The Species per Path Approach to Search-Based Test Data Generation

Phil McMinn
University of Sheffield
211 Portobello St
Sheffield
S1 4DP, UK
phil@dcs.shef.ac.uk

Mark Harman
King's College London
Strand, London
WC2R 2LS
UK
mark.harman@kcl.ac.uk

David Binkley
Loyola College
4501 North Charles Street
Baltimore
MD 21210-2699, USA
binkley@cs.loyola.edu

Paolo Tonella
ITC-irst
Via Sommarive, 18
38050 Povo
Trento, Italy
tonella@itc.it

ABSTRACT

This paper introduces the Species per Path approach to search-based software test data generation. The approach transforms the program under test into a version in which multiple paths to the search target are factored out. Test data are then sought for each individual path by dedicated 'species' operating in parallel. The factoring out of paths results in several individual search landscapes, with feasible paths giving rise to landscapes that are potentially more conducive to test data discovery than the original overall landscape.

The paper presents the results of two empirical studies that validate and verify the approach. The validation study supports the claim that the approach is widely applicable and practical. The verification study shows that it is possible to generate test data for targets with the approach that are troublesome for the standard evolutionary method.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Verification

Keywords: Automated test data generation, evolutionary testing, search-based software engineering, testability transformation

1. INTRODUCTION

Because generating test data by hand is tedious, expensive and error-prone, automated test data generation has remained a topic of interest for the past three decades. Several techniques have been proposed, including symbolic execution [7, 15], constraint solving [9, 22] and search-based approaches [18], also known as search-based testing.

A variety of search techniques have been proposed for structural test data generation: local search in the form of the alternating variable method [10, 17], simulated annealing [26, 27] and evolutionary algorithms [14, 23, 29]. This paper concentrates on the application of evolutionary algorithms, although the results are also generally applicable to other search methods.

Instead of attempting to ascertain test data from the static form of a program, search-based methods explore the input domain of a program by executing it and observing the results. The program is instrumented in order to feed information back to the search method, which helps guide it to the location of the test data. The information takes the form of a *fitness function*, which is computed using cues such as the values of variables appearing in branch predicates, or the branches executed through the program by some input data.

This paper exposes and addresses the *path problem* for search-based test data generators. The path problem occurs when the input domain for the program is largely dominated by paths through the program which cannot lead to execution of the target, potentially suffocating the fitness function and preventing the search from receiving adequate guidance to the required test data. The problem is experienced for the example program in Figure 1a and the coverage of the target T. Figure 1c shows a search 'landscape', a helpful visualization of the fitness function. The landscape is plotted in three dimensions, with two of the dimensions showing the values of the two input variables (`a` and `b`), with the third indicating the value of fitness for each pair of values. The landscape reveals that it will not be easy for the search to find test data for T. There is only a single input vector which causes it to be executed, and the fitness landscape around it is flat. The lack of variation in fitness values leaves the search with no clues with respect to direction in which it should proceed, resulting in a high likelihood of failure to find the test data.

This paper proposes the Species per Path (SpP) solution to the problem. The initial stage of the SpP approach is a program transformation which factors out several paths to the target. The search effort is then distributed amongst the paths, with several 'species' working in parallel, each dedicated to finding test data for an individual path. Where the path problem causes difficulties for standard evolutionary test data generators, the SpP approach is likely to improve search success rate. This is because the transformation allows each species to exploit additional fitness guidance tailored to each path.

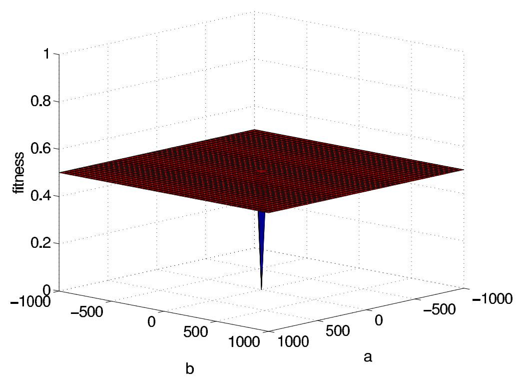
ISSTA'06, July 17–20, 2006, Portland, Maine, USA.

```
CFG
Node
 (s)      void example1(double a, double b)
          {
 (1)          if (a != 0) {
 (2)              b = 2;
              }

 (3)          if (b != 0) {
 (4)              a = 2;
              }

 (5)          if (a == 0 && b == 0) {
                  // T
              }
          }
```

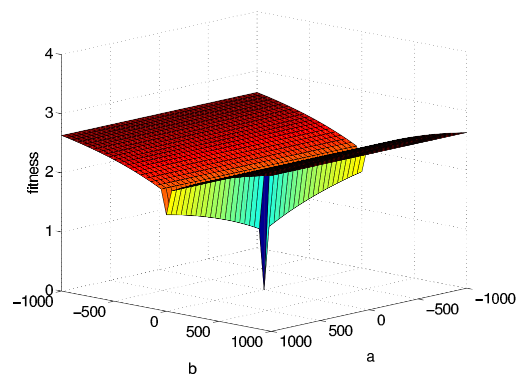
(a) Program Fragment

```
void example1_transformed(double a, double b)
{
    if (a != 0) {
        b=2;
        if (b != 0) {
            a=2;
            if (a==0 && b==0) {
                // T1
            }
        } else if (a==0 && b==0) {
            // T2
        }
    } else if (b != 0) {
        a=2;
        if (a==0 && b==0) {
            // T3
        }
    } else if (a==0 && b==0) {
        // T4
    }
}
```

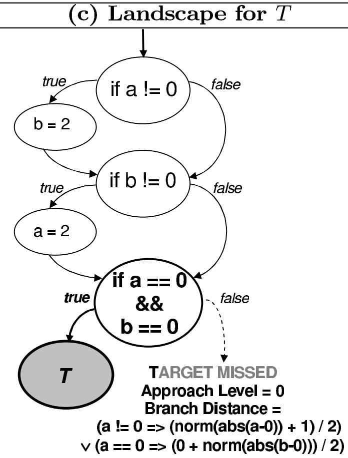
(b) Transformed program



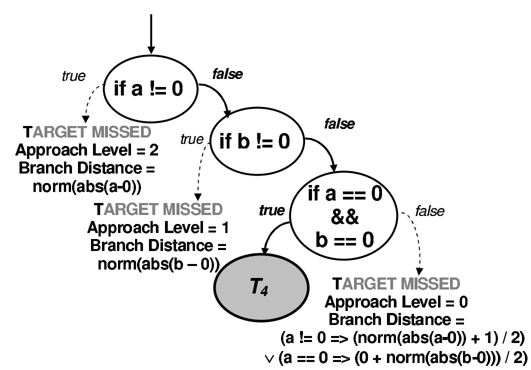
(c) Landscape for T



(d) Landscape for T_4



(e) Fitness computation for T



(f) Fitness computation for T_4

Figure 1: An example program fragment (a), its transformation (b), and the difference in fitness landscapes (c) and (d) and fitness computation (e) and (f). The landscape of T (c) in the original program is flat, providing the search with no direction to the location of the required test data. However, the SpP approach allows the search to explore the landscape of T_4 (d), which has a downward sloping surface providing the search with clear guidance as to the whereabouts of the necessary input data. This extra information is a result of extra guidance being included in the fitness function for T_4. The fitness computation for T is only dependent on one conditional (e), whereas the fitness computation for T_4 in the transformed version takes into account all the conditionals corresponding to a specific path in the original program.

Figure 1b shows the transformed program in which the four paths to T have been separated out. The effect this has on the landscape is visually evident in Figure 1d, which depicts the search landscape for target T_4, corresponding to a feasible path to T in the original program. The plateau is replaced with a directional surface that is significantly easier to search, even using a simple search technique such as gradient descent.

The primary contributions of this paper are the following:

1. It is shown how the natural evolutionary metaphor of 'species' [8] can be exploited to improve test data generation, by considering the paths to some target node to define inherently separate 'species of test data'.

2. The approach is justified both by a theoretical argument and by the results of an empirical study into its practical effectiveness for improving test data generation.

3. The paper also reports the results of a separate empirical study on a large set of open source programs, concerned with the validity of the approach; its applicability and algorithmic practicability.

The rest of this paper is organized as follows. Section 2 introduces the background behind the evolutionary test data generation approach, and some important basic concepts. Section 3 discusses the path problem in more detail, whilst Section 4 presents the proposed Species per Path solution. Section 5 presents experimental results to support the claims concerning the effectiveness of the approach with three studies which experience the path problem, one of which is taken from production code. Section 6 presents the results of an additional empirical study to validate the approach by showing that the method is widely applicable. Section 7 presents related work and Section 8 concludes with closing comments and future work.

2. BACKGROUND

This section presents an overview of the background to the SpP approach, including search-based test data generation, evolutionary algorithms, and computation of the fitness function for the coverage of individual structural targets. Before this, however, some basic concepts are reviewed.

2.1 Basic Concepts

A control flow graph (CFG) of a program is a directed graph $G = (N, E, s, e)$ where N is a set of nodes, E is a set of edges, and s and e are unique entry and exit nodes to the graph. Each node $n \in N$ corresponds to a statement in the program, with each edge $e = (n_i, n_j) \in E$ representing a transfer of control from node n_i to n_j. Nodes corresponding to decision statements (for example an `if` or `while` statement) are referred to as *branching nodes*. In the example of Figure 2, Nodes 1, 2 and 3 are all branching nodes. Outgoing edges from these nodes are referred to as *branches*. The branch executed when the condition at the branching node is true is referred to as the *true branch*. Conversely, the branch executed when the condition is false is referred to as the *false branch*. The predicate determining whether a branch is taken is referred to as a *branch predicate*. The branch predicate of the *true* branch from branching Node 1 in the program of Figure 2 is '`a >= b`'. The false branch predicate is '`a < b`'. A *path* through a CFG is a sequence of nodes $P = \langle n_1, n_2, \ldots n_m \rangle$ such that for each $i, 1 \leq i < m, (n_i, n_{i+1}) \in E$.

Control dependence [11] is used to describe the reliance of a node's execution on the outcome at previous branching nodes. For a program node i with two exits (for example, an `if` statement), program node j is control dependent on i if one exit from i always results in j being executed, while the other exit may not result in j being executed. In Figure 2, Node 2 is control dependent on Node 1, and Node 3 is control dependent on Node 2. Node 3 is not *directly* control dependent on Node 1. However, Node 3 is *transitively* control dependent on Node 1. For structured programs, control dependence reflects the program's nesting structure.

2.2 Search-Based Test Data Generation

Evolutionary algorithms [31] combine characteristics of genetic algorithms and evolution strategies, using simulated evolution as the model of a search method, employing operations inspired by genetics and natural selection. Evolutionary algorithms maintain a population of candidate solutions referred to as *individuals*. Individuals are iteratively recombined and mutated in order to evolve successive generations of potential solutions. The aim is to generate 'fitter' individuals within subsequent generations (*i.e.*, better candidate solutions). This is performed by a *recombination* operator, which forms offspring from the components of two parents selected from the current population. The new offspring form part of the new generation of candidate solutions. Mutation performs low probability random changes to solutions, introducing new genetic information into the search. At the end of each generation, each solution is evaluated for its fitness, using a problem-specific *fitness function*. Using fitness values, the evolutionary search decides whether individuals should survive into the next generation or be discarded.

In applying evolutionary algorithms to structural test data generation, the 'individuals' of the search are input vectors. The fitness function is derived from the context of the current structure of interest in the program. The fitness function is to be minimized by the search: thus, lower numerical values represent 'fitter' input vectors that are closer to executing the target structure. When a zero fitness value has been found, the required test data has also been found.

The fitness function is made up of two components – the *approach level* and the *branch distance*. The approach level measures how close an input vector was to executing the current structure of interest, on the basis of how near the execution path was to reaching it in the program's control flow graph. Central to this metric is the notion of a *critical branching node*. A critical branching node is simply a branching node with an exit that, if taken, causes the target to be missed. In other words, the set of critical branching nodes is the set of nodes on which the target structure is control dependent (either directly or transitively). In Figure 2, Nodes 1, 2 and 3 are all critical branching nodes with respect to the target. The approach level for an individual, as seen in Figure 2, is the number of critical branching nodes left unencountered by the execution path taken through the program.

At the point where control flow diverges away from the target, the *branch distance* is calculated. The branch distance reflects how 'close' the alternative branch from the last

```
CFG Node
(s)     void example(int a, int b, int c, int d)
        {
(1)       if (a >= b)
          {
(2)         if (b <= c)
            {
(3)           if (c == d)
              {
                // T
          ...
```

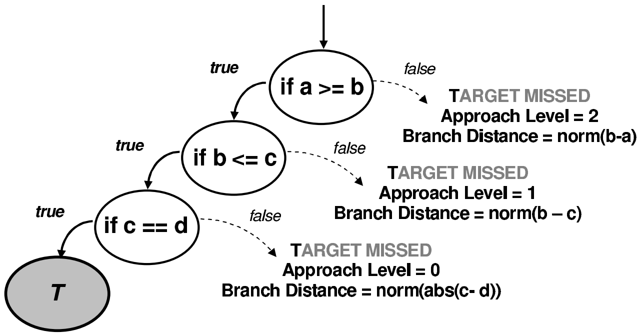


Figure 2: Calculating the fitness function for evolutionary testing.

critical branching node was to being taken. A raw distance is computed using the values of the variables or constants involved in the branching condition. For example, if the false branch were taken from Node 3 in Figure 2, the raw distance for the alternative *true* branch is computed using the formula `abs(c - d)`. The closer the values of `c` and `d`, the smaller the distance value, and the closer the branch is to being executed as true. A full list of formulas for different types of branch predicate can be found in Tracey *et al.*[27]. The raw distance value is then normalized (Equation 1 below) and added to the approach level to make up the complete fitness value (Equation 2 below).

$$branch\_distance = norm(raw\_dist) \quad (1)$$

$$fitness = approach\_level + branch\_distance \quad (2)$$

The fitness function derived for a target is only concerned with the branching nodes which absolutely must be executed in a certain way in order for the target to be reached in the control flow graph. It is not concerned with the specifics of the complete path taken to the target – this discovery is usually left as part of the search.

3. THE PATH PROBLEM

The test data generation problem involves finding test data to execute some path p that ends with the target T. As described in the last section, the path p does not have to be selected by the tester for search-based testing, it is sought as part of the search process.

Let P be the set of feasible paths which execute a target T. The path problem occurs when the total number of input vectors executing each path in P account for only a small proportion of the overall input domain. In such instances, paths in P are unlikely to be executed by random chance. Furthermore, the search may not be equipped with adequate guidance to find test data to execute any $p \in P$. This is because the fitness function is concerned only with a subset of branching nodes that exist within each p – the set of critical branching nodes. Whilst critical branching nodes determine whether T will be reached in the control flow structure of the program, other branching nodes may also exist within p that control assignments to variables that are also responsible for determining whether T will be executed or not. However, it is not possible to know which of these branching nodes should be included for the purposes of computing the fitness function, since P is not known *a priori* (finding P is in general undecidable), and neither is the exact sequence of nodes in each p.

The path problem exists in the example of Figure 1a where there are four potential paths ending in T. However, three of them are infeasible, and the feasible path $\langle s, 1, 3, 5, T\rangle$ is only executed by a single input vector. The fitness function only utilizes fitness information at Node 5, since this is the only critical branching node that T is dependent upon. The search is unaware that the false branches must be taken from Nodes 1 and 3, in order to avoid the fatal assignments at Node 2 and 4 respectively. The search landscape is flat (Figure 1c), giving no guidance to the search. This is because for each infeasible path it is impossible for Node 5 to be reached without the variable `a` having the value 2, at which point the short-circuit evaluation determines that the entire condition is false.

4. THE SPECIES PER PATH SOLUTION

The foundation of the SpP approach is the division of search effort across several parallel sub-populations or individual 'species', each assigned to the task of finding test data for each individual path. To this end:

- Each species uses a fitness function tailored for that path, and so the search can receive full guidance with respect to each branching node within it.
- No path is favoured above another path simply because it is executed by more of the input domain.

The SpP approach is likely to increase the chances of successful test data generation for search targets suffering from the path problem. Resources are guaranteed to feasible paths, and species allocated to these paths are equipped with a higher level of search guidance than is available to the conventional approach. Species allocated to infeasible paths simply fail to produce a result. These claims are confirmed by the results of the verification study, described in Section 5.

To facilitate the division of labour amongst species, program transformation and program slicing are used. Neither is strictly necessary because the paths can be identified without transformation, and slicing only serves to improve efficiency.

The transformation allows the original fitness function, as described in Section 2.2, to be applied. The transformation separates out the loop-free paths from node i to node j by bushing the program's Abstract Syntax Tree (AST) [12].

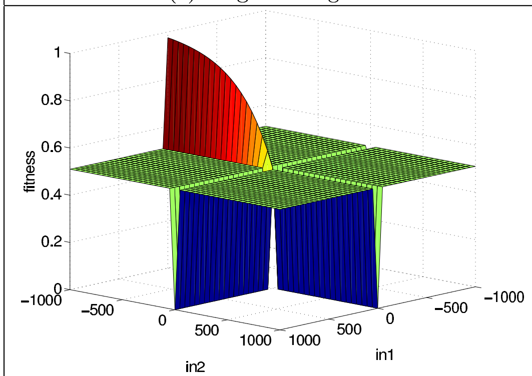
In essence, this transformation moves code from after the conditional into the consequent and alternative branches of the conditional.

```
CFG
Node
 (s)     void example2(double in1, double in2)
         {
 (1)         double a = 0;
 (2)         double b = 0;

 (3)         if (in1 == 0) {
 (4)             a = 20;
 (5)             b = in2 * 2;
             }

 (6)         if (in2 == 0) {
 (7)             a = in1 * 3;
 (8)             b = 100;
             }

 (9)         if (a >= 20 && b >= 20) {
(10)             // T
             }
         }
```

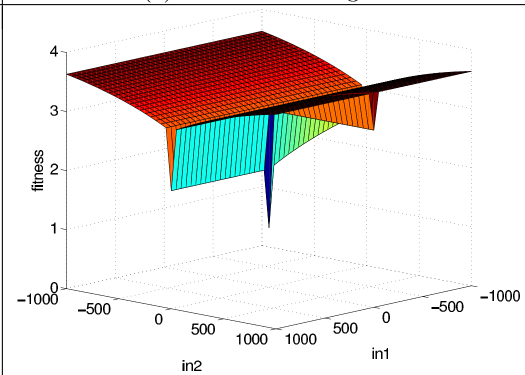
(a) Original Program

```
void example2_transformed(double in1, double in2)
{
    double a = 0;
    double b = 0;
    if (in1 == 0) {
        a = 20;
        b = in2 * 2;
        if (in2 == 0) {
            a = in1 * 3;
            b = 100;
            if (a >= 20 && b >= 20) {
                // T1
            }
        } else if (a >= 20 && b >= 20) {
            // T2
        }
    } else if (in2 == 0) {
        a = in1 * 3;
        b = 100;
        if (a >= 20 && b >= 20) {
            // T3
        }
    } else if (a >= 20 && b >= 20) {
        // T4
    }
}
```

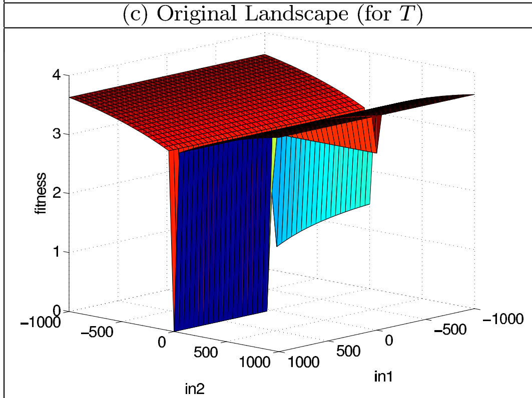
(b) Transformed Program



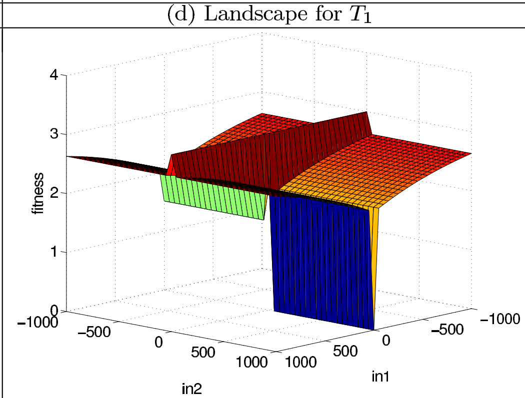
(c) Original Landscape (for T)



(d) Landscape for T_1

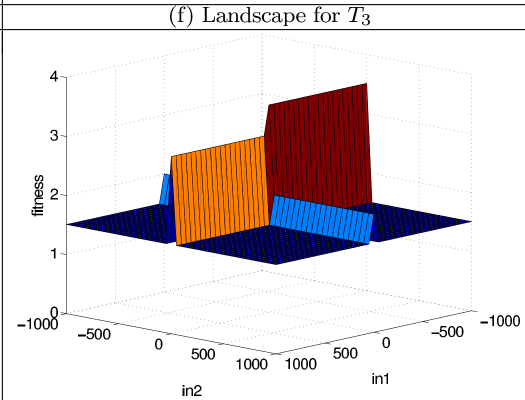


(e) Landscape for T_2



(f) Landscape for T_3

Figure 3: The path problem. T_1 and T_4 are infeasible – no part of their surface touches zero on the vertical axis. T_2 and T_3 are feasible, and although the surfaces contain ridges and local optima, the basin of attraction present in both their landscapes is considerably greater than the landscape of T, leading to higher search efficiency and success rate.



(g) Landscape for T_4

Prior to transformation, slicing is applied with respect to the target. This removes statements from the program that have no effect on the target, potentially reducing the number of paths, and thus the number of copies of the target in the transformed version of the program.

An example transformation can be seen from the program of Figure 1a to that of Figure 1b. In the transformed version there are four copies of the target corresponding to the four paths to T. Three of these copies, T_1, T_2 and T_3, turn out to be infeasible. T_4 is feasible. Figures 1e and 1f compare fitness computation for T and T_4 respectively. Whereas the initial two conditions at Nodes 1 and 3 are ignored for T in the original program, they become critical branching nodes for T_4 in the transformed version, and are therefore used in the fitness computation. The fact that both of these conditions have to be evaluated as false is crucial information which guarantees feasibility of the target.

The effect on the fitness landscape is seen by comparing that for T in the original program and that of T_4 in the transformed version. The landscape for T is flat (Figure 1c), providing no guidance to the search whatsoever. However landscape for T_4 (Figure 1d) makes things significantly easier for the search, providing clear direction as to where the required test data exist in the input domain.

A second example is presented in Figure 3. Figure 3a shows the original program. The target T is only executed by a path which takes the *true* branch from branching nodes 3 or 6 – branching nodes which are key but not critical for T in the original program. These nodes are only executed as true by a small number of input vectors. The resultant search landscape largely consists of plateaux (Figure 3c).

Figure 3b shows the transformed version of the program, which has been bushed to expose four paths, and consequently includes four copies of target T. The four landscapes corresponding to T_1 to T_4 are depicted in Figures 3d through 3g respectively. All of these landscapes have certain features in common with the original landscape. This is because the fitness function for T combines the results of the computation along each path over the areas of the input domain for which those individual paths are executed. The SpP approach separates this information out into the landscape for each species, adding further information to guide the search to the execution of each individual path. Targets T_2 and T_3 are on feasible paths. This is evident from their search landscapes, since parts of their respective surfaces touch zero on the vertical axes. Landscapes T_1 and T_4 are on infeasible paths – no part of the landscape touches zero on the vertical axis – meaning there is no value in the input space which executes the target branch. A search in either of the two infeasible landscapes is doomed to fail. However, searching the two feasible landscapes not only allows for successful discovery of test data, but a higher likelihood of success than if the original search landscape were to be used. Again, this is due to the extra fitness information which comes from the inclusion of extra branching nodes in the fitness function. Although the feasible landscapes produced by the SpP approach do contain areas of ridges and local optima, the basin of attraction to the global optimum is considerably larger.

5. EMPIRICAL VERIFICATION

Experiments were performed using the standard evolutionary approach and the SpP method with three programs: the examples from Figures 1 and 3 and a third example, called `space`, taken from production code. Although the first two examples are small (in terms of code size), the complexity of any search-based algorithm is a function of the search space, not the program size. Therefore, the important determinant of problem difficulty for search–based testing is the size of the input domain. As is described in the following sections, input domain sizes for all three test objects ranged from the order of 10^8 to 10^{14} possible input vectors.

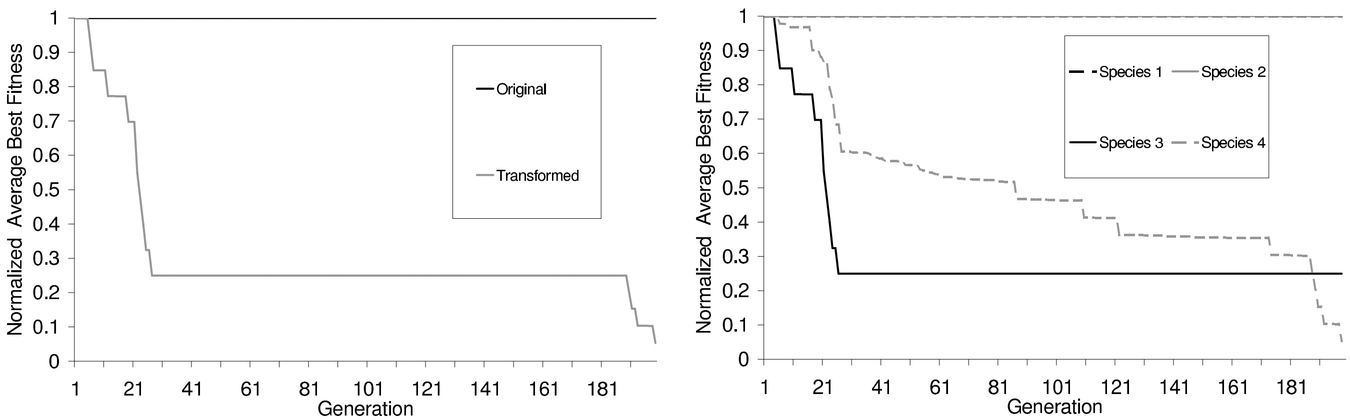
The evolutionary searches were performed with the publicly available *Genetic and Evolutionary Algorithm Toolbox (GEATbx)* [24]. A detailed explanation of each of the search parameters is beyond the scope of this paper, however they are fully documented at the GEATbx web-site [24], and are recorded here to allow replication of the experiments. Real-valued encodings were used to represent the input vectors, linear ranking was utilized with a selection pressure of 1.7. The generation gap was set at 0.8. Individuals were recombined using discrete recombination, and mutated using Mühlenbein mutation. Finally, each evolutionary search was terminated after 200 generations if test data was not found.

5.1 Example 1

The standard evolutionary approach described in Section 2.2 was applied to the original version of the program shown in Figure 1a using 240 individuals per generation. The SpP approach was applied to the transformed version shown in Figure 1b. With four targets, four species were allocated - one per target each having 60 individuals. Therefore both approaches used the same number of individuals overall per generation. The ranges of the input variables `a` and `b` were set to -1000.0 to 1000.0, with a precision of 0.1, giving an input domain (search space) size of approximately 4×10^8.

Figure 4a plots the search progress for the experiment. For the two approaches, the graph shows the best fitness obtained so far in the search for a particular generation, averaged over the ten repetitions of the experiment. This value is then normalized to allow for a direct comparison between the two methods. Due to the undirectional landscape of the target T, the original method fails to make any progress at all; thus the horizontal line at the top of the figure. In contrast, the SpP approach is more successful. The progress of each species is plotted in Figure 4b. Species 3 makes the best start, but since T_3 is infeasible, it hits a 'dead end' before the 30th generation. Species 1 and 2 also correspond to infeasible paths, and fail to make any progress (the progress line for species 1 and species 2 follow the same course). However, Species 4, corresponding to T_4, makes steady progress to the required test data. T_4 corresponds to a feasible path. The species is equipped with more fitness information than is available to the original approach, and was able to find test data in 9 out of the 10 repetitions. In one run, however, the species failed to reach the target within the 200 generations limit, which is why the average best fitness line for Species 4 gets close but does not actually touch zero.

The success rates of each method are summarized in Table 1, along with the number of test data evaluations for successful and unsuccessful searches. The original method fails to find test data in every run, but the SpP succeeds 90% of the time.



(a) Original vs. Transformed

(b) Species

Figure 4: Search results for Example 1. (a) Search progress using original and transformed versions of the program. Progress is plotted by aggregating each species for the transformed version. (b) Progress of each individual species using the transformed version of the program.

Table 1: Results for Example 1 - success rates and fitness evaluations

		Average No. of Fitness Evaluations	
	Success Rate	Successful Searches	Unsuccessful Searches
Original	0%	*n/a*	38,448
Transformed	90%	31,039	38,448

Table 2: Results for Example 2 - success rates and fitness evaluations

		Average No. of Fitness Evaluations	
	Success Rate	Successful Searches	Unsuccessful Searches
Original	50%	11,568	38,448
Transformed	100%	8,584	*n/a*

5.2 Example 2

The setup for Example 2 (Figure 3) was largely the same as that for Example 1. For the original version of the program, 240 individuals were used per generation. Again, with four paths factored out in the transformed version of the program, the SpP approach used four different species comprising 60 individuals each, and thus a total of 240 individuals per generation. Input ranges were -1000.0 to 1000.0 with a precision of 0.1 for both `in1` and `in2`, giving an input domain size of approximately 4×10^8.

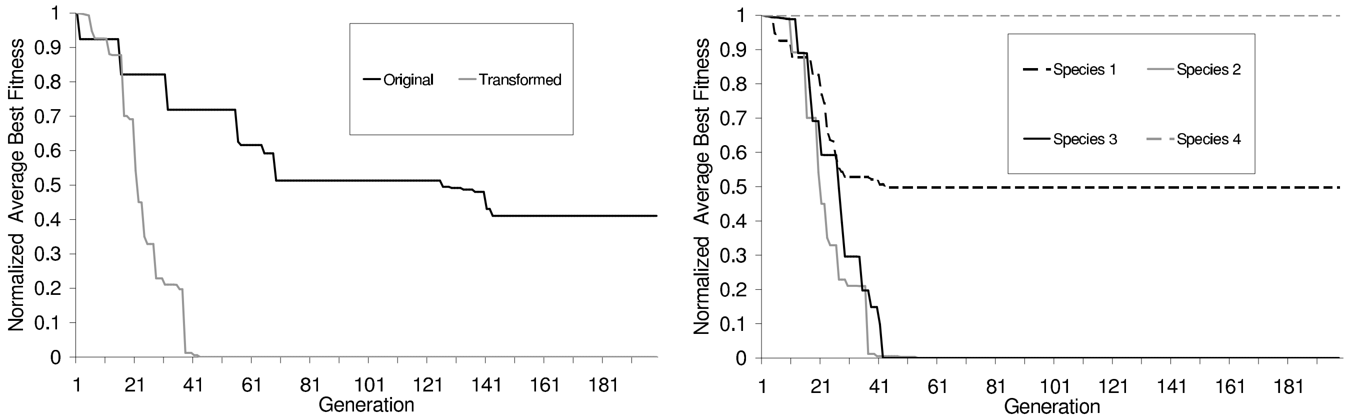
Experiments were repeated 10 times. The success rate with both original and SpP approaches is shown in Table 2. Searches using the conventional approach succeeded only half of the time. Searches always succeed using the SpP approach. The table shows that when comparing successful searches, SpP was more efficient. Figure 5a shows a progress plot comparing both approaches. Progress using the original version of the program is stilted, because of the plateaux in the landscape for T. Test data is more easily found using the SpP method. Species 2 and 3 correspond to feasible paths, and these are the species which are successful in making progress to the discovery of the required test data (Figure 5b).

5.3 Space Example

The final test object `space` was taken from EU space agency production code. The test object contains a target that experiences the path problem. There are eight paths to this target, resulting in eight copies of the target in the transformed version of the program and eight corresponding species.

For this program, different input ranges and species sizes were used. Search success using the standard evolutonary approach was limited to relatively smaller input domain sizes, as seen in Table 3. Even at this level, test data was not generated with 100% reliability. The SpP approach, however, was almost 100% reliable. Only on two repetitions of one experiment did the search fail. This was when the species size was small (30 individuals per species) in the operation of a relatively large domain (1.2×10^{14}). When comparing successful searches, the SpP method is more efficient, consistently finding test data in fewer fitness evaluations.

Figure 6 compares search progress with 480 individuals per generation for both approaches, and an input domain size of 1.4×10^{13}. Figure 6a compares search progress between the original method and SpP. The original approach fails to make any progress. The progress of each individual species are compared in Figure 6b. Of the eight species, only one corresponds to a feasible path – Species 7 – which finds test data by the 76^{th} generation at the latest. Species 4 and 6 get very close to a zero fitness value, but since their paths are infeasible, test data is not found. This can be seen in a close-up of species progress, depicted in Figure 6c.



(a) Original vs. transformed

(b) Species

Figure 5: Results for Example 2. (a) Search progress using original and transformed versions of the program. Progress is plotted by aggregating each species for the transformed version. (b) Progress of each individual species using the transformed version of the program.

Table 3: Detailed results for space

Input domain size		Success Rate	Average Fitness Evaluations: Successful Searches	Average Fitness Evaluations: Unsuccessful Searches
240 individuals per generation (30 per species)				
1.4×10^8	Original	10%	32,885	38,448
	Transformed	100%	16,765	*n/a*
1.4×10^{10}	Original	0%	*n/a*	38,448
	Transformed	100%	21,575	*n/a*
1.4×10^{13}	Original	0%	*n/a*	38,448
	Transformed	100%	20,756	*n/a*
1.2×10^{14}	Original	0%	*n/a*	38,448
	Transformed	80%	21,694	38,448
480 individuals per generation (60 per species)				
1.4×10^8	Original	60%	15,830	76,896
	Transformed	100%	18,612	*n/a*
1.4×10^{10}	Original	0%	*n/a*	76,896
	Transformed	100%	19,048	*n/a*
1.4×10^{13}	Original	0%	*n/a*	76,896
	Transformed	100%	32,096	*n/a*
1.2×10^{14}	Original	0%	*n/a*	76,896
	Transformed	100%	27,252	*n/a*
720 individuals per generation (90 per species)				
1.4×10^8	Original	80%	37,821	115,344
	Transformed	100%	16,832	*n/a*
1.4×10^{10}	Original	70%	49,763	115,344
	Transformed	100%	26,468	*n/a*
1.4×10^{13}	Original	0%	*n/a*	115,344
	Transformed	100%	27,043	*n/a*
1.2×10^{14}	Original	0%	*n/a*	115,344
	Transformed	100%	37,544	*n/a*

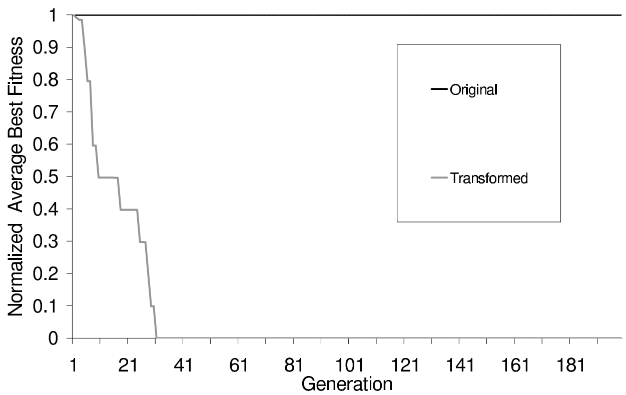
6. EMPIRICAL VALIDATION

This section validates the approach by showing that the transformation is widely applicable and that bushing will not lead to a prohibitively expensive explosion in the number of copies of the target that must be considered. The higher the number of copies, the greater the number of species that must be employed, and the less practical the approach becomes. A low value of predicates per block serves to validate that the technique does not require excessive (exponential in the number of predicates) expansion when replicating T.

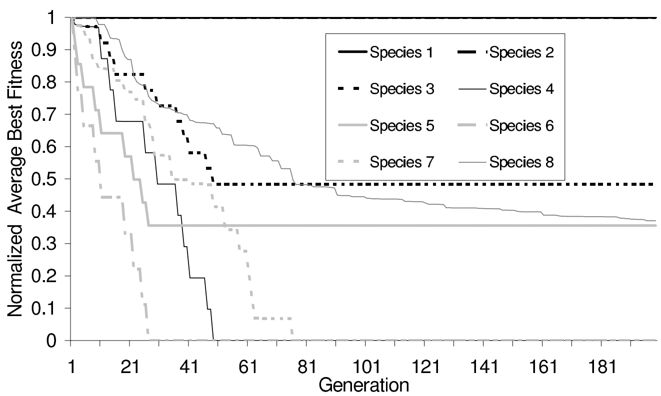
A collection of 45 programs including both open and propriety source was used in the validation study. The programs cover a wide variety of programming styles including interactive GUI-based code, batch array processing code, system utilities, games, etc. The results are presented in Table 4. The first two columns of the table include the number of predicates and the number of predicate–containing blocks. A *block* is a section of code in between loops or inside a single loop. The remaining columns provide statistics on chain lengths, which include the number of predicates per block, the longest chain length, and the count and percentage of *short chains*. A short chain is one of four or fewer predicates. When expanded by bushing, such a chain will give rise to no more than 16 paths, making the SpP approach viable. The final column shows the percentage of short chains: those having four or fewer predicates.

From the data in Table 3 it can be seen that a higher number of individuals increases the probability of success. Since the total number of individuals is divided among the species, a high number of paths could decrease the chance of a successful search. The data in Table 4 shows that this is not a problem. It is clear that most predicate chains (on average 98% as seen in the last row of Table 4) are short enough for the bushing technique to be effective. Furthermore, the average chain length of 1.5 predicates indicates that on average no more than four paths and thus four species need be considered.

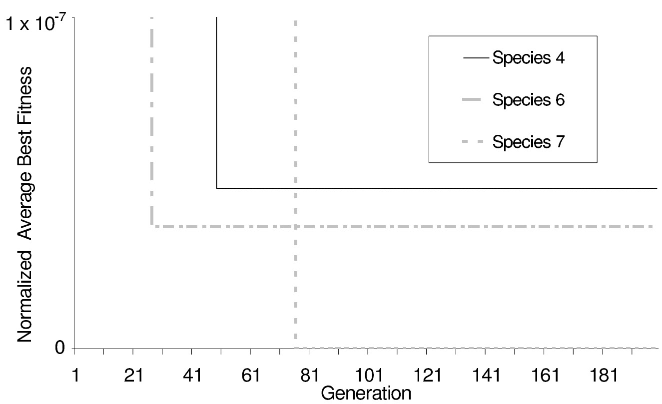
Figure 7 shows the lengths of all chains having one or more predicate for the program espresso, which is typical of the programs studied. In this example, only 21 of the 878



(a)



(b)



(c)

Figure 6: Results for space. (a) Search progress using original and transformed versions of the program. Progress is plotted by aggregating each species for the transformed version. (b) Progress of each individual species using the transformed version of the program. (c) Close-up for Species 4, 6 and 7.

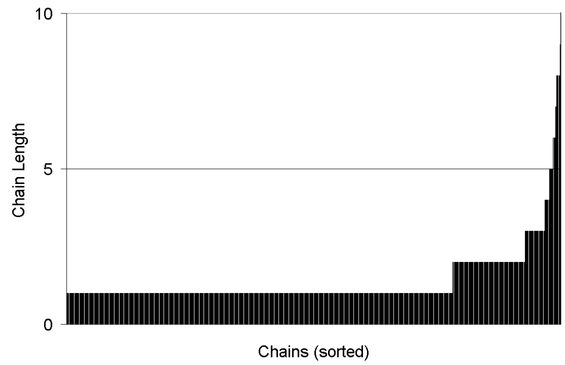


Figure 7: Sorted chain lengths for espresso.

blocks contain chains of five or more predicates. The graph is clipped at length 10 (the y-axis), which only affects the last chain of fourteen predicates.

7. RELATED WORK

Many techniques have been proposed to automate the generation of software test data. Methods based on static analysis of the program's source code, such as symbolic execution [7, 15] and constraint solving [9, 22], have been limited by the dynamic nature of software – for example the presence of unbounded loops and dynamic memory referencing, such as the use of pointers. For these reasons, the application of search techniques, such as evolutionary algorithms, to search a program's input domain for test data has been a topic of interest for many researchers in recent years.

This paper addresses the problem where search-based approaches do not have sufficient fitness information to find feasible paths to execute a target. This problem occurs when such paths occupy a small proportion of the program's input domain, and thus it is an example of the high domain to range ratio problem described by Voas and Miller [28] as a source of poor testability. High domain to range ratio is not necessarily a problem for search-based methods if adequate guidance is provided to the search. This is the principle behind the SpP approach.

For techniques like symbolic execution and constraint solving, as well as some early search-based approaches [21, 16, 32], the tester must select a path. The goal-oriented approach [17] was the first search-based technique to remove this requirement, but was also the first to suffer from the path problem described in this paper. The chaining approach [10] alleviates the path problem by attempting to find program nodes on which the target is data-dependent and which must be executed in the path to the target. These are explored in a series of chains. However no program transformation takes place, paths are not isolated, and thus the fitness function could potentially encounter 'noise' from other paths. Evolutionary-chaining hybrids [20] have the disadvantage that an entirely new search must be attempted for each chain.

The first author to propose the idea of parallel subpopulations for exploring different paths was Bottaci [4]. In a later paper [5], Bottaci implemented a multi-population method where each sub-population performs a test data search using a fitness function that is based on control dependency information of the original target and a set of dynamically identified sub-goals. Sub-goals are identified on the basis of branches that were not executed *en route* to the target. It

Table 4: Lengths of inter-loop non-nested predicate chains. The final column shows the percentage of *short* chains (those with less than 5 predicates).

Program	Predi-cate Count	No. of Blocks	Predi-cates per Block	Max. Length	Short Chains (%)
a2ps	3343	2118	1.6	37	96%
acct	544	352	1.5	11	97%
barcode	430	249	1.7	15	95%
bc	465	337	1.4	26	98%
byacc	820	542	1.5	10	97%
cadp	675	472	1.4	5	99%
compress	140	85	1.6	4	100%
cook-cook	3366	2274	1.5	40	98%
copia	91	60	1.5	3	100%
csurf-packages	1855	1613	1.2	8	99%
ctags	1250	950	1.3	8	99%
cvs	9688	5474	1.8	23	94%
diffutils	1554	1060	1.5	19	98%
ed	1191	870	1.4	27	99%
empire	8041	5804	1.4	18	98%
epwic	631	424	1.5	10	98%
espresso	1223	878	1.4	14	97%
findutils	1283	929	1.4	19	99%
flex2-4-7	816	487	1.7	30	97%
flex2-5-4	1063	645	1.6	32	97%
ftpd	2519	1547	1.6	146	99%
gcc.cpp	936	592	1.6	14	97%
gnubg-0.0	77	46	1.7	3	100%
gnuchess	1358	872	1.6	12	97%
gnugo	3018	2052	1.5	16	97%
go	79	48	1.6	3	100%
ijpeg	1260	858	1.5	11	98%
indent-1.10.0	925	466	2.0	55	93%
li	436	338	1.3	4	100%
named	9707	6787	1.4	62	97%
ntpd	2755	1843	1.5	24	96%
oracolo2	736	591	1.2	6	99%
prepro	717	585	1.2	6	99%
replace	55	45	1.2	6	97%
sendmail-8.7.5	4776	2654	1.8	51	94%
snns-batchman	6164	4832	1.3	24	99%
space	816	658	1.2	6	99%
spice	7418	4951	1.5	25	97%
termutils	356	233	1.5	8	98%
tile-forth-2.1	266	202	1.3	5	99%
time-1.7	131	80	1.6	5	98%
userv-0.95.0	987	681	1.4	13	96%
wdiff.0.5	255	169	1.5	13	98%
which	171	111	1.5	4	100%
wpst	934	843	1.1	4	100%
Sum	85,321	57,707			
Average	1,896	1,282	1.5	20	98%

is remarked that there may be a population for each sub-set of the non-target branches in the program, and that it is unclear how many of these cases arise in practice. The present paper shows how paths to the target can be assigned to species statically on the basis of slicing and transformation. It is the first paper to evaluate any form of species approach on a real world program and verify the results against the standard evolutionary method. It is also the first to present results that validate the practicality of the approach.

Other problems in search-based test data generation can be thought of as specific cases of the path problem. For example, the flag problem [2, 4, 12] is caused by a path being taken which does not set the flag to some required value. Other problems with enumerations and deceptive landscapes [20] can also be cast in terms of the path problem, and thus the SpP approach is likely to be successful for programs with these features.

The SpP approach uses program transformation to enable the original branch fitness function to be applied, also known as testability transformation [12]. Testability transformations have also been employed to tackle some of the problems mentioned above, including flags [1], unstructured programs [13] and nesting [19].

8. CONCLUSIONS AND FUTURE WORK

This paper introduces the Species per Path approach for search-based test data generation. The SpP approach is shown to generate test data more reliably and efficiently for test objects experiencing the the *path problem*, which is identified in this paper as a problem for search-based automated test data generators. Each species concentrates on finding test data to execute a specific path to the target. The factoring out of paths by a testability transformation provides each species with more fitness information than would otherwise be available to the standard evolutionary method.

These claims are supported by a verification empirical study that compares the original method with the SpP approach using three examples, including one taken from production code. A second empirical study concerning a varied collection of real world programs provides evidence to support the claim that the approach is also widely applicable.

Further work will extend the approach to use conditioned slicing [6] to remove paths which can be statically determined to be infeasible, thereby freeing up a species which would otherwise be fruitlessly occupied. Additional experimentation will be conducted to explore the possibility of competition between species as a mechanism for further improving the efficiency of the approach and exploiting the species metaphor.

9. ACKNOWLEDGMENTS

Gregg Rothermel kindly provided the space example.

Phil McMinn is funded by DaimlerChrysler Research and Technology and EPSRC grant GR/S62321.

Mark Harman is supported, in part, by EPSRC grants EP/D050863, GR/R98938, GR/S93684 and GR/T22872.

David Binkley is supported by National Science Foundation grant CCR-0305330.

10. REFERENCES

[1] A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In *Proc. International Symposium on Software Testing and Analysis (ISSTA 2004)*, pp. 43–52, Boston, Massachusetts, USA, 2004. ACM.

[2] A. Baresel and H. Sthamer. Evolutionary testing of flag conditions. In *Proc. Genetic and Evolutionary Computation Conference (GECCO 2003), Lecture Notes in Computer Science vol. 2724*, pp. 2442 – 2454, Chicago, USA, 2003. Springer-Verlag.

[3] D. Binkley and K. Gallagher. Program slicing. In M. Zelkowitz, editor, *Advances in Computing, Volume 43*, pp. 1–50. Academic Press, 1996.

[4] L. Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithm. In *Proc. Genetic and Evolutionary Computation Conference (GECCO 2002)*, pp. 1337 – 1342, New York, USA, 2002. Morgan Kaufmann.

[5] L. Bottaci. Use of branch cost functions to diversify the search for test data. In *Proc. Third UK Testing Workshop (UKTest 2005)*, pp. 151–163. University of Sheffield Computer Science Technical Report CS-05-07, 2005.

[6] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pp. 595–607. Elsevier Science B. V., 1998.

[7] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, Sept. 1976.

[8] C. Darwin. *On the Origin of Species by Means of Natural Selection*. Oxford University Press, 1859.

[9] R. A. DeMillo and A. J. Offutt. Experimental results from an automatic test generator. *ACM Transactions of Software Engineering and Methodology*, 2(2):109–127, Mar. 1993.

[10] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, 1996.

[11] J. Ferrante, K. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[12] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, Jan. 2004.

[13] R. Hierons, M. Harman, and C. Fox. Branch-coverage testability transformation for unstructured programs. *The Computer Journal*, 48(4):421–436, 2005.

[14] B. Jones, H. Sthamer, and D. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.

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

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

[17] B. Korel. Dynamic method for software test data generation. *Software Testing, Verification and Reliability*, 2(4):203–213, 1992.

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

[19] P. McMinn, D. Binkley, and M. Harman. Testability transformation for efficient automated test data search in the presence of nesting. In *Proc. UK Software Testing Workshop (UKTest 2005)*, pp. 165–182. University of Sheffield Computer Science Technical Report CS-05-07, 2005.

[20] P. McMinn and M. Holcombe. Hybridizing evolutionary testing with the chaining approach. In *Proc. Genetic and Evolutionary Computation Conference (GECCO 2004), Lecture Notes in Computer Science vol. 3103*, pp. 1363–1374, Seattle, USA, 2004. Springer-Verlag.

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

[22] A. J. Offutt. An integrated system for automatically generating test data. In *Proc. First International Conference on Systems Integration*, pp. 694–701, Morristown, NJ, Apr. 1990. IEEE Computer Society Press.

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

[24] H. Pohlheim. GEATbx - Genetic and Evolutionary Algorithm Toolbox, http://www.geatbx.com.

[25] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[26] N. Tracey, J. Clark, and K. Mander. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *Proc. International Workshop on Dependable Computing and Its Applications*, pp. 169–180. Computer Science Dept., University of Witwatersrand, South Africa, 1998.

[27] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Proc. International Conference on Automated Software Engineering*, pp. 285–288, Hawaii, USA, 1998. IEEE Computer Society Press.

[28] J. M. Voas and K. W. Miller. Software testability: The new verification. *IEEE Software*, 12(3):17–28, 1995.

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

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

[31] D. Whitley. An overview of evolutionary algorithms: Practical issues and common pitfalls. *Information and Software Technology*, 43(14):817–831, 2001.

[32] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. Application of genetic algorithms to software testing. In *Proc. 5th International Conference on Software Engineering and its Applications*, pp. 625–636, Toulouse, France, 1992.