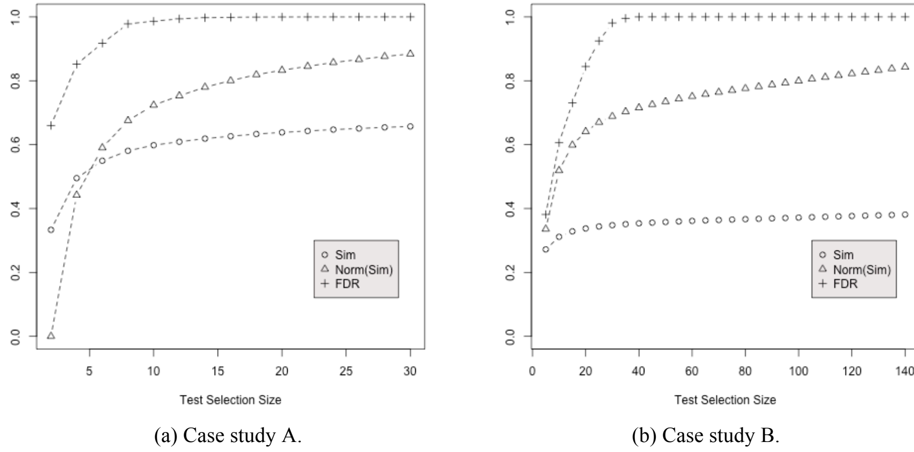
(b) Case study B

Fig 13. Regression line and confidence intervals fitted to data for all test selection sizes between 2 to 10 in case study A and between 5 to 30 in case study B

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.



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

Fig 14. Average FDR, similarity measure, and the normalized similarity measure for different test selection sizes using Best_STCS.

Table 5. 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 6. 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**

In general, the time complexity of set 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., Needleman takes in average three seconds for a 281*281 similarity matrix—(281*280)/2=39,340 similarity value calculations—in case study B, but Gower only requires half a second, on a PC with Intel Core2 Duo CPU 2.40 GHz). Therefore, similarity matrix generation in Best_STCS also scales up fairly well, linearly with ATC length and polynomial with test suite size ($O(|tp|*|TS|^2)$). 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 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. 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. Note that, in theory, the test case selection cost grows quadratically while the test suite execution cost grows linearly. Therefore, there should be a threshold after which the selection process is more costly than the entire test suite execution. However, by looking at the abovementioned selection costs for different test selection sizes, we can see, for example, that increasing the size of the problem (both test suite size and the test selection size) more than 40 times adds only 85 extra seconds for selection and saves (12000-850)* $Cost_{tc}$. Assuming $Cost_{tc}$ on average is five minutes, then the saving would be almost 40 days of non-stop test execution. Estimating that threshold is not feasible without having many large scale case studies, but exceeding the threshold would likely happen only by selecting a very large number of test cases from extremely large test suites. Therefore, in summary, we found Best_STCS to be a practically scalable selection technique, which is applicable in large systems with reasonable cost while retaining its high effectiveness.

5.5 Discussion of Validity Threats

[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.

In this subsection, we discuss the potential threats to the validity of the study using the framework introduced in [Wohlin et al. 2000] 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 [Ali et al. 2010a] 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., Needleman). 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 (Adaptive Random Testing 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. Note that it is in theory possible to apply a large scale study for tuning such parameters (e.g., using machine learning on different benchmark systems). In that case, one may be able to find a technique with a more optimal tuning than our suggestions in this paper (for similarity function and minimization algorithm). Though tuning is a problem that is common to all empirical analyses involving search algorithms, this nevertheless remains as an internal validity threat.

When we generated the simulated similarity matrices with different failure rates 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 failure rate 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 failure rate. 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. However the conclusion would be stronger if we had more case studies so that we could apply some statistical tests instead of simply averaging two cases' results. To compare selection techniques and assess the magnitude of the differences in cost and FDR, we used an effect size measure in addition to showing boxplots. The conclusion on the scalability of the selection process presented in Section 5.4 is not supported by rigorous statistical analysis due to the lack of data for extremely large case studies and remains a threat to the validity of the results. In addition, all conclusions on the effectiveness of STCS are based on 15+4 real faults in our two case studies and investigating more cases studies with different types of faults would help reducing this threat to the validity. Nevertheless, as software testing results on proprietary industrial systems are scarce in the literature, the empirical analysis reported in this paper still provides an important contribution.

***External validity**:* Our results rely on two proprietary 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 failure rates and different sizes). However, replicating our studies in various domains as many times as possible is of course required to gain higher confidence in our results and better understand their limitations. Despite the fact that one can never be sure whether case study results can be generalized to other systems, we have carefully tried to qualitatively explain our results, thus contributing to understand how they might be generalized.

Note that, in practice, modeling always involves people with modeling skills and application domain expertise, who are sometimes the same persons. Though in these studies we played the former role (at least in the initial stages) and, as a result, one might question the industrial representativeness of our cases, the important thing for our study is that we competently model actual commercial systems, at an adequate level of abstraction (determined by available domain expertise) and with actual faults. Though the limited number of faults can be a potential threat to external validity of the results, this is the price to pay for using, in practice, actual deployed faults. On both case studies, to maximize the number of faults, fault reports were collected across several releases and we made sure they were all applicable for the most recent release. Despite the relatively small number of faults, because of the way we designed and analyzed our experiments, we could nevertheless provide statistically valid conclusions. Since software testing results on proprietary industrial systems are scarce in the literature, the empirical analysis

reported in this paper still provides an important contribution despite its limitations in terms of external validity.

6. RELATED WORK

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 [Simão et al. 2006] the similarity function is based on all def-use pair 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 [Jiang et al. 2009; Leon and Podgurski 2003; Masri et al. 2007; Yoo et al. 2009]. Greedy search, adaptive random selection, and clustering are used in these studies for selection/prioritization. In [Ramanathan et al. 2008] 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.* [Feldt et al. 2008] 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.* [Ledru et al. 2009] 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. Adaptive Random Testing, as introduced in Section 4.3.3, for example, is one of the most well-known techniques in this group. Vega *et al.* [Vega et al. 2007] 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 [Cartaxo et al. 2009], where sequences of transitions in a Labeled Transition System model of the SUT are used to represent test paths. The similarity function is our *Counting*, as defined in Section 4.2.1.3, and the selection technique is a Greedy search, that is SimGrd in Section 4.3.1. If we tailor their approach for UML state machine-based testing, 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 that as an STCS baseline of comparison, Best_STCS is on average the best variant (18th best variant for case study A and 6th best for the case study B) and is a much better choice than their variant, which is ranked 127th for case study A, 111th for case study B, and 66th on average.

This study is an extension of the work we reported in [Hemmati et al. 2010b; Hemmati and Briand 2010; Hemmati et al. 2010a]. The general idea of STCS is introduced in [Hemmati et al. 2010a] and SB, TB, and TGB encodings are compared while using a *Counting* similarity function and a SSGA/SimGrd algorithm on case study B. We also compared the variant identified by TGB, *Counting*, and SSGA with coverage-based selection and random testing. In [Hemmati and Briand 2010] we have focused on comparison between six similarity functions (all similarity functions introduced in this paper except Gower and Sokal) on the same case study. In [Hemmati et al. 2010b], we further evaluated the approach proposed in [Hemmati et al. 2010a] when replacing SSGA with Agglomerative Hierarchical Clustering and Adaptive Random Testing. 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 [Hemmati et al. 2011], 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 alleviate the impact of outliers 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, especially in the context of embedded systems. 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 test case selection techniques for test suites generated from state machines, 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 [Lakhotia et al. 2007] 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. Going beyond state machine-based testing and applying test case selection for test suites generated from any form of model, or even randomly generated test suites, is also another potential road ahead.

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

AARTS, E.H.L. AND LENSTRA, J.K. 2003. *Local search in combinatorial optimization*. Princeton University Press.

ALI, S., BRIAND, L.C., HEMMATI, H. AND PANESAR-WALAWEGE, R.K. 2010a. A Systematic Review of the Application and Empirical Investigation of Search-based Test-Case Generation. *IEEE Transactions on Software Engineering 36*, 742-762.

ALI, S., HEMMATI, H., HOLT, N.E., ARISHOLM, E. AND BRIAND, L. 2010b. Model Transformations as a Strategy to Automate Model-Based Testing - A Tool and Industrial Case Studies Simula Research Laboratory, Technical Report(2010-01).

ARCURI, A. 2010. Longer is Better: On the Role of Test Sequence Length in Software Testing. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*2010, 469 - 478

ARCURI, A. AND BRIAND, L. 2011. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*2011.

ARCURI, A. AND YAO, X. 2008. Search Based Software Testing of Object-Oriented Containers. *Information Sciences 178*, 3075-3095.

BINDER, R.V. 1999. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Professional.

CARTAXO, E.G., MACHADO, P.D.L. AND NETO, F.G.O. 2009. 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*.

CHEN, T.Y., KUOA, F.-C., MERKELA, R.G. AND TSEB, T.H. 2010. Adaptive Random Testing: The ART of test case diversity. *Journal of Systems and Software 83*, 60-66.

CHEN, Y., PROBERT, R.L. AND URAL, H. 2009. Regression test suite reduction based on SDL models of system requirements. *Journal of Software Maintenance and Evolution: Research and Practice 21*, 379-405.

CORMEN, T.H., LEISERSON, C.E., RIVEST, R.L. AND STEIN, C. 2001. *Introduction to Algorithms*. The MIT Press.

DALAL, S., JAIN, A. AND POORE, J. 2005. Workshop on Advances in Model-Based Software Testing. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, 680.

DONG, G. AND PEI, J. 2007. *Sequence Data Mining*. springer.

DROSTE, S., JANSEN, T. AND WEGENER, I. 2002. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science 276*, 51-81.

DURBIN, R., EDDY, S.R., KROGH, A. AND MITCHISON, G. 1999. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press.

ELBAUM, S.G., MALISHEVSKY, A.G. AND ROTHERMEL, G. 2002. Test Case Prioritization: A Family of Empirical Studies. *IEEE Transactions on Software Engineering 28*, 159-182.

FELDT, R., TORKAR, R., GORSCHEK, T. AND AFZAL, W. 2008. Searching for Cognitively Diverse Tests: Towards Universal Test Diversity Metrics. In *Proceedings of the Proceedings of the 1st Workshop on Search-Based Software Testing*2008, 178-186.

FELLER, W. 1968. *An Introduction to Probability Theory and Its Applications*. Wiley.

GAMMA, E., HELM, R., JOHNSON, R., and VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

GOLDBERG, D.E. 2001. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional.
GUSFIELD, D. 1997. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press.
HAMLET, R. 1994. Random testing. In *Encyclopedia of Software Engineering* Wiley, 970-978.
HARMAN, M. 2007. The Current State and Future of Search Based Software Engineering. In *Proceedings of the Future of Software Engineering*2007 IEEE Computer Society, 342-357.
HARMAN, M. AND JONES, B.F. 2001. Search-based software engineering. *Information and Software Technology 43*, 833-839.
HARMAN, M. AND MCMINN, P. 2010. A theoretical and empirical study of search based testing: Local, global and hybrid search. *IEEE Transactions on Software Engineering (TSE) 36*, 226-247.
HAUPT, R.L. AND HAUPT, S.E. 1998. *Practical Genetic Algorithms*. Wiley-Interscience.
HEMMATI, H., ARCURI, A. AND BRIAND, L. 2010b. Reducing the Cost of Model-Based Testing through Test Case Diversity. In *Proceedings of the 22nd IFIP International conference on Testing Software and Systems (ICTSS), formerly TestCom/FATES*2010(b) LNCS 6435, 63-78.
HEMMATI, H., ARCURI, A. AND BRIAND, L. 2011. Empirical Investigation of the Effects of Test Suite Properties on Similarity-Based Test Case Selection, . In *Proceedings of the 4th International Conference on Software Testing, Verification and Validation (ICST)*2011.
HEMMATI, H. AND BRIAND, L. 2010. An Industrial Investigation of Similarity Measures for Model-Based Test Case Selection. In *Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering (ISSRE)*2010, 141-150.
HEMMATI, H., BRIAND, L., ARCURI, A. AND ALI, S. 2010a. An Enhanced Test Case Selection Approach for Model-Based Testing: An Industrial Case Study. In *Proceedings of the 18th ACM International Symposium on Foundations of Software Engineering (FSE)*2010(a), 267-276.
JAIN, A.K. 2009. Data Clustering: 50 Years Beyond K-means. *in press with Pattern Recognition Letters*.
JIANG, B., ZHANG, Z., CHAN, W.K. AND TSE, T.H. 2009. Adaptive random test case prioritization. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*2009, 233-244.
JONES, J.A. AND HARROLD, M.J. 2003. Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. *IEEE Transactions on Software Engineering 29*, 195-209.
JOURDAN, G.-V., RITTHIRUANGDECH, P. AND URAL, H. 2006. Test Suite Reduction Based on Dependence Analysis. In *Computer and Information Sciences – ISCIS 2006* Springer, 1021-1030.
KOREL, B., KOUTSOGIANNAKIS, G. AND TAHAT, L.H. 2007. Model-Based Test Prioritization Heuristic Methods and Their Evaluation. In *Proceedings of the 3rd Workshop on Advances in Model Based Testing, A-MOST*2007, 34-43.
LAKHOTIA, K., HARMAN, M. AND MCMINN, P. 2007. A multi-objective approach to search-based test data generation. In *Proceedings of the The Genetic and Evolutionary Computation Conference (GECCO)*2007, 1098-1105.
LEDRU, Y., PETRENKO, A. AND BORODAY, S. 2009. Using String Distances for Test Case Prioritisation. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE)*2009, 510-514.
LEON, D. AND PODGURSKI, A. 2003. A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases. In *Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering (ISSRE)*2003, 442-456.
LI, Z., HARMAN, M. AND HIERONS, R.M. 2007. Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on Software Engineering 33*, 225-237.
MA, X.Y., SHENG, B.K. AND YE, C.Q. 2005. Test-Suite Reduction Using Genetic Algorithm. In *Advanced Parallel Processing Technologies* Springer Berlin / Heidelberg, 253-262.
MANNING, C.D., RAGHAVAN, P. AND SCHÜTZE, H. 2008. *Introduction to Information Retrieval*. Cambridge University Press.
MASRI, W., PODGURSKI, A. AND LEON, D. 2007. An Empirical Study of Test Case Filtering Techniques Based on Exercising Information Flows. *IEEE Transactions on Software Engineering 33*, 454-477.
MATHUR, A.P. 2008. *Foundations of Software Testing*. Addison-Wesley Professional.
MCMASTER, S. AND MEMON, A. 2008. Call-Stack Coverage for GUI Test Suite Reduction. *IEEE Transactions on Software Engineering 34*, 99-115.
MICHALEWICZ, Z. AND SCHOENAUER, M. 1996. Evolutionary Algorithms for Constrained Parameter Optimization Problems. *Evolutionary Computation 4*, 1-32.
MOSCATO, P. AND COTTA, C. 2010. A Modern Introduction to Memetic Algorithms In *Handbook of Metaheuristics* Springer, 141-183.

ORSO, A., DO, H., ROTHERMEL, G., HARROLD, M.J. AND ROSENBLUM, D.S. 2007. Using component metadata to regression test component-based software. *Software Testing, Verification and Reliability 17*, 61-94.
PENDER, T. 2003. *UML Bible*. Wiley.
RAMANATHAN, M.K., KOYUTÜRK, M., GRAMA, A. AND JAGANNATHAN, S. 2008. PHALANX: a graph-theoretic framework for test case prioritization. In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing*2008, 667-673.
ROTHERMEL, G., HARROLD, M.J., RONNE, J.V. AND HONG, C. 2002. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability 12*, 219-249.
SIMÃO, A.D.S., MELLO, R.F.D. AND SENGER, L.J. 2006. A Technique to Reduce the Test Case Suites for Regression Testing Based on a Self-Organizing Neural Network Architecture. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC)*2006, 93 - 96.
TAN, P.N., STEINBACH, M. AND KUMAR, V. 2006. *Introduction to Data Mining*. Addison Wesley.
UTTING, M. AND LEGEARD, B. 2006. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann.
VAANDRAGER, F.W. 2006. Does it Pay Off? Model-Based Verification and Validation of Embedded Systems! In *Technical Report ICIS-R06019, ICIS* Radboud University Nijmegen, 43-66.
VARGHA, A. AND DELANEY, H.D. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics 25*, 101-132.
VEGA, D., SCHIEFERDECKER, I. AND DIN, G. 2007. Test Data Variance as a Test Quality Measure: Exemplified for TTCN-3. In *Proceedings of the 19th IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of SoftwareTestCom/FATES*2007, 351-364.
WHITE, L.J. AND COHEN, E.I. 1980. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering (TSE) 6*, 247-257.
WHITLEY, D. 1989. The GENITOR algorithm and selection pressure: why rank-based allocation of reproductive trials is best. In *Proceedings of the the third International Conference on Genetic Algorithms* 1989, 116-121.
WOHLIN, C., RUNESON, P., HOST, M., OHLSSON, M.C., REGNELL, B. AND WESSLEN, A. 2000. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers.
WOLPERT, D. AND MACREADY, W.G. 1997. No free lunch theorems for optimization. *IEEE Transactions Evolutionary Computation 1*, 67-82.
XIAO, M., EL-ATTAR, M., REFORMAT, M. AND MILLER, J. 2007. Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques. *Empirical Software Engineering 12*, 183-239.
XU, R. AND WUNSCH, D. 2005. Survey of Clustering Algorithms. *IEEE Transactions on Neural Netwoks 16*, 645-678.
YOO, S., HARMAN, M., TONELLA, P. AND SUSI, A. 2009. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the 18th ACM International Symposium on Software Testing and Analysis (ISSTA)*2009, 201-212.