Using Behaviour Inference to Optimise Regression Test Sets

Ramsay Taylor, Mathew Hall, Kirill Bogdanov, and John Derrick

Department of Computer Science, The University of Sheffield

Abstract. Where a software component is updated or replaced regression testing is required. Regression test sets can contain considerable redundancy. This is especially true in the case where no formal regression test set exists and the new component must instead be compared against patterns of behaviour derived from in-use log data from the previous version. Previous work has applied search-based techniques such as Genetic Algorithms to minimise test sets, but these relied on code coverage metrics to select test cases. Recent work has demonstrated the advantage of behaviour inference as a test adequacy metric. This paper presents a multi-objective search-based technique that uses behaviour inference as the fitness metric. The resulting test sets are evaluated using mutation testing and it is demonstrated that a considerably reduced test set can be found that retains all of the fault finding capability of the complete set.

1 Introduction

Where a software component is updated or replaced regression testing is required. It is often the case that there is no documented specification for the behaviour of the component. In the absence of a specification for the component, regression tests can be used to attempt to determine that a de-facto specification — the behaviour of the previous version — is adhered to.

In the absence of regression test sets, the only available regression test is the replay of trace data from in-service logs of the previous version. This trace data contains sequences of inputs and outputs, or sequences of function calls that are applied to the component. The traces can identify both sequences of behaviour that must be accepted, and sequences that should not occur. When sufficient log data is available this provides an accurate picture of the behaviour that the rest of the system expects from the component, so replay of traces forms a meaningful regression test.

Running regression tests is often expensive, usually due to the large number of tests that comprise a test suite. Often, multiple regression tests will exercise the same behaviour, resulting in time being spent on tests that add no value. This is especially true of test sets derived from in-service data.

An improved test suite can be developed by selecting a subset of the traces with enough traces to exercise each part of the behaviour for the lowest overall cost. Figure 1 illustrates the minimisation process. In the example, a subset of the eight regression tests is selected $(1, 4, 5)$, and must then be evaluated for its

B. Nielsen and C. Weise (Eds.): ICTSS 2012, LNCS 7641, pp. 184–199, 2012.

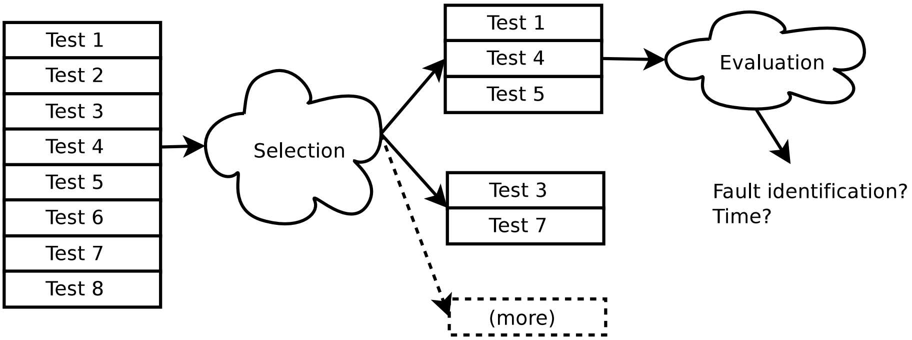


Fig. 1. Problems facing test set selection

suitability. The naive, exact solution is to repeat this process for all subsets of the test suite, but this is impractical for most test suites, and impossible for test suites generated from log data that can contain billions of lines [18]. It has been shown in [12] that finding a suitable regression test subset is NP complete and so heuristic approaches are required. A detailed survey of regression test selection literature can be found in [3].

The problem has two distinct elements. Firstly, given that considering all candidate sets is infeasible, the selection step must be performed more intelligently. Secondly, once a candidate is selected, difficulty remains in assessing its suitability as a reduced test set.

Genetic Algorithms (GAs) are one solution to the selection problem. They are a class of heuristic approaches that have been used successfully in many software engineering problems [9] and provide efficient selection of possible solutions from large search spaces. In a GA, a pool of good solutions is recombined to locate better solutions. This relies on measurement of the "fitness" of these solutions. Despite its efficiency, a GA will still consider a large number of solutions before it locates a suitable result, making its execution time highly dependent on both the accuracy and the speed of the evaluation mechanism.

Previous work has used code coverage to address the evaluation problem. Genetic Algorithm approaches to the selection of subsets were presented in [13] and [19], and sought to identify the minimum set of traces that achieves complete code coverage. Test sets selected for code coverage have been shown to provide a low "behavioural adequacy" [6] — that is, their fault finding ability is lower than test sets selected for behavioural coverage. An alternative measure of a test set is provided by behaviour inference. Using behaviour inference as a measure of test adequacy was first suggested in 1983 [17], and has been shown to provide a better foundation for selecting test sets [6]. While behavioural inference requires some computation time this grows with the number of test cases, regardless of their content. However, the time taken to execute a test set is unbounded as each test may depend on external factors such as communication, which may take an arbitrarily long time.

The contribution of this work is to apply the behaviour inference approach to the problem of regression test set reduction using multi-objective search techniques. The work is directed at the following three Research Questions:

RQ1. What reduction in test set size can be obtained using a GA with the behavioural inference fitness metric?
RQ2. Is behavioural inference a good selector of high quality test subsets?
RQ3. How intelligently does the GA search technique use a constrained budget?

This paper makes the following contributions:

- A fitness metric for test subsets based on comparisons of inferred behaviour between the subset and its parent.
- A multi-objective search approach based on the NSGA-II algorithm that applies this metric to identify a Pareto front of fast, accurate test sets.
- An evaluation of the solutions found by this approach that demonstrates the search efficiency and validates the results by comparing mutation testing outcomes between the subsets and their parent.

The remainder of this paper is organised as follows:

- Section 2 introduces the vending machine example that is used throughout this paper.
- Section 3 and Section 4 present the process of evaluating a test set using passive learning algorithms, and the genetic algorithm for efficiently searching the possible test subsets.
- The results of the process are evaluated in Section 5.
- Section 6 contains conclusions.

2 The Vending Machine Example

The example used throughout this paper is a simple vending machine simulation. The case study was implemented in Erlang[1] but the techniques developed operate on traces and are independent of the implementation language. The implementation consists of 60 lines of Erlang code. The module presents an interface with a number of functions that represent operations on the machine – `start,coin,choc,toffee` — some of which take parameters and all of which produce a return value.

The behaviour of the vending machine program can be best understood with reference to the state machine shown in Figure 2, which presents a convenient abstraction. The format for the transition labels is a triple with the name of the function, a list of arguments, and the expected return value. For example, {`coin,[2],ok`} represents a call to the `coin` function with the parameter 2 and the expected response `ok`. The machine is initialised with the `start` function, and can be re-initialised at any point, resetting it to the initial state. The machine

[1] Erlang is a concurrent, functional programming language developed by Ericsson and used extensively in the telecoms industry [2].

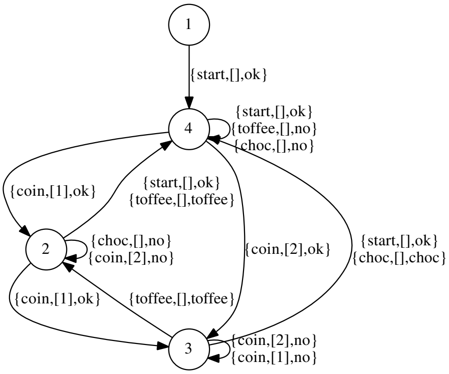


Fig. 2. The vending machine state machine

accepts 1p and 2p coins and charges 1p for `toffee` and 2p for `choc`. Since 2p is the most expensive item, the machine will reject coins that would take its stored value above 2p (this serves to make the machine's state space finite).

In the scenario in this example, a test set was generated randomly according to the following process. Elements of the trace alphabet were randomly selected and conjoined to produce traces. In this example the trace alphabet consists of the interface functions, combined with the valid arguments and valid responses for each function. These were then passed to a program that attempted to execute the generated sequence of function calls and checked that the returned values were consistent with those generated. Where there was an inconsistency — or where there was an exception thrown by the program — the trace was truncated at that point and marked as negative. If all elements of a trace completed successfully then it was marked as a positive trace. This forms a prefix-closed language describing the behaviour of the program.

```
+ {start,[],ok} {coin,[1],ok} {coin,[2],no} {coin,[1],ok}
+ {start,[],ok} {start,[],ok} {toffee,[],no} {choc,[],no}
- {start,[],ok} {coin,[2],ok} {toffee,[],no}
+ {start,[],ok} {coin,[2],ok} {toffee,[],toffee}
```

Fig. 3. Some example traces from the case study test set

A random test set was constructed by adding random traces to it until the set covered all the behaviour of the state machine inFigure 2. This produced a test set of 190 traces; some example traces are shown in Figure 3. The third trace is listed as negative because it expects the response `no` to the `toffee` call after giving the machine a 2p coin, when the implementation will instead respond `toffee`. This observed behaviour is listed in the fourth trace.

3 Using Behaviour Inference to Assess the Suitability of a Test Subset

As discussed in Section 1, a technique is needed to select possible test subsets and then evaluates their adequacy. This section defines an evaluation strategy using behaviour inference. This will form the fitness function for the selection technique presented in Section 4. The aim of this fitness function is to assess the *behaviour* exercised by the test cases, addressing the shortcomings present in previous work that uses code coverage.

In the example in Figure 1, the adequacy of the subset (1,4,5) must be evaluated against the full set (1,2,3,4,5,6,7,8). To do this a model is inferred of the behaviour covered by the full set, and then a model is inferred of the behaviour covered by the subset. These are then compared to assess the extent to which coverage has been maintained. This is particularly pertinent in the regression case where the only specification of correct behaviour is the log data of an existing system, so the model inferred from the complete set is the only available standard against which test set coverage can be measured.

3.1 Behaviour Inference

Behaviour inference builds finite state machine models from trace data using algorithms inspired by language learning models.

Passive learners aim to learn state machine language representations from partial data, usually a subset of the possible sequences of the language. Evidence driven state merging (EDSM) algorithms such as BlueFringe [11] operate on a set of positive and negative traces from a system or language and produce a state machine that accepts positive traces and rejects (in the form of a transition to a failure state) negative traces. It was shown to be highly effective in the "Abadingo One" [11] competition for learning algorithms.

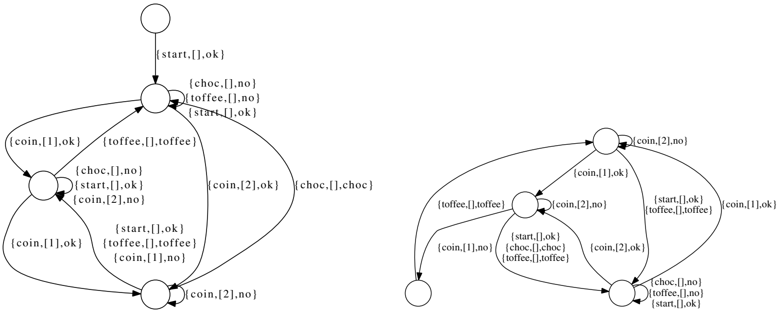
We use the StateChum system [1], which was developed to implement an EDSM algorithm based on BlueFringe with the objective of reverse engineering state machine representations of software behaviour from software trace data [16]. The algorithm operates on prefix closed languages, which are a good model of software execution traces since traces cannot exist with failing prefixes. The supplied traces are merged into a Prefix Tree Automaton (PTA), which represents an accurate but excessively large FSM representation of the behaviour covered by the supplied traces. The algorithm proceeds by merging states to reduce the size of the state machine without altering the accepted language. The choice of states to merge is based on various *evidence*, such as similar outgoing traces, and the absence of explicit negative traces in the test set that would become accepted after the merge.

3.2 State Machine Comparison

A key component of the evaluation mechanism presented in this section is the comparison of inferred state machines. This paper will use what is called the

Balanced Classification Rate (BCR) measure to compare state machines, and thus compare the behaviour coverage of the test sets that produced them.

The BCR metric measures the accuracy of a state machine against a reference machine in terms of the correct classification of traces as either positive (accepted) or negative (rejected). Each of the possible traces of the reference machine are classified by the machine being measured and four sets are produced: true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). For machines with cycles the set of possible traces is infinite; this paper uses the W-Method [4] to produce a suitable set of traces that cover all of the behaviour in a finite set. For example, in Figure 4 the right hand FSM accepts traces that do not start with {`start,[],ok`}, whereas the correct model does not, so these form false positives.



(a) Parent

(b) BCR:0.81

Fig. 4. The vending machine behaviour as inferred from the complete random trace set, and a behaviour model with a BCR score of 0.81 inferred from less than 4% of the trace set

These figures are used to compute two numbers, C_{plus} and C_{minus}, where $C_{plus} = TP/(TP + FN)$ and $C_{minus} = TN/(TN + FP)$. The BCR value is the harmonic mean between C_{plus} and C_{minus} and is defined as: $BCR = (2 * C_{plus} * C_{minus})/(C_{plus} + C_{minus})$. This produces a "score" between 0.0 and 1.0, where 1.0 represents correct classification of all traces, and smaller numbers represent decreasing levels of accuracy.

3.3 Subset Evaluation Metric

With these components in place, each candidate test subset can be compared to a reference model to produce a numeric adequacy value. The behaviour inference process is applied to the complete test suite and the inferred model forms the reference for all subset evaluations. This inference only needs to be conducted once, at the beginning of the process.

Each candidate test subset is compared to the reference set by inferring a model from the subset and then computing the BCR score (as described in

Section 3.2) of that model against the reference model. The BCR score gives a numeric metric to the accuracy of the model, and permits the user of the process to allow a trade-off between speed and accuracy. In cases where the optimal, completely accurate regression test still requires many hours to run it may be useful to produce a regression test that can be run in minutes and still find 80% of faults. This could serve as a quick regression test during the development program, that is supplemented by the full regression test at the conclusion of a development phase. By using a numeric metric of accuracy, such as BCR, this process allows quantitative decisions to be made about the appropriate accuracy/speed trade-off that would not be possible using less semantic measures such as code coverage.

Inferring a model from a set of recorded traces is much faster than actually performing tests themselves. Thus using recorded traces, either from in-service log data or a run of the existing regression test set, the time taken to evaluate the behavioural coverage of a regression test subset can be minimised versus running the tests to evaluate it. For example, the inference process on the 146 line trace set takes under 1 second, whereas evaluating the complete set of traces against an implementation takes over 50 seconds. It is shown in Section 5.3 that model inference allows many thousands of candidates to be evaluated in less than an hour. Model inference therefore provides a process by which candidate solutions can be quickly evaluated, allowing many thousands of possible subsets to be evaluated in a reasonable time.

4 Test Subset Selection Using Multi-objective Search

Having developed a suitable fitness function, the other component of the technique is the selection of candidate test subsets. This section presents a multi-objective search algorithm using the NSGA-II algorithm.

4.1 Genetic Algorithms

This paper uses genetic algorithms to approach the NP complete problem of selecting test subsets from a given test suite. The field of Search-Based Software Engineering [9] includes various techniques that provide general, meta-heuristic approaches to problems for which it is impossible or impractical to find perfect solutions. Genetic Algorithms (GAs) are one such technique that uses an evolutionary metaphor to direct and optimise a limited search of a very large search space. This makes them ideal for a task such as test suite optimisation where the search space is large, and while the best solution is desirable, there is a large space of sub-optimal, but still satisfactory solutions that a GA can discover.

The critical elements of a GA are: a representation of an individual candidate solution as a collection of “genes”, and a “fitness function” that produces a numerical measure of the “correctness” of this candidate.

The search consists of the application of *operators* on a population comprised of a number of chromosomes, each of which containing one or more genes. The minimum operators required are selection, crossover and mutation.

The selection operator uses the fitness values of the chromosomes to determine which are selected in the next generation. Crossover and mutation are concerned with exploring the search space. Crossover models reproduction, where two individuals are mixed together to produce offspring. Crossover operators traditionally consist of a series of swap operations, however more complicated ones exist. The mutation operator serves to prevent the search from remaining in local optima by ensuring diversity in the population. It operates by randomly altering one or more genes in an individual to produce a new individual.

4.2 Chromosome Representation and Genetic Operators

The chromosome representation for a test set subset consists of a set of bits where each bit represents a trace or test from the original test set. Bits set to 1 represent that that trace is included in this subset, 0 represents that the trace is not included. Each bit is considered a gene by the implementation. The chromosome is therefore comprised of Tc bits, where Tc is the number of traces in the original set. Crossover is enacted by picking some n where $n < Tc$; the new chromosome is formed from bits 1 to n from one parent, and bits $n + 1$ to Tc from the other.

Random mutation of a bit would simply invert it. As the number of bits set to 1 or 0 decreases (i.e. the chromosome is “mostly 1s” or “mostly 0s”) this becomes less effective, since inverting a random bit becomes decreasingly likely to further reduce the number of 0s or 1s. This is a significant issue for test set minimisation in cases where a significant optimisation is possible, or very little optimisation is possible, since the ideal solution may have very few traces included (1s) or very few traces excluded (0s).

This implementation uses two mutation operators: one that randomly “turns on” a bit, and one that randomly “turns off” a bit. This ensures that every generation contains some new solutions with more traces and some with fewer traces, even as the population tends towards one extreme or the other.

The binary nature of the individual genes also has a significant impact on the generation of the initial population for the search. If the initial chromosomes are simply generated randomly then they will all have approximately 50% of the bits set and approximately 50% unset. In theory the crossover and mutation will eventually correct for this, but it was found that producing a more even distribution of initial individuals allowed the search to focus more rapidly. Consequently, individuals for the initial population are created by first selecting a random number of bits to be set, and then a random distribution of those bits across the chromosome. This results in an even distribution of individuals from some with 0 bits set to some with Tc bits set.

4.3 Multi-objective Search Algorithms

The test set reduction problem involves optimising for both test speed and fault identification. In such instances where there are multiple, independent variables to optimise, this can be achieved by weighting the squares of each variable and using this as the fitness function for a conventional GA, however this has the possibility that a solution with a very high score for one objective will overpower solutions that have a more balanced profile. Consequently, this work uses a specialised multi-objective genetic algorithm, which has a more intricate selection process that seeks to avoid this drawback.

The search heuristic used in this paper is based on the "Non-dominated Sorting Genetic Algorithm II" (NSGA-II) [5], which has been demonstrated as effective in test set reduction by [19]. For each candidate solution the NSGA-II selection mechanism measures the number of other solutions that "dominate" this one, where dominance is defined as being superiour in all objectives. Those that are undominated represent the Pareto front of the currently explored solutions. These are retained as the population for the next generation.

Where there are many Pareto-optimal solutions it becomes necessary to select a subset. The subset is chosen to avoid clustering — that is, the selection mechanism attempts to distribute the chosen solutions evenly over the Pareto front. This distribution creates a much more effective basis for the next generation, since breeding solutions that excel in different objectives should produce offspring which posses both qualities. As an example: breeding a very accurate test set with a very fast one may produce offspring that retain most or all of the accuracy but are considerably quicker.

4.4 Fitness Evaluation

The minimisation process has two variables to be optimised: fault identification and cost.

Fault identification is estimated using the *BCR score* for the learned machine from a candidate individual as described in Section 3.3.

The "cost" of the test set may be defined as the time taken. For the purpose of the case study in this paper, uniform cost is assigned to all tests in a candidate subset, therefore the "cost" is a function of the size of the test set. In a more complex testing scenario it would be important to consider other costs, such as the execution time of each test. This may not be related to trace length, as some tests may include features such as external synchronisation that causes them to take arbitrarily long times. Wherever trace count (Tc) is used in the subsequent discussion this could be substituted for trace *cost*, which would be the sum of individual costs of tests in a test set. The *trace improvement* of a subset is defined as the number of test cases removed from the total set, (*i.e.* the number of 0 bits in the chromosome). This is expressed as a fraction of the total chromosome size: $Trace\ Improvement = \frac{Tc - Tc_{individual}}{Tc}$.

4.5 Selection

The two values, BCR score and Trace Improvement, measure the solution's fitness in each objective dimension. A solution's dominance is then measured by comparison to the other solutions in the current population. The selector proceeds by choosing undominated solutions. If the number of them exceeds a defined limit then it begins to reduce the set by measuring the "distance" between solutions. The distance is measured as the sum of the absolute difference between the accuracies and the trace improvements. Solutions with identical accuracy and trace improvement have a zero distance and one of the pair will be removed. If the size is still excessive then a threshold distance is established at 0.01 and steadily increased. Any solutions within the threshold distance are pruned and the threshold increased until the population size falls below the limit.

5 Evaluation

This evaluation seeks to assess the test set improvement that can be obtained using the multi-objective behavioural adequacy approach, as well as to validate the testing capability of the solutions it generates compared to that of the original test set. The research questions this evaluation answers are those defined in Section 1.

The evaluation was conducted on the vending machine example given in Section 2. A test was produced containing 190 randomly generated traces, which covered all behaviour in the program, such that running the EDSM algorithm (Section 3.1) on it will produce the state machine in Figure 2. The random nature of the test set means it is likely to contain duplication — traces that exercise the same behaviour, offering no value. The objective of the evaluation was to determine how much of this duplication the GA removed

5.1 RQ1: Reduction in Test Set Size

This research question is answered in terms of the *trace improvement* (as defined in Section 4.4) the search yields for test sets that exercise the same behaviour as an unoptimised test set.

The test set was optimised using the multiobjective genetic algorithm as detailed in Section 4.3. The individuals were assessed in terms of the constituent variables that make up their fitness (BCR and trace improvement).

The genetic algorithm used a 100% crossover rate, generating one extra individual for every pair of parents, a population size of 10 after each round of selection, and a mutation rate matched to the size of the trace set so that each individual in the population produces one mutant with one gene altered. Crossover and mutation were only applied to the surviving members of the previous fitness evaluation. Fitness values were recorded for all individuals evaluated as the search progressed. The search was repeated 30 times to ensure statistical significance.

The genetic algorithm was limited to 75 generations, which produced an average of 3500 evaluations of distinct individuals. A random set of 3500 individuals was generated and evaluated in the same way. This process was repeated 30 times to produce 30 sets of evaluations, each of a size that is comparable to a run of the GA.

5.2 Results

RQ1: Reduction in Test Size. The individual with the highest trace improvement and a 1.0 BCR was selected from each of the 30 sets of individuals

Table 1. Trace Improvement scores for the best individual with a 1.0 BCR score from each iteration for both search types

	Min	Max	Median	Mean	St. Dev.	N	p-value
GA	0.757	0.936	0.921	0.909	0.037	30	< 0.005
Random	0.710	0.873	0.808	0.806	0.028	30	

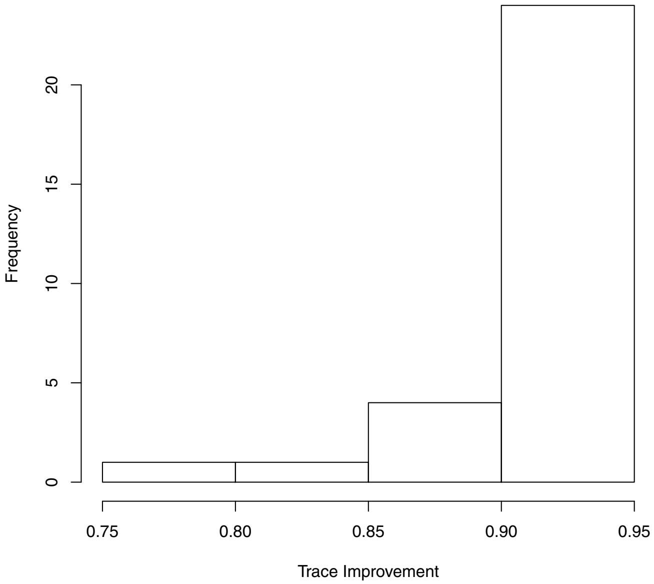


Fig. 5. Histogram of the trace improvement of the best individual with a 1.0 BCR score from each GA search

from the minimisation step. This was performed for both the genetic algorithm implementation as well as the random set. Statistics for the trace improvement values of these two samples are shown in Table 1 as well as visually for the GA in Figure 5.

The results demonstrate that a high reduction in test case size is achieved using the approach. Both search algorithms located reduced test sets with a high trace improvement, with the minimum being over 0.7. Consistency is demonstrated by the small standard deviation, which equates to a difference of 4% of the original test set size.

5.3 RQ2: Efficacy of the Behavioural Adequacy Metric

The validity of the behavioural adequacy metric as a measure of the fault identification of a test subset is assessed using mutation testing. Mutation testing [8] evaluates test sets by simulating faults in a software system and measuring the test set's ability to identify the faults.

Mutation testing modifies the software's source code to produce a "mutant". The mutant code is recompiled and then tested by the test suite. If the mutant fails the test suite it is referred to as "killed", if not then it is "alive". A large percentage of simple syntactic mutations result in code that does not compile; many of those that do compile have no functional change. Semantic mutation testing (SMT) [10] is intended to improve on this by applying mutation to a parsed form of the program. The *muTestErl* [7] system for Erlang semantic mutation testing was used in this evaluation.

To use mutation testing to evaluate BCR, the complete test set was first run on 4807 mutants. The complete set kills all but 386 of these mutants. These remaining mutants contain modifications that do not alter the observable behaviour of the program — many change the type of the internal representation of the stored coin value from integer to floating point, for example, but the Erlang implementation performs suitable implicit casting at each application of an arithmetic operator to allow all the updates and comparisons to continue to operate.

A selection of 288 randomly generated individuals were then subjected to mutation testing. Each of these individuals was a reduced test set, for which the BCR was calculated. The same set of mutants was then run on each of these reduced test sets, each killing some number of mutants. The result of these runs are shown in a scatterplot in Figure 6.

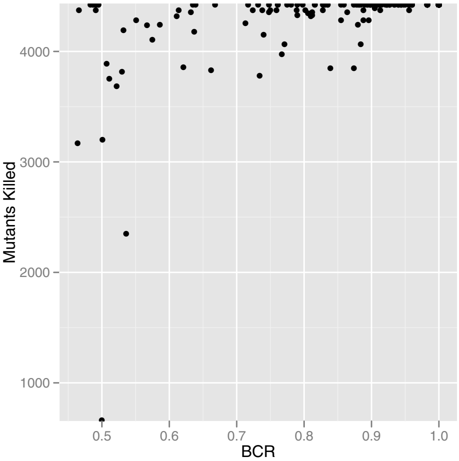


Fig. 6. Scatter-plot showing number of mutants killed against BCR for a selection of random test sets

The scatterplot shows that mutant killing power does not correlate highly with BCR (Pearson $r = 0.457$), however, it also shows that for particularly high values of BCR that the mutant killing power is as high as the original test set. In this evaluation BCR is observed to be prone to *false negatives*, but does not appear to suffer from false positives: of all the test sets with a BCR of 1.0, none killed fewer mutants.

These results suggest that BCR is likely to be too conservative, as smaller test sets with an equal mutant killing power will be discarded if they have a low BCR score. The multiobjective search used in this approach can mitigate this problem; adjacent solutions to test sets with a high trace improvement but low BCR score may still be explored, possibly yielding better adjacent solutions. This also has the implication that individuals on the Pareto front with sub-1.0 BCR scores (but better trace improvement) should not be rejected outright, as they are potentially more suitable minimisations of the test set.

RQ3: How Intelligently does the GA Search Use a Constrained Budget? The search space for an exhaustive approach for the case study is 2^{190}. The results for **RQ1** demonstrate the search reliably finds a test set that is significantly smaller than the original regression test set, despite only exploring a small fraction ($30,000$ candidates) of the total search space.

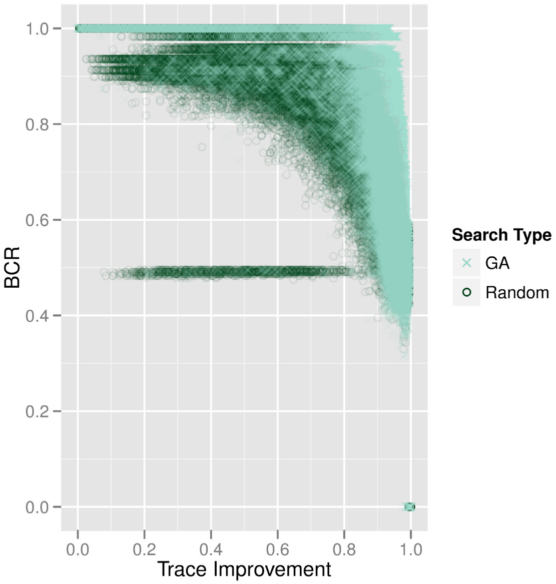


Fig. 7. Scatter-plot of the individuals evaluated by each search

Figure 7 shows a scatterplot of the trace improvement versus BCR score for each of the individuals selected by both the random and the genetic algorithm searches. Maximised values in both direction represent objectively better solutions. The plot clearly shows a Pareto front has been located by the genetic algorithm, which is beyond the region explored by the random searcher. Along this front are the best individuals found by the search. This front demonstrates the tradeoff between behavioural adequacy (BCR) and trace improvement. As

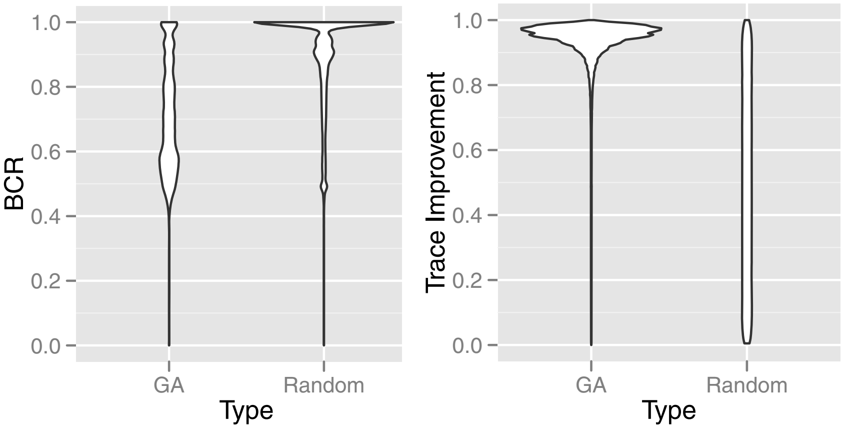


Fig. 8. Violin plots of BCR score and trace improvement of all individuals generated by each search

the findings for **RQ2** suggest that some solutions with sub 1.0 BCRs may be equally suitable as those with 1.0 BCR scores, the individuals to the right of the plot may actually be better overall.

The random search results exhibit a pattern where a large number of individuals with a widely varied trace improvement have the same BCR score. This is due to the common lack of a critical trace amongst this subset of the population and illustrates the difficulty of the selection problem, as well as demonstrating the benefit of intelligently searching the space, rather than randomly sampling it.

The "p-value" column of Table 1 gives the p-value for a one-sided Wilcoxon test. The p-value gives an indication of the likelihood that the null hypothesis (H_0) is true. In this case, H_0: median(GA) $\ngtr$ median(Random). As the p-value is small, the null hypothesis is rejected, indicating the difference in medians is statistically significant, demonstrating that the GA outperforms the random searcher.

Figure 8 shows a visualisation of the BCR scores and trace improvements of the solutions considered by both search algorithms. The higher density towards the higher BCR scores for the random searcher can be explained by the uniform distribution of trace improvement - solutions with more traces (*i.e* a lower trace improvement) are more likely to have a high BCR score. The genetic algorithm exhibits a more varied distribution of BCR scores, accounted for by the amount of solutions with higher trace improvement it considered; this corresponds to the Pareto front that is illustrated in Figure 7.

5.4 Threats to Validity

There are several factors that constitute potential threats to validity.

The performance of the GA may be due to chance. It is possible that the results of the genetic algorithm may be entirely due to random chance. This risk is mitigated firstly by the large number of repetitions of the search, reducing the possibility of a non-representative sample being observed. The comparison of the GA against the random set shows that its performance is higher than a purely stochastic search process, meaning that this threat is eliminated.

The case study may not be representative of software systems. This may impact the amount of improvement observed for other systems. The trace improvement that can be obtained is dependent on the amount of duplication in the test set; the technique presents no benefit if a test set is already minimal. This does not impact the findings shown however, as it demonstrates that, given a means to calculate the BCR score of a candidate test set (and a test set that can be minimised), a reduction is possible using this technique.

6 Conclusions and Future Work

This work presents an application of a genetic algorithm to the test suite minimisation problem. Although genetic algorithms have been used before, this is the first instance where the *behaviour* exercised by the tests has been used as a selection criterion. The evaluation demonstrates that the approach is able to locate high-quality minimised test sets using BCR to estimate the behaviour exercised by the tests, identifying a test set that exhibits the same behaviour, despite having 90% of the traces removed from it. This is validated in two aspects, comparison with a random algorithm, as well as the muation testing of the results to compare BCR accuracy.

The findings show this is a promising algorithm, however the survey only applies it to a small example. Real world performance is not likely to vary, however, as the learning process occurs separately from the software. GAs can handle large search spaces well, so larger test sets should be reduicible using this technique as well as smaller ones (albeit with a higher fitness evaluation budget). Immediate future work will seek to apply the minimisation process to larger and more varied case studies to test this hypothesis.

Although in this work BCR worked well, it is possible to utilise other comparison techniques [15,14] that aim to ensure that a less than perfect score corresponds to a small change in an inferred model. Whereas BCR is generally sensitive to transitions close to the initial state, these methods aim to ensure the measure of difference computed is relatively equally sensitive to a missing or added transition anywhere in an automaton. Future work could use these alternatives to improve the accuracy of the test adequacy measure, therefore allowing potentially better trace improvement scores to be reached.

References

1. StateChum, `http://statechum.sourceforge.net/` (accessed March 14, 2012)

2. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in ERLANG. Prentice Hall (1996)
3. Biswas, S., Mall, R., Satpathy, M., Sukumaran, S.: Regression Test Selection Techniques: A Survey. Informatica 35(3), 289–321 (2011)
4. Chow, T.S.: Testing Software Design Modeled by Finite-State Machines. IEEE Trans. Softw. Eng. 4(3), 178–187 (1978)
5. Deb, K., Agrawal, S., Pratap, A., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation 6(2), 182–197 (2002)
6. Fraser, G., Walkinshaw, N.: Behaviourally Adequate Software Testing. In: Proceedings of the Fifth International Conference on Software Testing, Verification and Validation (ICST) (2012)
7. Guo, Q., Derrick, J., Taylor, R.: Mutation Testing of Erlang Distributed and Concurrent Applications. In: Automated Software Engineering (submitted, 2012)
8. Hamlet, R.G.: Testing programs with the aid of a compiler. IEEE Transactions on Software Engineering 3, 279–290 (1977)
9. Harman, M., Jones, B.F.: Search-based software engineering. Information & Software Technology 43(14), 833–839 (2001)
10. Clark, J.A., Dan, H., Hierons, R.M.: Semantic Mutation Testing. Science of Computer Programming (2011) (page under print)
11. Lang, K.J., Pearlmutter, B.A., Price, R.A.: Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm. In: Honavar, V.G., Slutzki, G. (eds.) ICGI 1998. LNCS (LNAI), vol. 1433, pp. 1–12. Springer, Heidelberg (1998)
12. Lin, J.-W., Huang, C.-Y.: Analysis of test suite reduction with enhanced tie-breaking techniques. Information & Software Technology 51(4), 679–690 (2009)
13. Mansour, N., El-Fakih, K.: Simulated Annealing and Genetic Algorithms for Optimal Regression Testing. Journal of Software Maintenance 11(1), 19–34 (1999)
14. Pradel, M., Bichsel, P., Gross, T.R.: A framework for the evaluation of specification miners based on finite state machines. In: ICSM, pp. 1–10. IEEE Computer Society (2010)
15. Walkinshaw, N., Bogdanov, K.: Automated Comparison of State-Based Software Models in terms of their Language and Structure. ACM Transactions on Software Engineering and Methodology 22(2) (2012)
16. Walkinshaw, N., Bogdanov, K., Holcombe, M., Salahuddin, S.: Reverse engineering state machines by interactive grammar inference. In: Proceedings of the 14th Working Conference on Reverse Engineering (WCRE). IEEE (2007)
17. Weyuker, E.J.: Assessing Test Data Adequacy through Program Inference. ACM Transactions on Programming Languages and Systems 5(4), 641–655 (1983)
18. Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M.: Experience mining google's production console logs. In: Proceedings of the 2010 Workshop on Managing Systems via Log Analysis and Machine Learning Techniques, p. 5. USENIX Association (2010)
19. Yoo, S., Harman, M.: Using hybrid algorithm for Pareto efficient multi-objective test suite minimisation. Journal of Systems and Software 83(4), 689–701 (2010)