The Impact of Input Domain Reduction on Search-Based Test Data Generation

Mark Harman, Youssef Hassoun, Kiran Lakhotia
CREST, King's College London
Strand, London
WC2R 2LS, UK

Phil McMinn
University of Sheffield
Regent Court,
211 Portobello St
Sheffield, S1 4DP, UK

Joachim Wegener
DaimlerChrysler AG
Alt-Moabit 96a,
D-10559 Berlin
Germany

ABSTRACT

There has recently been a great deal of interest in search–based test data generation, with many local and global search algorithms being proposed. However, to date, there has been no investigation of the relationship between the size of the input domain (the search space) and performance of search–based algorithms. Static analysis can be used to remove irrelevant variables for a given test data generation problem, thereby reducing the search space size. This paper studies the effect of this domain reduction, presenting results from the application of local and global search algorithms to real world examples. This provides evidence to support the claim that domain reduction has implications for practical search–based test data generation.

Categories and Subject Descriptors. D.2.5 [Software Engineering]: Testing and Debugging – *Testing Tools*; I.2.8 [Artificial Intelligence]: Problem Solving, Control Methods, and Search – *Heuristic Methods*

General Terms. Algorithms, Experimentation, Measurement, Performance, Theory

Keywords. Automated test data generation, evolutionary testing, genetic algorithms, hill climbing, search space reduction, input domain reduction

1. INTRODUCTION

The generation of adequate test sets remains one of the primary cost drivers for software testing. Testing itself is also widely believed to consume a significant portion of overall development effort [2]. This makes the automated construction of adequate test data an important software engineering concern and one in which the level of interest has remained high for several decades.

The past decade has witnessed a particularly dramatic increase in activity concerning search–based test data generation, with many authors proposing both local search [12, 23] and global search [14, 17, 20, 24] algorithms for test data generation. Search algorithms are guided by a fitness function that captures the test adequacy criterion. This makes the overall approach very general, since many different test criteria can be captured, simply by changing the fitness function. The technique has been applied to many test data generation problems, including criteria for white box structures [12, 14, 17, 20, 24, 28], functional characteristics [25], stress–based behaviour [5], integration test ordering [4] and safety properties and exceptions [22, 23].

Despite this increasingly large literature, there have, hitherto, been no published results that explore the relationship between the size of the search space and the performance of search–based algorithms. This paper is the first to characterise and empirically explore the search–space/search–algorithm relationship for search–based test data generation.

The flexibility of the search–based approach leads to many test data generation application areas, each of which may lead to a different relationship between search space and search algorithm. In this paper we chose to study the structural test data generation paradigm, because this has received the most attention from other authors, making it not only the most mature area of research, but also the most overdue for a deeper analysis.

The stochastic nature of metaheuristic search algorithms indicates that any theoretical treatment will always need to be supported by empirical results. To address this, the paper provides an analysis of the expected relationship between search space and algorithm performance, supported by experiments, showing that the predictions made find application in real world examples of industrial and open source test data generation problems.

The results provide evidence to support the claim that both hill climbing and genetic algorithms for test data generation have a lot to gain from search space reduction, with statistically significantly improved results in many cases. Random search, insofar as it is effective as a test data generation technique is unaffected by search space reduction.

The primary contributions of the paper are as follows:

1. The paper introduces a theoretical analysis of random, local and global search, exploring the impact of domain reduction on search–based algorithm performance.

2. The paper presents the results of verification and validation experiments designed to test these predictions.

3. The paper evaluates the implications of the work, studying test data generation problems from open source and industrial applications.

The rest of the paper is organised as follows. Section 2 overviews search-based test data generation. Section 3 introduces the technique used to reduce the search space. Section 4 provides a theoretical analysis of the impact of input variable removal on the search techniques, whilst Sections 5 and 6 present the empirical study and results. Naturally there are threats to validity in any empirical study such as this, and these are detailed in Section 7. Section 8 presents related work, and Section 9 concludes.

2. SEARCH-BASED TESTING

Search–based test data generation uses metaheuristic algorithms to generate test data. This paper focuses on structural testing, in particular branch coverage. In order to cover a particular branch in a unit under test, the goal is to construct an input vector for a function which drives execution of the program down the branch of interest. The search space is formed from the set of possible input vector parameter–value combinations.

Metaheuristic algorithms require a numerical formulation of the test goal, from which a fitness function can be derived. The purpose of the fitness function is to guide the search into promising, unevaluated areas of the search space. For coverage of a branch, the fitness function is calculated by normalizing a 'branch distance' measure with another metric known as the 'approach level'. The goal of the search is to find the global minimum of the fitness function, *i.e.* zero.

When execution of a test case diverges from the target branch, the branch distance expresses how close an input came to satisfying the condition of the predicate at which control flow for the test case went 'wrong'. For example, for the coverage of the true branch from node 1 in Figure 1(a), the branch distance is computed using the formula `|dist-0|`. The closer `dist` is to zero, the 'closer' the true branch is to being taken. This can be seen in a plot of the fitness landscape (Figure 1(c)).

The approach level calculation comes into effect when there are several conditions that must be satisfied in order to execute the target, for example the true branch from node 8. It is a measure of how many of the target's control dependencies were not encountered in the path executed by the input vector. For structured programs, the approach level reflects the levels of nesting surrounding the target (Figure 1(b)).

The following sections provide a detailed overview along with the parameters used for the search techniques in this paper, in order to facilitate replication of the empirical study which appears in Section 5. The test data searches were performed using the IGUANA tool [16].

2.1 Random Testing (RT)

Random testing (RT) has been shown to be a surprisingly effective way of generating test data [8]. It is possible to cover many structural targets using this simple technique, because there are often several input vectors that can be selected that are good enough to execute most of the structures of a program. For example, the false branch from node 1 is easily covered in Figure 1(a), because it is executed by all input vectors apart from those for which `dist` is zero. The chances of executing the alternative true branch at random however, are significantly lower. For such test targets, more intelligent techniques are required.

2.2 Hill Climbing (HC)

Hill climbing (HC) is a metaheuristic search technique that seeks to improve one candidate solution by exploring its neighbouring search space. The implementation of the HC algorithm for this paper is based on the 'alternating variable method' introduced by Korel [12]. This method explores the 'neighbourhood' of each input variable in the input vector in turn. If changes in the values of the input variable do not result in an increased fitness, the search considers the next input variable, and so on - recommencing from the first input variable if necessary - until no further improvements can be made or test data has been found.

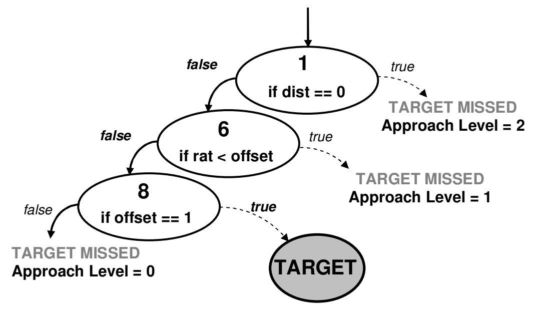
Consider the example shown in Figure 1. Suppose the target is the true branch from node 8, and the initial random input is $< 1, 50, 1, 1 >$. The input reaches node 8 but takes the false branch. The alternating variable method begins to perform 'exploratory moves' on each input variable by inducing the smallest possible increase, followed by the smallest possible decrease. Suppose the accuracy of the double variables in the example of Figure 1 is set to 0.1 for the purposes of the search. The moves made around the `dist` variable are 0.9 and 1.1. However, no improvement is made - `offset` is no closer to becoming `1`. The search continues on to consider the next variable in the input vector, `offset`. An increased value of 50.1 moves `offset` closer to the value of `1` required at node 8 via the division statement at node 3. At this point, the search makes accelerated 'pattern' moves in the direction of improvement. In this paper, the value of the i^{th} move m_i made in the direction of improvement, $dir \in \{-1, 1\}$ is computed using $m_i = s^i \cdot acc_v \cdot dir$, where acc_v is the accuracy of the v^{th} variable, and s is the *repeat base* ($s = 2$ for experiments in this paper). Successive moves are made for `offset` until it overshoots the required value of `100`, and the new value generated represents a decrease in fitness. At this point, exploratory moves are recommenced in order to establish a new direction. The search continues in this fashion until the test data is found, or the current input vector cannot be improved because local moves do not offer an improved fitness. The latter case is a common problem for local search techniques - the tendency to converge prematurely at a sub–optimal solution. This may be, for example, at the top of locally optimal 'hills', or along ridges or plateaux in the fitness landscape, where there are no variation in fitness values. When this happens, the search is restarted at another randomly chosen point in the search space. The number of restarts is potentially unlimited and only restricted by the limit on the overall budget allowance of fitness evaluations permitted.

2.3 Evolutionary Testing (ET)

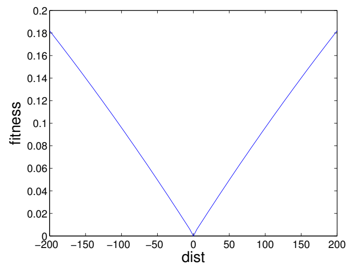
It is well known that local search techniques can suffer from the problem of becoming trapped in local optima. To overcome this problem many authors have considered global search techniques, most notably genetic algorithms [13, 14, 17, 20, 24], giving rise to the so-called evolutionary testing (ET) approach. The genetic algorithm (GA) used for evolutionary testing (ET) in this paper is based on the approach described by Wegener *et al.* [24]. GAs differ from local search techniques in that they maintain a population of candidate solutions, also referred to as 'individuals'. The Wegener model splits the overall population of 300 individuals into six competing subpopulations, which begin with 50 individuals each.

```
          double gradient_calc_radial_factor(
                  double dist, double offset,
                  double x, double y)
          {
              double r, rat;

(1)           if (dist == 0.0) {
(2)               rat = 0.0;
              } else {
(3)               offset = offset / 100.0;
(4)               r   = sqrt (x * x + y * y);
(5)               rat = r / dist;

(6)               if (rat < offset)
(7)                   rat = 0.0;
(8)               else if (offset == 1.0)
(9)                   rat = (rat >= 1.0) ? 1.0 : 0.0;
                      // ...
```

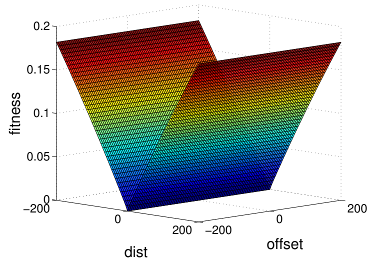
(a) Code snippet



(b) Approach level calculation for the true branch from node 8



(c) Fitness landscape for true branch from node 1, plotted for the `dist` input variable only



(d) Fitness landscape for true branch from node 1, plotted for `dist` and `offset` input variables only

Figure 1: Code from the `gimp` open source graphics package and corresponding fitness analysis

A GA is a loop of five main steps: fitness evaluation, selection, crossover, mutation and reinsertion. For selection, the Wegener model utilizes linear ranking. Linear ranking [27] sorts the population into fitness order, assigning a ranked fitness to each individual based on a selection pressure value $Z = 1.7$. Ranked fitnesses are allocated such that the current best individual in a population has a fitness of Z, the median individual a fitness of 1.0 and the worst candidate solution a fitness of $2-Z$. Once individuals have been assigned a fitness, a selection operator is applied to the population. The method used is stochastic universal sampling [1], where the probability of an individual being selected is proportionate to its fitness value. This type of selection favours 'fitter' individuals. Weaker individuals retain a selection chance, but with a relatively smaller probability. Individuals are then removed two-by-two at random from the selection pool for recombination. A discrete recombination [19] strategy is used to produce new 'offspring'. Discrete recombination operates such that offspring individuals receive 'genes' (*i.e.* input variable values) from either parent with an equal probability.

The next stage in the GA is to apply a mutation operation to the offspring, based on the breeder genetic algorithm [19]. Mutation is applied with a probability p_m of $1/len$, where len is the length of the input vector. The mutation operator applies a different mutation step size $10^{-p}, 1 \leq p \leq 6$ for each of the six subpopulations. A mutation range r is defined for each input parameter by the product of p and the domain size of the parameter. The 'mutated' value of an input parameter can thus be computed as $v_i = x_i \pm r_i \cdot \delta$. Addition or subtraction is chosen with an equal probability. The value of δ is defined to be $\sum_{x=0}^{15} \alpha_x \cdot 2^{-x}$, where each α_x is 1 with a probability of 1/16 else 0. If a mutated value is outside the allowed bounds of a variable, its value is set to either the minimum or maximum value for that variable.

The final stage of the GA cycle is the reinsertion process. An elitist reinsertion strategy is used. The top 10% of the current generation is retained and the remaining individuals discarded and replaced by the best offspring.

Subpopulations compete with one another for a slice of individuals. A progress value, p, is computed for each population at the end of a generation. This value is obtained using the formula $0.9 \cdot p + 0.1 \cdot rank$. The average fitness $rank$ for a population is obtained by linearly ranking its individuals as well as the populations amongst themselves (again with $Z = 1.7$). After every 4 generations, the populations are ranked according to their progress value and a new slice of the overall population is computed for each, with weaker subpopulations transferring individuals to stronger ones. However, no subpopulation can lose its last 5 individuals, preventing it from dying out. Finally, a general migration of individuals takes place after every 20^{th} generation, where subpopulations randomly exchange 10% of their individuals with one another.

3. INPUT DOMAIN REDUCTION

For search-based test data generation, the search space is the input domain of the test object under consideration. The input domain includes global variables and the formal parameters to the function containing the structure of interest. Each uncovered branch is taken in turn as the search target. As such, it is possible that not every input variable will be responsible for determining whether each branch will be covered or not. This information can be determined through static analysis. For example, in Figure 1, only the input variable `dist` is relevant for the predicate at node 1. Thus, when attempting to cover branches from this node, search effort on the values of the other variables - `offset`, `x` and `y` is wasted, since these variables cannot influence coverage of the branch. Removing these variables from the search space could potentially improve the performance of the search.

In some instances, no reduction is possible, because all input variables can potentially play a part in deciding whether the target branch is traversed. For example, the predicate at node 6, `rat < offset`, is dependent on all input values. The variable `rat` is assigned using the input variable `dist` and the internal variable `r` at node 5. The internal variable `r` is previously assigned using the input values of `x` and `y` at node 4. Elsewhere in the program, node 8 is control dependent on the outcome at node 6. Thus it is automatically dependent on all the input variables that node 6 is dependent upon - *i.e.* the entire input vector. In order to remove irrelevant input variables from the search, the variable dependence tool VADA [3, 9] was used.

4. THEORETICAL ANALYSIS

The following sections lay theoretical foundations for the impact of input domain reduction via irrelevant input variable removal on each of the search techniques.

4.1 Random Testing

The basis for the analysis of RT is the probability of generating an input for a target. This can be expressed as the cardinality of the set of input vectors which will execute the target, $card(E)$, divided by the cardinality of the input domain $card(D)$. This can be illustrated for the true branch from node 1 of Figure 1. Suppose each variable has the range 0.0 to 999.9 (with an accuracy of 0.1). The value of 0.0 must be found for `dist`, but any value can be chosen from the domain of the other input variables. Thus $card(E) = 1 \times 10000 \times 10000 \times 10000$, or 10^{12}. With $card(D) = 10^{16}$, the probability of generating test data in one trial is 1/10000. On average then, 10000 trials will be required to find the desired input.

Perhaps surprisingly, input domain reduction, in the form of irrelevant input variable exclusion, will not affect the probability of coverage of a target using random search. Because excluded input variables cannot affect whether the target is covered, any of their values can be selected. Thus removing them reduces $card(D)$ and $card(E)$ in equal proportion. Therefore, input variable exclusion will not have any significant effect on the performance of random search. To illustrate, reconsider the true branch from node 1 of Figure 1, for which `dist` is the only relevant variable. If the irrelevant input variables are ignored, $card(D)$ is only 10000, but there is only one value of `dist` which will execute the target. Thus the probability of generating test data in a single trial remains at 1/10000.

4.2 Hill Climbing

HC begins with a randomly generated starting point. As established in the previous section, this initial solution is no more likely to have hit the target with or without irrelevant input variable exclusion from the search space.

Furthermore, it can be shown that the randomly generated starting point will be no nearer the required test data than it would have been under normal circumstances. Figure 1(c) plots the fitness landscape for the true branch of the `gimp` function for the relevant variable `dist` only. Contrast this with the fitness landscape in Figure 1(d), plotted for `dist` and an irrelevant input variable to the branch predicate - `offset`. The irrelevant variable simply introduces an extra dimension in the landscape with no variation in fitness when the value of the relevant input is fixed.

However, a performance increase is possible for HC as the search commences from its initial position. HC performs exploratory moves on each input variable in turn. Thus, the exclusion of irrelevant variables will save these wasteful moves - moves that cannot result in an improvement in fitness. Two moves will be saved per irrelevant variable - the increase and decrease moves described in Section 2.2. If a 'cycle' c of the HC search is defined as a complete sweep of moves through all the variables of the input vector, and $irrelevant$ is the set of irrelevant variables for the current branch, the expected performance improvement per cycle i can be defined as $card(irrelevant) * 2$. Thus it is possible to place bounds on the overall improvement, in terms of the number of required fitness evaluations, on the basis of the number of times the input vector has been cycled through, c, in the test data search: $i * (c - 1) \leq improvement \leq i * c$.

However, since the number of cycles undertaken in a search is dependent on the starting position and the underlying shape of the fitness landscape for a branch, it is impossible to predict the number of cycles that will be undertaken by the search in advance. Thus although the theory suggests that HC will be subject to an improved performance as a result of search space reduction, the gain can only be assessed via an empirical study.

4.3 Evolutionary Testing

GAs begin with a random population of solutions. As described for RT and HC, randomly generated solutions are neither more likely to execute the target nor to be nearer the target in the search space as a result of input variable exclusion. Thus the population aspect of GAs yields no improvement with respect to the form of search space reduction considered here.

GAs make progress towards a solution using mutation and crossover. As established in the last section and throughout this paper, changing the value of an irrelevant input variable can never result in fitness improvement. Thus, exclusion of irrelevant variables can result in performance improvements for GAs because the mutation operator will be concentrating its effort on relevant variables only.

The average number of mutations per individual is the mutation probability p_m, which is $1/len$, where len is the length of the input vector. Thus the number of wasted mutations w is on average $1/(len - card(irrelevant))$. Therefore the average improvement for a search as a result of irrelevant variable removal can be stated as $g * p * w$, where g is the number of generations undertaken by the search, and p is the size of the population.

Unfortunately however, the number of generations required to reach a solution is as hard, if not harder, to predict than the number of cycles for HC. Not only must the underlying fitness landscape be taken into consideration, but also probabilistic effects of mutation, mutation size, crossover, and also competition and migration between subpopulations. Therefore, as for HC, although the theory suggests search improvement from irrelevant input variable exclusion, only an empirical study can establish the extent of its benefit.

Table 1: Details of the test objects

Test Object / Function	Lines of Code	No. of Branches	Approx. Domain Size (10^x)
defroster			
Defroster_main	250	56	69
f2			
F2	418	24	81
gimp-2.2.4			
gradient_calc_radial_factor		6	21
gradient_calc_square_factor		6	21
gradient_calc_conical_sym_factor		8	31
gradient_calc_conical_asym_factor		6	31
gradient_calc_bilinear_factor		6	34
gradient_calc_spiral_factor		8	37
gradient_calc_linear_factor		8	31
Total	867		
spice			
cliparc		64	44
clip_to_circle		42	30
Total	269		
tiff-3.8.2			
TIFF_SetSample		14	10
TIFF_GetSourceSamples		18	15
PlaceImage		16	38
Total	182		
synthetic			
synthetic_example	138	78	126
Grand Total	**2,124**	**360**	

5. EMPIRICAL STUDY

The empirical study in this paper analyses the impact of removing irrelevant variables from the search space for each search technique. The four research questions to be addressed by the empirical study are as follows:

Research Question 1 - Impact on RT. The theoretical analysis of Section 4 predicts there will be no effect on removing irrelevant input variables on the performance of the search. Does this hypothesis hold?

Research Question 2 - Impact on HC. It is predicted that removing irrelevant input variables will have an impact on search performance. Is this the case, and what is the performance increase?

Research Question 3 - Impact on ET. Similarly, it is predicted that removing irrelevant input variables will have an impact on search performance for the GA. Is this true, and if so, what is the performance increase?

Research Question 4 - Relative Impact. What is the relative impact of irrelevant input variable removal from the search space for HC compared to ET?

The empirical study was performed on 360 branches, drawn from 6 different programs, two of which were provided by DaimlerChrysler, while the other four are open source. Details of the subjects used in the empirical study can be seen in Table 1.

The minimum and maximum values of the input variables, along with their accuracy in the case of floating point variables, were specified for the search process. From this information, the input domain size - the search space size where reduction is not performed - can be computed. Input domains ranged from approximately 10^{10} to 10^{81} for the real world examples. Each separate branch denotes a separate search problem, for which the size of the search space is one of the more important factors in determining the difficultly of the problem. These are large search problems; 10^{80} is widely believed to be the approximate size of the observable universe in cubic metres.

The programs `f2` and `defroster` are industrial case studies provided by DaimlerChrysler. An S–Class Mercedes car has over 80 such embedded controllers, which, taken together represent approximately 0.5GB of object code. The two systems used in this study are production code for engine and rear window defroster control systems. The code is machine generated from a design model of the desired behaviour. As such, it is not optimized for human-readability, making manual determination of search space non–trivial. The test objects are therefore ideal candidates for automated search-space reduction strategies.

To complement the industrial examples, three open-source case studies were selected. Seven functions were selected from `gimp-2.2.4`, a graphics manipulation package. `spice` is an open source general purpose analogue circuit simulator. Two functions were tested, which were clipping routines for the graphical front-end. `tiff-3.8.2` is a library for manipulating images in the Tag Image File Format (TIFF). The functions tested comprised routines for placing images on pages, and the building of 'overview' compressed sample images. Finally, `synthetic` was a test problem especially designed for the experiments. It contains 20 input variables. There are 39 decision nodes, the first using the first input variable, the second using the first two input variables, and so on up to the twentieth node, which uses the entire vector. The remaining 19 nodes also incrementally include variables from the input space, but this time starting from the end of the vector - node 21 requires only the last variable, whilst node 39 uses the last 19. For `synthetic`, the search space ranges up to 10^{126}. This synthetic program is included to allow experiments with a range of search space sizes of the same program.

The test data generation experiments were performed 60 times for each combination of branch and search method, with and without irrelevant input variable removal. If test data was not found to cover a branch after 100,000 fitness evaluations, the search was terminated.

The success or failure of each search was recorded, along with the number of test data evaluations required to find the test data, if the search was successful. From this the 'success rate' of each branch can be calculated - the percentage of the 60 runs for which test data to execute the branch was found. The 60 runs were performed using an identical list of fixed seeds for random number generation, so as to provide a basis for assessment with tests for statistical significance using paired t-tests. A confidence level of 99% was applied. Such tests are necessary to provide robust results in the presence of the inherently stochastic behaviour of the search algorithms.

6. RESULTS

Research Question 1 - Impact on RT. The results provide evidence to support the claim that RT is unaffected by the removal of irrelevant variables from the search space. Although the results of the empirical study for RT exhibit variation before and after the application of search space reduction, no obvious relationship was found between input domain size and the performance of RT. The variation is merely a result of the shortened input vector lengths, which result in different input vectors being generated from the same sequence of random numbers.

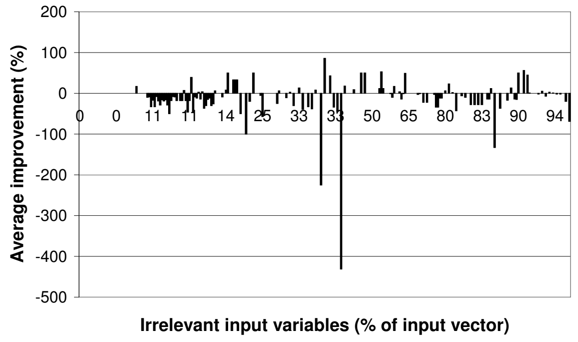


Figure 2: Average Improvement for RT

In particular, no branches were found for which there was a statistically significant difference in the numbers for fitness evaluations before and after reduction. There is, therefore, no evidence to suggest that the effort required for random testing is reduced by removing irrelevant variables. The lack of improvement for RT contrasts strongly with those for HC and ET, as can be seen by comparing Figure 2 for RT with Figures 3 and 5 for HC and ET. These figures show the percentage difference between average numbers of test data evaluations for each branch recorded before and after search space reduction. The branches are ordered along the $x-$axis according to the proportion of the input vector removed as a result of irrelevant variable analysis. Some $x-$values appear more than once along the $x-$axis simply because several branches share the same proportion of irrelevant variables. In Figure 2, no trend exists to suggest an increased (or even decreased) performance of RT as the percentage of irrelevant variables increases and the search space becomes smaller. Further inspection of the results showed that the existence of spikes in the graph is because of the comparison of a small number of successful trials between the reduced and unreduced versions of the experiments.

In terms of the achievement of branch coverage, RT successfully found test data for 281 distinct branches in at least one of the sixty runs before reduction, and 271 afterwards. However, deeper inspection of the results showed that the ten additional branches in the unreduced search space had been covered with an extremely low success rate (corresponding to a successful search in at most 2 of the 60 runs).

Table 2 records branches for which the success rates fluctuate by +/- 5% (equivalent to 3 runs). This table also supports the observation that removal of irrelevant variables has little or no impact upon the effectiveness of RT; the table shows that success rate is improved by 5% or more for four branches. However, performance is also worsened by 5% or more for a further four branches.

In conclusion, no relationship is found between input variable reduction and the performance of RT. This empirical finding supports the hypothesis for RT set out in Section 5.

Research Question 2 - Impact on HC. The effort required to find test data using HC was found to be significantly reduced by the removal of irrelevant variables from the search space. However, though effort was reduced, effectiveness was not; the search did not find test data where it previously could not.

In total, 82 branches were found for which the two sets of numbers of test data evaluations (before and after reduction) were found to be significantly different. For all of these branches, the performance was improved with reduction. 38 of these branches were never covered by random search and appear in Table 3.

Figure 3 shows the performance difference with HC against search space size for all branches considered in this experiment. Although there are a number of cases of improvement, some branches exhibited no improvement at all, simply because the test data was easily found by random search. Search space reduction is superfluous in such instances. There are some spikes where HC appears to have performed worse with reduction. Again, a deeper inspection of the results showed that the existence of these spikes is due to the comparison of a small number of successful trials between the reduced and unreduced versions of the experiment. Overall, the graph shows the existence of a general trend showing the improvement of HC performance as the percentage of irrelevant variables removed from the search increases. In some cases, the relative improvement is as much as 80%.

However, despite the decrease in effort occasioned by search space reduction, there is no improvement in effectiveness. That is, only a single extra branch was covered after reduction (*i.e.* an extra branch for which test data was found in at least one of the sixty runs), compared to the 354 branches covered originally. Furthermore, there was also little change in the success rate for each branch. Table 2 shows that the success rate was improved for two branches but worsened for two others.

Figure 4 shows the results for the synthetic example. This figure shows the range and mean value of the number of fitness evaluations (on the $y-$axis) against the increasing ability to remove irrelevant variables (on the $x-$axis). Note that the figure uses a logarithmic scale. Because of the order of consideration of variables by the local search algorithm, there is an inter-play between the location of irrelevant variables in the input vector and the effect of removing these variables on the number of fitness evaluations required to cover the branch. If irrelevant variables appear at the end of the input vector (left hand side of Figure 4), then they will remain unencountered during at least one cycle of exploratory moves through the input vector. Thus, the impact of their removal from the search is much less pronounced. However, with irrelevant variables appearing at the beginning of the vector the improvement is very noticeable (right hand side of Figure 4).

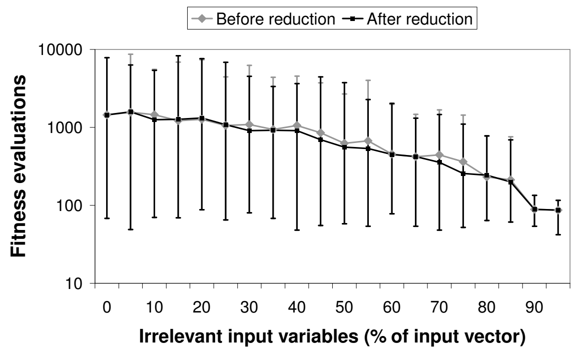
In conclusion, the results, in particular the significant improved performance for 82 branches, provide evidence to support the hypothesis that removing irrelevant input variables from the search space has a positive impact on the effort required for test data generation using HC.

Research Question 3 - Impact on ET. As for HC, the results for ET support the claim that the search is improved by removing from the search space input variables that are irrelevant to each branch. Effectiveness, however, is not significantly altered.

In total, 86 branches were found for which the removal of irrelevant variables resulted in a statistically significant reduction in effort. 37 of these branches were never covered by random search. Details appear in Table 3. Figure 5 shows the performance difference with ET against search space size for all branches considered in this experiment. The figure suggests the existence of a trend: as the percentage of irrelevant variables increases, and the search space becomes

Table 2: **Success rate fluctuation before and after irrelevant variable removal from the search space. Branches are only listed if a +/-5% change was experienced for either RT, HC or ET. The 'Irrel. Vars' column expresses the percentage of variables of the program's input vector that are irrelevant for the branch**

Test Object	Function (Branch ID)	Irrel. Vars	RT Before/After	HC Before/After	ET Before/After
defroster	Defroster_main (12T)	85%	**48%/55% (6%)**	100%/100% (0%)	100%/100% (0%)
	Defroster_main (14T)	90%	**48%/60% (11%)**	100%/100% (0%)	100%/100% (0%)
	Defroster_main (16T)	80%	**36%/28% (-8%)**	100%/100% (0%)	100%/100% (0%)
	Defroster_main (20T)	80%	**36%/28% (-8%)**	100%/100% (0%)	100%/100% (0%)
	Defroster_main (31F)	65%	**25%/33% (8%)**	100%/100% (0%)	100%/100% (0%)
	Defroster_main (36F)	60%	**15%/21% (6%)**	100%/100% (0%)	100%/100% (0%)
	Defroster_main (49F)	65%	0%/0% (0%)	100%/100% (0%)	**90%/100% (10%)**
	Defroster_main (68T)	80%	**75%/68% (-6%)**	100%/100% (0%)	100%/100% (0%)
f2	F2 (11F)	82%	0%/0% (0%)	100%/100% (0%)	**63%/86% (23%)**
	F2 (15T)	94%	90%/85% (-5%)	100%/100% (0%)	**30%/56% (26%)**
	F2 (32T)	94%	80%/85% (5%)	**8%/78% (70%)**	**0%/16% (16%)**
gimpdrawableblend	gradient_calc_spiral_factor (1T)	85%	**31%/40% (8%)**	100%/100% (0%)	100%/100% (0%)
spice	clip_to_circle (36T)	14%	0%/0% (0%)	**63%/53% (-10%)**	3%/1% (-1%)
	cliparc (22T)	33%	0%/0% (0%)	**11%/5% (-6%)**	0%/0% (0%)
	cliparc (67T)	11%	**98%/90% (-8%)**	98%/100% (1%)	100%/100% (0%)
tiff	PlaceImage (16T)	54%	1%/3% (1%)	**8%/18% (10%)**	38%/43% (5%)
	PlaceImage (20T)	54%	0%/0% (0%)	1%/1% (0%)	**15%/35% (20%)**
	TIFF_GetSourceSamples (15T)	33%	**10%/1% (-8%)**	100%/100% (0%)	100%/100% (0%)
synthetic	synthetic_example (23T)	85%	1%/1% (0%)	100%/100% (0%)	**90%/100% (10%)**
	synthetic_example (24T)	80%	3%/3% (0%)	100%/100% (0%)	**93%/100% (6%)**
	synthetic_example (4T)	80%	3%/3% (0%)	100%/100% (0%)	**90%/100% (10%)**



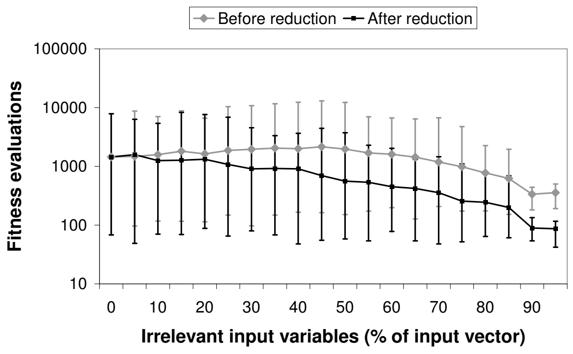


Figure 4: Average fitness evaluations for HC and the synthetic example. Left: irrelevant input variables featuring at the start of input vector. Right: irrelevant variables featuring at the end

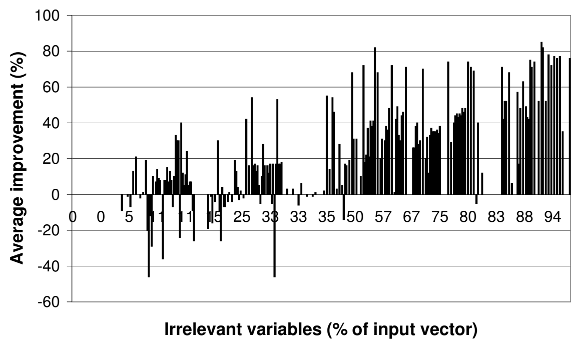


Figure 3: Average Improvement for HC

smaller, the performance of ET improves. Furthermore, for the synthetic example, the results also point to a reduction in effort with increased ability to remove irrelevant variables (Figure 6).

Table 2 shows that success rate was improved for several branches by more than 5%. However, the results did not suggest a major difference in terms of coverage: test data was found for 349 branches by at least one of the sixty runs before reduction, whilst 350 branches were covered after reduction.

In conclusion, the results provide evidence to support the hypothesis that removing irrelevant input variables from the search space has a positive impact on ET.

Research Question 4 - Relative Impact. The empirical study showed no evidence for a relationship between input variable reduction and performance of RT, but did show an increase in performance for HC and ET. Figure 7 compares improvement for HC and ET. First the change in average performance was found for each branch using each technique by subtracting average performance without reduction from average performance with reduction. The average performance difference for HC was then subtracted from that of ET for each branch.

The bar chart shows that most of the branches appear on the positive $y-$axis, indicating that ET tended to benefit

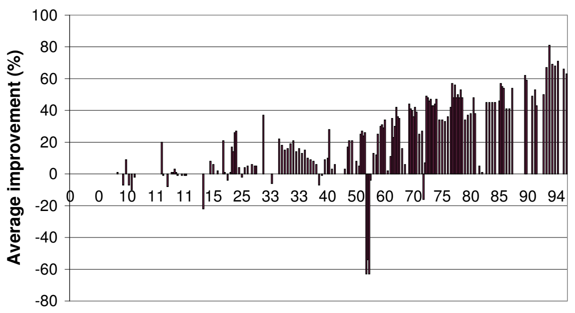


Figure 5: Average Improvement for ET

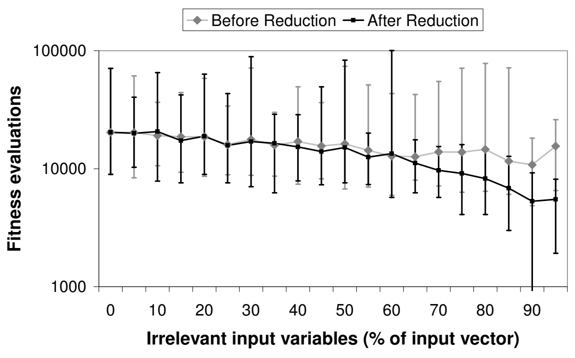


Figure 6: Average fitness evaluations for ET

more from the reduced search space compared to HC. The difference in the number of evaluations can also be observed for significant branches in Table 3, which portrays an order of magnitude difference. Since HC is more efficient in covering test targets for which it can generate test data, improvement in terms of the actual number of evaluations is lower. Figures 3 and 5 both show, however, that in terms of relative improvement, both techniques can experience performance increases up to a level of at least 80%. A further statistical test was devised for branches for which that ET and HC had performed with significantly with irrelevant variable removal. The improvement with reduction, in terms of fitness evaluations, was calculated and normalized for both HC and ET. At a confidence level of 99%, paired t-tests were applied to the two sets of normalized improvement values for each branch and each technique. 33 branches were found to be significant. 9 of these branches, which were never covered by RT, appear in boldface in Table 3. To conclude, therefore, the evidence suggests that ET has more to gain from reduction of the search space than HC, in terms of both actual and relative performance.

7. THREATS TO VALIDITY

Naturally there are threats to validity in any empirical study such as this. This section briefly outlines the potential threats to validity and how they were addressed. The hypotheses studied in this paper concerned relationships between the size of the search space and the performance of search algorithms employed for branch coverage. Therefore,

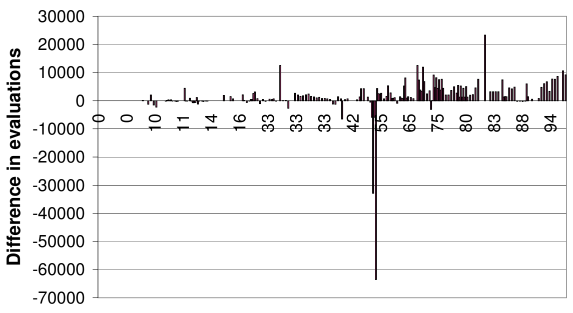


Figure 7: Difference in improvement for HC and ET

one issue to address is the so-called *internal validity*, *i.e.*, to check whether there has been a bias in the experimental design which could affect the causal relationship under study.

For determining the appropriate size of the search space by eliminating contributions due to irrelevant variables, the VADA tool was used. VADA has been used and tested by DaimlerChrysler as part of their Evolutionary Testing System (ETS) - an industrial application for test data generation. However, VADA is, nonetheless a research prototype tool. Therefore, a manual check was performed on results obtained to ensure that the tool had correctly identified the variables that could potentially affect the predicates of interest.

Another potential source of bias comes from the inherent stochastic behaviour of the metaheuristic search algorithms under study. The most reliable (and widely used) technique for overcoming this source of variability is to perform tests for statistical significance on a sufficiently large sample of result data. Such a test is required whenever one wishes to make the claim that one technique produces superior results to another. A set of results are obtained from a set of runs (essentially sampling from the population of random number seeds).

To show that one technique is superior to another, a test to see if there is a statistical significant difference in the means is performed. For the results reported upon here, the t–test was used with the confidence level set at 99%. In order to ensure normality of the sample means, an assumption of this statistical test, it is important to have a sample size of at least 30. To ensure that this constraint was comfortably achieved, each experiment was repeated 60 times.

Another source of bias comes from the selection of the programs to be studied. This impacts upon the *external validity* of the empirical study. That is, the extent to which it is possible to generalise from the results obtained. Naturally, it is impossible to sample a sufficiently large set of programs such that the full diversity of all possible programs could be captured. The rich and diverse nature of programs makes this an unrealistic goal. However, where possible, a variety of programming styles and sources have been used. The study draws upon code from real world programs, both from industrial production code and from open source. Furthermore, it should be noted that the number of test problems considered is 360, providing a relatively large pool of results from which to make observations.

Table 3: Branches for which the two sets of test data trials (before and after irrelevant variable removal from the search space) differed significantly using paired *t*-tests with a confidence level of 99%. Input domain reduction is expressed as the percentage of variables of the program's input vector that are irrelevant for the branch ('Irrel. Input Vars'). AE refers to the average number of test data evaluations for the branch, whilst SD is the standard deviation. 'Obs. Sig.' is the observed significance level (*p*-value). Entries shown in boldface are statistically significant. 'Imp. Obs. Sig.' is the significance level observed after comparing the sample means of the normalized average improvement for HC and ET in applying irrelevant variable removal

Test Object / Function	Irrel.	HC			ET			Imp.
(Branch ID)	Input	Before	After	Obs.	Before	After	Obs.	Obs.
	Vars	AE/SD	AE/SD	Sig.	AE/SD	AE/SD	Sig.	Sig.
defroster								
Defroster_main (12F)	85%	**281 / 56**	**82 / 17**	**0.000**	**16331/4141**	**8738/2408**	**0.000**	**0.000**
Defroster_main (14F)	90%	**64 / 2**	**36 / 2**	**0.000**	**9687/4248**	**3682/1502**	**0.000**	**0.002**
Defroster_main (16F)	80%	**71 / 12**	**40 / 4**	**0.000**	**9688/4810**	**4200/1763**	**0.000**	0.487
Defroster_main (18F)	75%	**91 / 12**	**60 / 5**	**0.000**	**18849/7621**	**9700/3399**	**0.000**	0.185
Defroster_main (18T)	75%	**71 / 12**	**44 / 5**	**0.000**	**9688/4810**	**5001/2109**	**0.000**	0.962
Defroster_main (20F)	80%	**69 / 12**	**40 / 4**	**0.000**	**9453/4021**	**4200/1763**	**0.000**	0.286
Defroster_main (22F)	75%	**92 / 13**	**60 / 5**	**0.000**	**17913/7927**	**9700/3399**	**0.000**	0.921
Defroster_main (22T)	75%	**69 / 12**	**44 / 5**	**0.000**	**9453/4021**	**5001/2109**	**0.000**	0.952
Defroster_main (24F)	70%	**114 / 13**	**84 / 7**	**0.000**	**28413/11671**	**15854/5237**	**0.000**	0.120
Defroster_main (24T)	70%	**92 / 13**	**68 / 6**	**0.000**	**17913/7927**	**10549/3739**	**0.000**	0.277
Defroster_main (30F)	80%	**71 / 12**	**40 / 4**	**0.000**	**8895/3962**	**4448/2033**	**0.000**	0.777
Defroster_main (35F)	60%	**64 / 12**	**45 / 6**	**0.000**	**9413/3526**	**6596/2455**	**0.000**	0.240
Defroster_main (39F)	55%	**80 / 12**	**66 / 9**	**0.000**	**17483/6677**	**13135/4419**	**0.000**	0.898
Defroster_main (39T)	55%	**64 / 12**	**49 / 9**	**0.000**	**9413/3526**	**6876/2349**	**0.000**	0.621
Defroster_main (40F)	55%	**94 / 44**	**60 / 20**	**0.000**	**10199/4320**	**7794/2543**	**0.000**	**0.004**
Defroster_main (40T)	55%	**74 / 19**	**58 / 13**	**0.000**	**9958/3507**	**7322/2662**	**0.000**	0.631
Defroster_main (44F)	75%	**94 / 12**	**62 / 5**	**0.000**	**17399/7595**	**9879/3508**	**0.000**	0.829
Defroster_main (44T)	75%	**71 / 12**	**46 / 5**	**0.000**	**8895/3962**	**5059/2319**	**0.000**	0.538
Defroster_main (45F)	70%	**81 / 19**	**51 / 7**	**0.000**	**9645/4070**	**5788/2252**	**0.000**	0.363
Defroster_main (45T)	70%	**86 / 16**	**52 / 8**	**0.000**	**9568/4056**	**6134/2454**	**0.000**	0.042
Defroster_main (48F)	70%	**116 / 12**	**83 / 6**	**0.000**	**28181/12233**	**16252/6150**	**0.000**	0.663
Defroster_main (48T)	70%	**94 / 12**	**67 / 6**	**0.000**	**17426/7602**	**10586/4341**	**0.000**	0.561
Defroster_main (49F)	65%	**110 / 16**	**74 / 9**	**0.000**	**17337/6958**	**12050/8754**	**0.003**	0.171
Defroster_main (49T)	65%	**104 / 18**	**74 / 9**	**0.000**	**19223/10255**	**11074/4551**	**0.000**	0.846
Defroster_main (55F)	80%	**73 / 12**	**40 / 4**	**0.000**	**9484/4118**	**4448/2033**	**0.000**	0.809
Defroster_main (60F)	75%	**93 / 12**	**62 / 5**	**0.000**	**17506/6391**	**9879/3508**	**0.000**	0.804
Defroster_main (60T)	75%	**73 / 12**	**46 / 5**	**0.000**	**9484/4118**	**5059/2319**	**0.000**	0.739
Defroster_main (61F)	25%	78 / 16	80 / 16	0.159	**9844/4248**	**7243/3097**	**0.000**	0.086
Defroster_main (61T)	25%	93 / 19	91 / 20	0.344	**11487/7682**	**8356/3712**	**0.008**	0.648
Defroster_main (63F)	20%	81 / 19	84 / 20	0.275	**9926/4244**	**7812/3218**	**0.000**	0.042
f2								
F2 (11F)	82%	**1452 / 1195**	**875 / 619**	**0.000**	**62606/12635**	**38712/8528**	**0.000**	0.213
gimp-2.2.4								
gradient_calc_conical_asym_factor (3F)	50%	**238 / 36**	**164 / 27**	**0.000**	**20691/4775**	**16350/3792**	**0.000**	**0.007**
gradient_calc_conical_sym_factor (3F)	50%	**238 / 36**	**164 / 27**	**0.000**	**20691/4775**	**16350/3792**	**0.000**	**0.007**
gradient_calc_spiral_factor (3F)	57%	**238 / 36**	**164 / 27**	**0.000**	**21729/4464**	**16350/3792**	**0.000**	0.068
spice								
clip_to_circle (4F)	42%	**137 / 21**	**99 / 16**	**0.000**	11943/2896	11220/2420	0.158	**0.000**
tiff-3.8.2								
TIFF_GetSourceSamples (7T)	33%	79 / 41	84 / 57	0.375	**11624/3268**	**9233/2984**	**0.000**	0.011
synthetic								
synthetic_example (25T)	75%	**985 / 879**	**257 / 203**	**0.000**	**13515/10769**	**9111/2450**	**0.003**	**0.000**
synthetic_example (28T)	60%	**1607 / 1481**	**451 / 416**	**0.000**	12221/5665	11943/3707	0.533	0.040
synthetic_example (29T)	55%	**1693 / 1710**	**536 / 487**	**0.000**	14367/6848	12527/3318	0.104	0.744
synthetic_example (30T)	50%	**1962 / 2093**	**560 / 591**	**0.000**	16011/10343	15134/10028	0.707	**0.000**
synthetic_example (32T)	40%	**1993 / 2111**	**909 / 831**	**0.000**	17077/8346	15329/4157	0.136	0.703
synthetic_example (33T)	35%	**2048 / 2092**	**922 / 847**	**0.000**	16347/6247	16443/5315	0.973	0.774
synthetic_example (3T)	85%	214 / 163	201 / 143	0.534	**11674/11686**	**6831/1851**	**0.003**	**0.000**

Nonetheless, caution is required before making any claims as to whether these results would be observed on other programs, possibly from different sources and in different programming languages. As with all such experimental software engineering, further experiments are required in order to replicate the results contained here. However, the results show that there do indeed exist cases where there is a statistically significant relationship between search space reduction and improved performance of search algorithms for test data generation. They also provide evidence to support the claim that the more sophisticated genetic algorithm has more to gain from such search–based reduction than simpler hill climbing and that random search is unaffected.

8. RELATED WORK

Although the search space reduction question has been asked and answered for other search problems [6, 21] it has never been answered before for search-based test data generation. The question is a highly timely one, as the last ten years have experienced an explosion in work in the area, resulting in a survey by McMinn in 2004 [15]. The field began in 1976 with the work of Miller and Spooner [18], who applied numerical maximization techniques to generate floating point test data for paths. Korel was the first to apply the technique referred to as HC in this paper [12], whilst Xanthakis *et al.* were the first to apply GAs [28]. Harman and McMinn [10] applied a theoretical analysis and empirical comparison of RT, HC and ET for branch coverage. The present paper is the first to consider search space reduction for any of these techniques.

Korel proposed a method [12] closely related to the search space reduction technique proposed here, but for path coverage. In his approach, each input variable in the search underwent a risk analysis using an influences graph constructed using dynamic data-flow information. The value of an input variable would remain fixed if it was highly likely that changes would impact current segments of the path that were currently being traversed correctly, or, if the input variable did not affect the path at all. Unfortunately, however, the method was not empirically evaluated.

The VADA tool used in our experiments to reduce the search space produces dependence information as a by-product of static program slicing [26]. Source code analysis techniques such as symbolic execution [11] or abstract interpretation [7] could further support search space reduction by finding constraints or defining ranges on the input variables relevant to the branch under investigation. However, this remains a topic for future work.

9. CONCLUSIONS

This paper has theoretically and empirically evaluated the impact of removing irrelevant input variables from the search for test data for the coverage of individual program branches. The theoretical analysis predicted that this form of search space reduction would not have a significant effect on random testing, but could enhance the performance of more intelligent search techniques, such as hill climbing and genetic algorithms. An empirical study, performed on 360 branches from open source code and embedded controller production code supplied by DaimlerChrysler, was found to support these claims.

Acknowledgements

The authors would like to thank John Spooner (SAS UK) for statistical advice.

Mark Harman is supported by EPSRC Grants EP/D050863, GR/S93684 & GR/T22872, by EU grant IST-33472 (EvoTest) and also by the kind support of DaimlerChrysler Berlin and Vizuri Ltd., London. Youssef Hassoun and Kiran Lakhotia are funded by EU grant IST-33472 (EvoTest).

10. REFERENCES

[1] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the 2nd International Conference on Genetic Algorithms and their Application*, Hillsdale, New Jersey, USA, 1987. Lawrence Erlbaum Associates.

[2] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, 1990.

[3] D. W. Binkley and M. Harman. Analysis and visualization of predicate dependence on formal parameters and global variables. *IEEE Transactions on Software Engineering*, 30(11):715–735, 2004.

[4] L. C. Briand, J. Feng, and Y. Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. In 14^{th} *IEEE Software Engineering and Knowledge Engineering (SEKE)*, pages 43–50, Ischia, Italy, 2002.

[5] L. C. Briand, Y. Labiche, and M. Shousha. Stress testing real-time systems with genetic algorithms. In *Proccedings of the Genetic and Evolutionary Computation Conference (GECCO 2005)*, pages 1021–1028. ACM, 2005.

[6] S. Chen and S. Smith. Improving genetic algorithms by search space reductions. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999)*, pages 135–140. Morgan Kaufmann, 1999.

[7] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. *ACM SIGPLAN Notices*, 31(1):178–190, Jan. 2002.

[8] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, 1980.

[9] M. Harman, C. Fox, R. M. Hierons, L. Hu, S. Danicic, and J. Wegener. VADA: A transformation-based system for variable dependence analysis. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, pages 55–64, 2002. IEEE Computer Society Press.

[10] M. Harman and P. McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. *International Symposim on Software Testing and Analysis (ISSTA 2007)*, to appear, 2007.

[11] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[12] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[13] N. Mansour and M. Salame. Data generation for path testing. *Software Quality Journal*, 12(2):121–134, 2004.

[14] G. McGraw, C. Michael, and M. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001.

[15] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[16] P. McMinn. IGUANA: Input generation using automated novel algorithms. A plug and play research tool. *Technical Report, Department of Computer Science, University of Sheffield*, 2007.

[17] P. McMinn, M. Harman, D. Binkley, and P. Tonella. The species per path approach to search-based test data generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2006)*, pages 13–24, 2006. ACM.

[18] W. Miller and D. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223–226, 1976.

[19] H. Mühlenbein and D. Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm: I. continuous parameter optimization. *Evolutionary Computation*, 1(1):25–49, 1993.

[20] R. Pargas, M. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

[21] F. Schmiedle, R. Drechsler, and B. Becker. Exact routing with search space reduction. *IEEE Transactions on Computers*, 52(6):815–825, 2003.

[22] A. Schultz, J. Grefenstette, and K. Jong. Test and evaluation by genetic algorithms. *IEEE Expert*, 8(5):9–14, 1993.

[23] N. Tracey, J. Clark, K. Mander, and J. McDermid. Automated test-data generation for exception conditions. *Software Practice and Experience*, 30(1):61–79, 2000.

[24] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[25] J. Wegener and O. Bühler. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2004), Lecture Notes in Computer Science vol. 3103*, pages 1021–1028, Springer-Verlag, 2004.

[26] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[27] D. Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In J. D. Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 116–121, San Mateo, California, USA, 1989. Morgan Kaufmann.

[28] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. Application of genetic algorithms to software testing (Application des algorithmes génétiques au test des logiciels). In *5th International Conference on Software Engineering and its Applications*, pages 625–636, Toulouse, France, 1992.