SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2004; **14**:105–156 (DOI: 10.1002/stvr.294)

Search-based software test data generation: a survey

Phil McMinn*,†

Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello Street, Sheffield S1 4DP, U.K.

SUMMARY

The use of metaheuristic search techniques for the automatic generation of test data has been a burgeoning interest for many researchers in recent years. Previous attempts to automate the test generation process have been limited, having been constrained by the size and complexity of software, and the basic fact that, in general, test data generation is an undecidable problem. Metaheuristic search techniques offer much promise in regard to these problems. Metaheuristic search techniques are high-level frameworks, which utilize heuristics to seek solutions for combinatorial problems at a reasonable computational cost. To date, metaheuristic search techniques have been applied to automate test data generation for structural and functional testing; the testing of grey-box properties, for example safety constraints; and also non-functional properties, such as worst-case execution time. This paper surveys some of the work undertaken in this field, discussing possible new future directions of research for each of its different individual areas. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: search-based software engineering; automated software test data generation; evolutionary testing; metaheuristic search; evolutionary algorithms; simulated annealing

1. INTRODUCTION

The use of metaheuristic search techniques for the automatic generation of test data has been a burgeoning interest for many researchers in recent years. In industry, test data selection is generally a manual process—the responsibility for which usually falls on the tester. However, this practice is extremely costly, difficult and laborious. Automation in this area has been limited. Exhaustive enumeration of a program's input is infeasible for any reasonably sized program, yet random methods are unreliable and unlikely to exercise 'deeper' features of software that are not exercised by mere chance. Previous efforts have been limited by the size and complexity of the software involved, and the basic fact that, in general, test data generation is an undecidable problem.

*Correspondence to: Phil McMinn, Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello Street, Sheffield S1 4DP, U.K.
†E-mail: p.mcminn@dcs.shef.ac.uk

Received 12 June 2003
Accepted 29 January 2004

The application of metaheuristic search techniques to test data generation is a possibility which offers much promise for these problems. Metaheuristic search techniques are high-level frameworks which utilize heuristics in order to find solutions to combinatorial problems at a reasonable computational cost. Such a problem may have been classified as NP-complete or NP-hard, or be a problem for which a polynomial time algorithm is known to exist but is not practical. They are not standalone algorithms in themselves, but rather strategies ready for adaption to specific problems. For test data generation, this involves the transformation of test criteria to *objective functions*. Objective functions compare and contrast solutions of the search with respect to the overall search goal. Using this information, the search is directed into potentially promising areas of the search space.

Search-based software test data generation is just one example of search-based software engineering [1,2]. To date, metaheuristic search techniques have been applied to automate test data generation in the following areas:

- the coverage of specific program structures, as part of a structural, or white-box testing strategy;
- the exercising of some specific program feature, as described by a specification;
- attempting to disprove automatically certain grey-box properties regarding the operation of a piece of software, for example trying to stimulate error conditions, or falsify assertions relating to the software's safety;
- to verify non-functional properties, for example the worst-case execution time of a segment of code.

This paper surveys work undertaken in these areas and the results achieved. Section 2 begins by reviewing some of the search techniques used. Section 3 discusses their application to structural testing, which to date has received the greatest share of attention from search-based testing researchers. Section 4 presents work in the area of functional testing, followed by grey-box testing (Section 5) and finally non-functional testing (Section 6). At the end of each section, the paper outlines possible directions for future research appropriate to that area.

2. METAHEURISTIC SEARCH TECHNIQUES

In order to adapt a metaheuristic search technique to a specific problem, a number of different decisions have to be made—for example, the way in which solutions should be encoded so that they can be manipulated by the search. A good choice of encoding will ensure that similar solutions in unencoded space are also 'neighbours' in representational space. In this way, the search will be allowed to move easily from one solution to another that shares a similar set of properties. These movements are dependent on the evaluation of candidate solutions, performed using a problem-specific *objective function*. With feedback from the objective function, the search seeks 'better' solutions based on knowledge and experience of previous candidates. A good objective function is therefore critical to the success of the search. Solutions that are 'better' in some respect should be rewarded with better objective values, whereas poorer solutions should be punished with poorer objective values. Whether a 'better' objective value is, in practice, a higher value or lower value is dependent on whether the search is seeking to minimize or maximize the objective function. An objective function which is being maximized reflects the relative 'goodness' of candidate solutions, whereas an objective function to be

Select a starting solution $s \in S$
Repeat
    Select $s' \in N(s)$ such that $obj(s') > obj(s)$ according to ascent strategy
    $s \leftarrow s'$
Until $obj(s) \geq obj(s'), \forall s' \in N(s)$

Figure 1. High-level description of a hill climbing algorithm for a problem with solution space S; neighbourhood structure N; and obj, the objective function to be maximized.

minimized (more usually referred to in this context as a *cost function*) reflects the relative undesirability of solutions.

The next section outlines some metaheuristic techniques that have been used in software test data generation, namely hill climbing, simulated annealing and evolutionary algorithms. Further treatment of these search techniques can be found in reference [3]. The last decade has seen the emergence of many new techniques, which have not been exploited by the test data generation techniques presented here. Reference [4] gives treatment to some of these.

2.1. Hill climbing

'Hill climbing' is a well known local search algorithm. Hill climbing works to improve one solution, with an initial solution randomly chosen from the search space as a starting point. The neighbourhood of this solution is investigated. If a better solution is found, then this replaces the current solution. The neighbourhood of the new solution is then investigated. If a better solution is found, the current solution is replaced again, and so on, until no improved neighbours can be found for the current solution.

This progressional improvement is likened to the climbing of hills in the 'landscape' of a maximizing objective function. In this landscape, peaks characterize solutions with locally optimal objective values, and troughs signify solutions with the locally poorest objective values. In a 'steepest ascent' climbing strategy, all neighbours are evaluated, with the neighbour offering the greatest improvement chosen to replace the current solution. In a 'random ascent' strategy (sometimes referred to as 'first ascent'), neighbours are examined at random and the first neighbour to offer an improvement is chosen. A high level description of the algorithm can be seen in Figure 1.

Hill climbing is simple and gives fast results. However, it is easy for the search to yield sub-optimal results when the hill climbed leads to a solution that is locally optimal, but not globally optimal. In such cases, the search becomes trapped at the peak of a hill, unable to explore other areas of the search space. The search will also become stuck along plateaux in the landscape. In such circumstances, no neighbouring solution is deemed to offer an improvement over the current solution, since they all have the same objective value. Therefore, in non-trivial landscapes, results obtained with hill climbing are highly dependent on the starting solution. A common extension to this algorithm is to incorporate a series of 'restarts' involving different initial solutions to sample more of the search space and minimize this problem as much as possible.

```
Select a starting solution s ∈ S
Select an initial temperature t > 0
Repeat
    it ← 0
    Repeat
        Select s′ ∈ N(s) at random
        Δe ← obj(s′) − obj(s)
        If Δe < 0
            s ← s′
        Else
            Generate random number r, 0 ≤ r < 1
            If r < e^(−δ/t) Then s ← s′
        End If
        it ← it + 1
    Until it = num_solns
    Decrease t according to cooling schedule
Until Stopping Condition Reached
```

Figure 2. High-level description of a simulated annealing algorithm for a problem with solution space S; neighbourhood structure N; *num_solns*, the number of solutions to consider at each temperature level t; and *obj*, the objective function to be minimized.

2.2. Simulated annealing

It is desirable to have a search framework that is less dependent on the starting solution. Simulated annealing is similar in principle to hill climbing. However, by probabilistically accepting poorer solutions, simulated annealing allows for less restricted movement around the search space. The probability of acceptance p of an inferior solution changes as the search progresses, and is calculated as

$$p = \mathrm{e}^{-\delta/t}$$

where δ is the difference in objective value between the current solution and the neighbouring inferior solution being considered, and t is a control parameter known as the *temperature*. The temperature is cooled according to a *cooling schedule*. Initially the temperature is high in order to allow free movement around the search space and so that dependency on the starting solution is lost. As the search progresses, the temperature decreases. However, if cooling is too rapid, not enough of the search space will be explored, and the chances of the search becoming stuck in local optima are increased. The basic algorithm, for minimizing an objective function can be seen in Figure 2.

The name 'simulated annealing' originates from the analogy of the technique with the chemical process of annealing—the cooling of a material in a heat bath. If a solid material is heated past its melting point, and then cooled back into a solid state, the structural properties of the cooled solid depend on the rate of cooling. An algorithm proposed by Metropolis *et al.* [5] simulates the change

in energy of the system when subjected to a cooling process, until it converges into a steady state. This algorithm was later proposed as the basis of the search mechanism by Kirkpatrick *et al.* [6].

2.3. Evolutionary algorithms

Evolutionary algorithms use simulated evolution as a search strategy to evolve candidate solutions, utilizing operators inspired by genetics and natural selection.

Genetic algorithms are probably the most well known form of evolutionary algorithm, having been conceived by John Holland in the United States during the late 1960s. Genetic algorithms are closely related to evolution strategies, which were developed independently at about the same time in Germany by Ingo Rechenburg and Hans-Paul Schwefel. For genetic algorithms, the search is primarily driven by the use of recombination—a mechanism of exchange of information between solutions to 'breed' new ones, whereas evolution strategies principally use mutation—a process of randomly modifying solutions. Although these different approaches were developed independently, and with different directions in mind, recent work has incorporated ideas from both traditions—narrowing the differences between the two. The discussion here, however, focuses on genetic algorithms. For more information on evolution strategies, see references [7–9].

2.3.1. Genetic algorithms

The name 'genetic algorithm' comes from the analogy between the encoding of candidate solutions as a sequence of simple components, and the genetic structure of a chromosome. Continuing with this analogy, solutions are often referred to as *individuals* or *chromosomes*. The components of the solution are sometimes referred to as *genes*, with the possible values for each component called *alleles*, and their position in the sequence the *locus*. Furthermore, the actual encoded structure of the solution for manipulation by the genetic algorithm is called the *genotype*, with the decoded structure known as the *phenotype*. For many applications, the genotype is simply a string of binary digits (this issue will be revisited in the context of test data generation). For example, a vector of three integers <112, 255, 52> in the range [0, 255] might be represented as <01110000, 11111111, 00110100>. For real values, a decision must be made on the precision to be used and what mapping should be used to the binary strings. One possibility, for example, is to scale real values onto integer values according to the required precision, and then use an integer encoding.

Genetic algorithms maintain a population of solutions rather than just one current solution. Therefore, the search is afforded many starting points, and the chance to sample more of the search space than local searches. The population is iteratively recombined and mutated to evolve successive populations, known as *generations*.

The recombination operator takes two parent solutions and 'breeds' them to produce two new offspring. In one-point recombination, a single crossover point is chosen at random. A recombination of two individuals <0, 255, 0> and <255, 0, 255>, 000000001111111100000000 and 111111110000000011111111 in encoded form, with a single-point crossover chosen to occur at locus 12, would take place as follows:

```
000000001111 | 111100000000   ↣   000000001111000011111111
111111110000 | 000011111111       111111110000111100000000
```

This produces two offspring: <0, 240, 255> and <255, 15, 0>.

Various selection mechanisms can be used to decide which individuals should be used to create offspring for the next generation. Key to this is the concept of the 'fitness' of individuals. The fitness of an individual can be the value obtained directly from the objective function, or this value scaled in some way. The idea of selection is to favour the fitter individuals, in the hope of breeding fitter offspring. However, too strong a bias towards the best individuals will result in their dominance of future generations, thus reducing diversity and increasing the chance of premature convergence on one area of the search space. Conversely, too weak a strategy will result in too much exploration, and not enough evolution for the search to make substantial progress.

Holland's original genetic algorithm [10] used *fitness-proportionate selection*. In this selection mechanism, the expected number of times an individual is selected for reproduction is proportionate to the individual's fitness in comparison with the rest of the population. The process is analogous to the use of a roulette wheel. Each individual is allocated a slice of the wheel in proportion to its fitness. The wheel is then spun N times in order to pick N parents. At the end of each spin, the position of the wheel marker denotes an individual selected to be a parent for the next generation. Fitness-proportionate selection has difficulties in maintaining a constant *selective pressure* throughout the search. Selective pressure is the probability of the best individual being selected compared with the average probability of selection of all individuals. In the first few generations of the search, fitness variance is usually high. With fitness-proportionate selection, selective pressure will also be high, since the most highly fit individuals will be granted the greatest opportunities to become parents. This can lead to premature convergence. Also in later generations, when fitness values amongst individuals are similar and the fitness variance of the population is correspondingly low, selective pressure is also low. This can lead to stagnation of the search.

Linear ranking of individuals is a technique which proposes to circumvent this problem. Individuals are sorted by fitness, with selection performed according to rank, rather than through the direct use of fitness values. A linear ranking mechanism with bias Z, where $1 < Z \leq 2$, allocates a selective bias of Z to the top individual, a bias of 1.0 to the median individual, and $2 - Z$ to the bottom individual. With a constant bias applied throughout the search, selective pressure is more constant and controlled [11].

Tournament selection [12] is a noisy but fast rank selection algorithm. The population does not need to be sorted into fitness order. Two individuals are chosen at random from the population. A random number, $0 < r \leq 1$, is then chosen. If r is less than p (where p is the probability of the better individual being selected), the fitter of the two individuals 'wins' and is chosen to be a parent, otherwise the less fit individual is chosen. The competing individuals are returned to the population for further possible selection. This is repeated N times until the required number of parents have been selected. In all probability, every individual is sampled twice, with the best individual selected for reproduction twice, the median individual once, with the worst individual remaining unselected. The resulting selective bias is dependent on p. If $p = 1$, then in all probability a ranking with a bias of 2.0 towards the best individual is produced. If $0.5 < p \leq 1$, then the bias is less than 2.0.

Once the set of parents has been selected, recombination can take place to form the next generation. Crossover is applied to individuals selected at random with a probability p_c (referred to as the *crossover rate* or *crossover probability*). If crossover takes place, the offspring are inserted into the new population. If crossover does not take place, the parents are simply copied into the new population. After recombination, a stage of mutation is employed, which is responsible for introducing or reintroducing genetic material into the search, in the interests of maintaining diversification. This is

```
Randomly generate or seed initial population P
Repeat
    Evaluate fitness of each individual in P
    Select parents from P according to selection mechanism
    Recombine parents to form new offspring
    Construct new population P′ from parents and offspring
    Mutate P′
    P ← P′
Until Stopping Condition Reached
```

Figure 3. High-level description of a genetic algorithm.

usually achieved by flipping bits of the binary strings at some low probability rate p_m, which is usually less than 0.01.

A high-level description of a genetic algorithm can be seen in Figure 3. The initial population is generated at random, or *seeded* with pre-set individuals. The search is terminated when some stopping criterion has been met, for example when the number of generations has reached some pre-imposed limit.

2.3.2. *Advanced encodings and operators*

Traditionally, chromosomes are represented as a string of binary digits. A problem with standard binary encoding is the disparity that can occur between solutions that are close to each other in unencoded solution space, but are far apart in the encoded binary representation. For example, in a standard binary encoding, the integer `7` is represented as `0111`, yet `8` is represented as `1000`. Therefore, the crossover and mutation operators must change all four bits to move from one integer value to the neighbouring one. An alternative is the use of a Gray code. A Gray code is a binary representation where adjacent integers are also Hamming distance 1 neighbours in Hamming space. For example, in *Standard Binary Reflected Gray Code*, `7` is represented as `0100`, and `8` as `1100`. Empirical evidence has shown that Gray codes are generally superior to standard binary encodings [13,14].

Goldberg argues that binary representation decomposes the chromosome into the largest number of smallest possible building blocks in order for the recombination and mutation operators to work most effectively [15]. However, this is disputed by Antonisse [16], who advocates the use of more expressive alphabets. Davis [17] supports this view. For nine real-world applications using genetic algorithms over a variety of problem domains, Davis found that real-valued representations always outperformed binary encodings (real-valued encodings are also the representational choice of evolution strategies [9]). Of course, the use of a real-valued encoding raises the question of how crossover and mutation should work. The crossover operator only requires an underlying sequence representation and, as such, can operate as for binary encodings. Possibilities for the mutation operator include the replacement of a real number in the chromosome with a new, randomly generated number. More advanced mutation

operators are based on *real number creep*. These operators sweep across the chromosome, pushing values up and down by a small amount. In this way, an element of local search is incorporated [17].

Genetic algorithms have been successfully applied to a wide range of problems. For introductory texts, see [15,18]. For shorter overviews and tutorials, see references [9,19,20].

3. STRUCTURAL (WHITE-BOX) TESTING

Structural, or white-box testing is the process of deriving tests from the internal structure of the software under test. This section summarizes some of the achievements in automating structural test data generation through the use of metaheuristic techniques. These are compared with earlier related approaches. Before this, some basic concepts are reviewed.

3.1. Basic concepts

Many forms of structural testing make reference to the *control flow graph* (CFG) of the program in question. A CFG for a program F 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 respective unique entry and exit nodes to the graph. Each node $n \in N$ is 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 node n_j. An example of a CFG can be seen for a version of a triangle classification program in Figure 4. The triangle classification program is a benchmark used in many testing papers. Assuming three non-zero, non-negative integer lengths for the sides of a triangle, the program decides if the triangle is isosceles, scalene, equilateral, or invalid. Nodes corresponding to decision statements (for example, an `if` or a `while` statement) are referred to as *branching nodes*. In the triangle example, branching nodes are nodes 1, 5, 9, 13, 16 and 18. Outgoing edges from these nodes are referred to as *branches*. The condition determining whether a branch is taken is referred to as the *branch predicate*. For the true branch from node 1, the branch predicate is `a > b`.

An *input vector* I is a vector $I = (x_1, x_2, \ldots, x_k)$ of input variables to the program F. The domain of an input variable x_i, $1 \leq i \leq k$, is the set of all values that x_i can take on. The *domain* of the program F is the cross product $D = D_{x_1} \times D_{x_2} \times \cdots \times D_{x_k}$ where each D_{x_i} is the domain for the input variable x_i. A *program input* $\mathbf{x}$ is a single point in the k-dimensional input space D, $\mathbf{x} \in D$.

A path P through a CFG is a sequence $P = \langle n_1, n_2, \ldots, n_m \rangle$, such that for all i, $1 \leq i < m$, $(n_i, n_{i+1}) \in E$. A path is said to be *feasible* if there exists a program input for which the path is traversed, otherwise the path is said to be *infeasible*.

A *definition* of a variable v is a node which modifies the value of v, for example an assignment statement or an input statement. The variable `type` is defined in the triangle program at node 14. A *use* of a variable v is a node in which v is referenced, for example in an assignment statement, an output statement, or a branch predicate expression. In the triangle classification example, the variables `a` and `b` are used at node 1.

A *definition-clear path* with respect to variable v is a path within which v is not modified. In the triangle example, all paths from node 13 are definition-clear with respect to variables `a`, `b` and `c`. However, no path from node 13 is definition clear with respect to `type`.

The term *control dependency* is used to describe the reliance of a node's execution on the outcome at previous branching nodes [21]. A node z is *post-dominated* by a node y in G if and only if every path

CFG Node	
s	`int tri_type(int a, int b, int c)`
	`{`
	`    int type;`
1	`    if (a > b)`
2-4	`    {   int t = a; a = b; b = t;   }`
5	`    if (a > c)`
6-8	`    {   int t = a; a = c; c = t;   }`
9	`    if (b > c)`
10-12	`    {   int t = b; b = c; c = t;   }`
13	`    if (a + b <= c)`
	`    {`
14	`        type = NOT_A_TRIANGLE;`
	`    }`
	`    else`
	`    {`
15	`        type = SCALENE;`
16	`        if (a == b && b == c)`
	`        {`
17	`            type = EQUILATERAL;`
	`        }`
18	`        else if (a == b || b == c)`
	`        {`
19	`            type = ISOSCELES;`
	`        }`
	`    }`
e	`    return type;`
	`}`

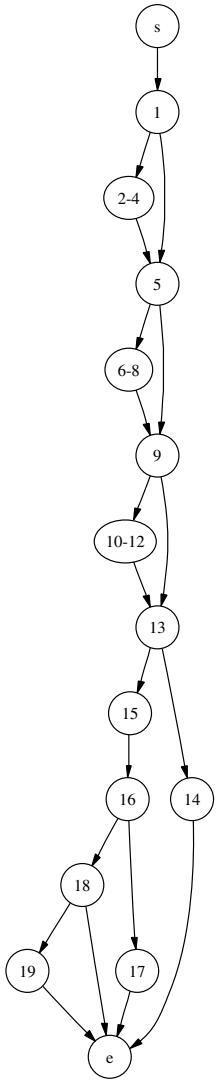


Figure 4. A triangle classification program and its corresponding CFG.

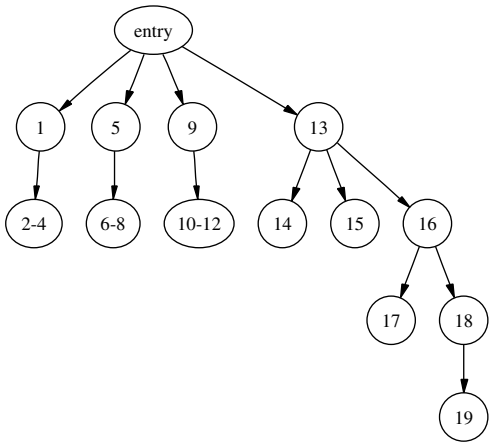


Figure 5. Control dependence graph for the triangle classification program from Figure 4.

from y to the exit node e contains z. Node z post-dominates a branch (y, x) if and only if every path from y to the exit node e through (y, x) contains z. The node z is *control dependent* on y if and only if z post-dominates one of the branches of y, and z does not post-dominate y. In the triangle example, node 17 is control dependent on node 16, which in turn is control dependent on node 13. Node 13 itself has no control dependencies, other than that of the external condition, *entry*, that causes the procedure to be executed. This information can be captured by a control dependence graph. Figure 5 shows the control dependence graph for the triangle program.

The techniques now described have been implemented for experimentation with a variety of programming languages. For consistency, however, all examples here are presented in C.

3.2. Static structural test data generation

Static structural test data generation is based on analysis of the internal structure of the program, without requiring that the program is actually executed.

3.2.1. Symbolic execution

Symbolic execution [22,23] is not the execution of a program in its true sense, but rather the process of assigning expressions to program variables as a path is followed through the code structure. The technique can be used to derive a constraint system in terms of the input variables, which describes the conditions necessary for the traversal of a given path [24–26].

A forward traversal (or forward substitution) of a path can be demonstrated with the triangle classification program in Figure 4. Say the path $\langle s, 1, 5, 9, 10, 11, 12, 13, 14, e\rangle$ is to be executed. The input variables `a`, `b` and `c` are assigned the constant variables `i`, `j` and `k`, respectively. At nodes

1 and 5, the respective false branches are to be taken. Therefore, the first and second constraints of the constraint system for this path are:

(1) `i <= j`
(2) `i <= k`

The path also requires that the true branch be taken from node 9. This requires the addition of a third constraint:

(3) `j > k`

The following expressions are assigned at nodes 10–12, respectively:

```
t = j
b = k
c = t
```

A fourth and final constraint from node 13 then needs to be added. With `a = i`, `b` now equal to `k`, and `c = t = j`, this becomes

(4) `i + k <= j`

Backward path traversal is also possible, starting with the final node and following the path in a reverse manner to the start node. The resulting constraint system is the same as for forward traversal, but no storage is required for the intermediate symbolic expressions of variables. Forward traversal, however, allows for early detection of infeasible paths if the constraints generated are inconsistent. Consider the path $\langle s, 1, 2, 3, 4, 5, 6, 7, 8, 9, 13, \ldots, e\rangle$, which requires that the true branches are taken from nodes 1 and 5, and that the false branch from node 9 is taken. The constraints derived from the branching predicates from the initial section of the path through to node 9 are

(1) `i > j`
(2) `j > k`
(3) `i <= j`

Clearly constraints 1 and 3 are contradictory, indicating that the path is infeasible. Backward traversal would have meant symbolic execution of the path backwards from e through to 13 first, and then backwards through the nodes to node 1 before it would be possible to determine this fact.

Solutions to the constraint system are input data which will execute the path. Constraint satisfaction problems are in general NP-complete [27]. However, if the constraints are linear, linear programming techniques can be applied [24]. Heuristic methods can be used to attempt the finding of a solution where this is not the case. For example, Boyer *et al.* [25] employ hill climbing. Ramamoorthy *et al.* [26] use a trial and error procedure, monitoring the effects of random-value assignments to variables in the constraint system. It is unlikely, however, that this procedure would be efficient for non-trivial programs.

If the test goal is the execution of a particular statement, all paths leading to the statement are explored. This is a problem in the presence of loops, due to the potential number of paths that may need to be examined. In Clarke's test data generator system [24], a path has to be manually selected by the tester. Many generators symbolically simply execute the loop K times, where K is specified by

the tester or chosen by the system [26]. A large number of constraints generated using this method, however, are not satisfiable.

Symbolic execution has several other problems, for example resolving computed storage locations such as array subscripts.

```
a[i] = 0;
a[j] = 1;
if (a[i] > 0)
{
    // perform some action
}
```

In the above code fragment, it is not known in general whether `a[i]` and `a[j]` refer to the same element, because the variables `i` and `j` are not bound to specific values. This information is important, since if `i` and `j` are equal, then the value of `a[i]` in the condition is `1` and the branch predicate evaluates to true. If not, the value of `a[i]` is `0` and the predicate evaluates to false. Boyer *et al.* [25] and Ramamoorthy *et al.* [26] suggest possible solutions to this problem. Both methods significantly increase the complexity and memory requirements of the symbolic execution system. A similar problem occurs with the use of pointers. In the following example, it is not known if `a` and `b` refer to the same location. Without this knowledge, the expression to assign to `c` cannot be determined.

```
*a = 0;
*b = 1;
c = *a;
```

Further difficulties include the handling of procedure calls. A common solution is simply to inline the called procedure into the calling routine [26]. However, the number of paths can grow very rapidly with this approach.

Although any computable function can be written without the use of arrays, pointers or procedure calls, it is not normal practice for programmers to avoid such constructs simply because of the flexibility they offer, and the role they play in reducing the complexity of program code.

3.2.2. Domain reduction

Domain reduction is a test data generation technique that was originally employed as part of constraint-based testing, developed by DeMillo and Offutt [28]. Constraint-based testing builds up constraint systems which describe the given test goal. The solution to this constraint system brings about satisfaction of the goal. The original purpose of constraint-based testing was to generate test data for mutation testing. *Reachability constraints* within the constraint system describe conditions under which a particular statement will be reached. *Necessity constraints* describe the conditions under which a mutant will be killed. Symbolic execution is used to develop the constraints in terms of the input variables. *Domain reduction* is then used to attempt a solution to the constraints. This procedure begins with the domains of each input variable. These can be derived from type or specification information, or be supplied by the tester. The domains are then reduced using information in the constraints, beginning with those involving a relation operator, a variable and a constant, and constraints involving a relation

operator and two variables. Remaining constraints are then simplified by back-substituting values. When no further simplification is possible, the input variable with the smallest remaining domain is chosen, and a random value is assigned to it. The value of this variable is then back-substituted throughout the constraint system, in order to allow further reduction of the domains of remaining variables. If all variables can be assigned values in this manner, then the constraint system will have been satisfied; otherwise the variable assignment stage is repeated, in the hope of this time successfully selecting appropriate random numbers for the variables.

With constraint-based testing, constraints must be computed before they are analysed. Since these constraints are derived using symbolic execution, the method suffers from similar problems involving loops, procedure calls and computed storage locations. Dynamic domain reduction was introduced by Offutt *et al.* [29] with the intent of addressing some of these issues. Although called *dynamic* domain reduction, the technique still has the characteristic that the program is not executed with real input values. As with standard domain reduction, dynamic domain reduction starts with the domains of the input variables. However, in contrast to standard domain reduction, these domains are reduced 'dynamically' during the symbolic execution stage, using constraints composed from branch predicates encountered as the path is followed. If the branch predicate involves a variable comparison, the domains of the input variables responsible for the outcome at the decision are split at some arbitrary 'split point', rather than assigning random input values. For example, if the initial domains of two input variables `y` and `z` are $[-10 \ldots 10]$ and a branch predicate `y < z` is encountered which needs to be executed as true, the domains might be split leaving the domain of `y` to be $[-10 \ldots 0]$ and `z` to be $[1 \ldots 10]$. A back tracking procedure can be used to correct any spurious split points if the execution can only proceed so far down the specified path, and is unable to continue further due to a bad decision made earlier in the reduction process.

Despite setting out to deal with problems traditionally encountered by techniques based on symbolic execution, dynamic domain reduction still suffers with difficulties due to computed storage locations and loops. Furthermore, it is not clear how domain reduction techniques handle non-ordinal variable types, such as enumerations.

3.3. Dynamic structural test data generation

As has already been discussed, the relationship between input data and internal variables for structural test data generation is difficult to analyse statically in the presence of loops and computed storage locations. *Dynamic methods* execute the program in question with some input, and then simply observe the results via some form of program instrumentation. Since array subscripts and pointer values are known at runtime, many of the problems associated with symbolic execution can be circumvented.

3.3.1. Random testing

Random testing simply executes the program with random inputs and then observes the program structures executed. This technique works well for simple programs. However, structures that are only executed with a low probability are often not covered. Consider the triangle classification example once more (Figure 4). The true branch from node 16 requires that the three input values for `a`, `b` and `c` are all equal. Such a branch is unlikely to be executed by chance. Even if the domain of integer values for each variable were limited to values between 1 and 100, the probability of all three variables being

selected with the same value is 1 in 10 000. In such cases a more directed search technique is required to locate test data.

3.3.2. *Applying local search*

Miller and Spooner [30] were the first to combine the results of actual executions of the program with a search technique. Their method was originally designed for the generation of floating-point test data; however, the principles are more widely applicable. The tester selects a path through the program, and then produces a straight-line version of it, containing only that path. Branching statements are then replaced with a 'path constraint' of the form $c_i = 0$; $c_i > 0$; or $c_i \geq 0$; where c_i is an estimate of how close the constraint is to being satisfied. For example, a branch predicate of the form `a == b` might be rearranged into the path constraint $abs(a - b) = 0$. Take the triangle example and the execution of the path $\langle s, 1, 5, 9, 10, 11, 12, 13, 14, e\rangle$ again. The straight-line program with its respective path constraints would be re-arranged as follows:

```
int tri_type(int a, int b, int c)
{
    int type;
    (c1 = (a − b)) > 0
    int t = a; a = b; b = t;
    (c2 = (c − a)) >= 0
    (c3 = (b − c)) >= 0
    (c4 = (c − (a + b))) >= 0
    type = NOT_A_TRIANGLE;
}
```

Note that the values of c_2, c_3 and c_4 are dependent on the computations between c_1 and c_2. However, this information is not required for the derivation of the path constraints, as it would be for the process of test data generation using symbolic execution.

Using these constraints, a function f is constructed. The value of f provides a real-valued estimate of how close all of the constraints are to being satisfied, being negative when one or more of the constraints remains unsatisfied, and positive when all of the constraints are satisfied. Input values of `a`, `b` and `c` are then sought through the use of numerical maximization techniques, which attempt to push the value of f closer and closer to zero, in the hope of eventually making it positive.

Under normal conditions, execution of the complete path is not possible until branch predicates encountered along the path are evaluated in the required manner. However, in the straight-line version of the program, it is possible for runtime errors to occur which would not have been possible in the original program. In the following segment of code, if execution is allowed to proceed down the true branch with values of `i` less than zero, or greater than `size`, an error will be induced, because the array index used in the assignment statement will be out of bounds:

```
if (i >= 0 && i < size)
{
    a[i] = 0;
}
```


Table I. Korel's objective functions for relational predicates.

Relational predicate	f	*rel*
$a > b$	$b - a$	$<$
$a \geq b$	$b - a$	$\leq$
$a < b$	$a - b$	$<$
$a \leq b$	$a - b$	$\leq$
$a = b$	$abs(a - b)$	$=$
$a \neq b$	$-abs(a - b)$	$<$

It was not until 1990 that the ideas of Miller and Spooner were extended by Korel [31] for Pascal programs. In this work, the test data generation procedure worked on an instrumented version of the original program without the need for a straight-line version to be produced. The search targeted the satisfaction of each branch predicate along the path in turn, circumventing issues encountered by the work of Miller and Spooner. To execute some desired path, the program is initially executed with some arbitrary input. If during execution an undesired branch is taken—one which deviates from the desired path—a local search for program inputs is invoked, using an objective function derived from the predicate of the desired, alternative branch. This objective function describes how 'close' the predicate is to being true. The value obtained is referred to as the *branch distance*.

Take the triangle example and the execution of the path $\langle s, 1, 5, 9, 10, 11, 12, 13, 14, e \rangle$ again. If the function is executed with the program input `(a=10, b=20, c=30)`, control flow successfully follows the false branches from nodes 1 and 5. However, control flow diverges away from the intended path down the false branch at node 9. At this point the local search is invoked to change the program inputs so that the alternative true branch is taken. If, in general, the branch predicate is assumed to be of the form *a op b*, where *a* and *b* are arithmetic expressions and *op* is a relational operator, an objective function of the form f *rel* 0 is derived, where f and *rel* are given in Table I. The function is to be minimized, being positive (or zero if *rel* is '$<$') when the current branch predicate for the required branch is false, and negative (or zero if *rel* is '$=$' or '$\leq$') when it is true. For the predicate of the true branch from node 9, the objective function is `c - b > 0`. The value of this function for the program input `(a=10, b=20, c=30)` is `30 - 20 = 10`. The program must be instrumented so that objective values can be computed. This can be performed within the branching expression, for example, as follows:

```
if (eval_obj(9, b, c))
{
    ...
```

Here, the program function `eval_obj` reports branch distances at node 9 using the local values of `b` and `c`. This function will then return a Boolean value corresponding to the evaluation of the original branching expression, in order for program execution to resume as normal.

```
void nested_example(int a, int b, int c)
{
    if (a == b)
        if (b == c)
            if (c < 0)
                // target
}
```

Figure 6. Example with nested structures.

The local search for deriving input values in accordance with the objective function is known as the *alternating variable* method. Each input variable is taken in turn and its value adjusted, keeping the other variable values constant. The first stage of manipulating an input variable is called the *exploratory* phase. This probes the neighbourhood of the variable by increasing and decreasing its original value. If either move leads to an improved objective value, a *pattern* phase is entered. In the pattern phase, a larger move is made in the direction of the improvement. A series of similar moves is made until a minimum for the objective function is found for the variable. The next input variable is then selected for an exploratory phase.

Consider the triangle example again, for which execution had diverged from the intended path at node 9. Decreases and increases of `a` have no effect on the objective value. Therefore, `b` is chosen. A decrease of `b` leads to a worse objective value, but an increase leads to an improvement. The pattern phase is entered for `b`, which will be increased until `b > c`. Suppose the value `31` is reached. The new input vector is now `(a=10, b=31, c=30)`. Control flow now proceeds through branching node 9 as desired; however, execution now diverges away at node 13, since the value of `a + b` at the node is greater than the value of `c`. The local search is invoked again, this time to adjust the input values so that the true branch is taken from node 13, whilst maintaining the already correct sub-path up to this node. The new objective function, derived from the true branch predicate, is `(a + b) - c <= 0`. A decrease of the input value of `b` leads to a violation of the sub-path up to node 9, yet an improved value of the objective function is found for an increase of `b` (since the internal values of `b` and `c` are swapped at nodes 10–12). Eventually, the input vector `(a=10, b=40, c=30)` will be found. This input vector evaluates branching node 13 as true, and the complete path is executed.

As with all local searches, the final result is dependent on the starting solution. Consider the example of Figure 6. If the input is initially selected as `(a=10, b=10, c=10)`, control flow proceeds directly down to the final branching node. However, the variable `c` cannot be changed to a value less than `0`, because the already successful sub-path up to the final branching node will be violated. In this case, the search will fail.

Heuristic search methods have the potential to make moves through variable values that cannot lead to an improvement in the value of the current cost function. This can lead to many wasteful and costly executions of the program. In the triangle example, changing the value of the input variable `c` does not have an effect on branching node 1. In order to make the search more efficient, Korel's work makes use of extra information derived from the program, in the form of an 'influences' graph.

An influences graph is used to detect which input variables are able to influence the outcome at the current branching node, as determined using dynamic data flow analysis. A risk analysis of input variables is also undertaken in order to decide if they could potentially violate the already successful sub-path. For example, at node 5, it is more attractive to manipulate `c` rather than `a` or `b`, since changing `a` or `b` may change the current successful sub-path through node 1.

Gallagher and Narasimhan [32] built on Korel's work for programs written in Ada. In particular, this was the first work to record support for the use of logical connectives within branch predicates. For predicates of the form *A and B*, the overall objective value is formed from the summation of the individual objective values of the expressions *A* and *B*. For predicates of the form *A or B*, the objective value is the minimum value of the individual objective values of the expressions.

3.3.3. The goal-oriented approach

In his paper published in 1992, Korel developed what became known as the goal-oriented approach [33]. All of the techniques concentrate on the execution of a path. For fulfilling a structural coverage criterion like statement coverage, this means a path has to be selected for each individual uncovered statement. The goal-oriented approach removes this requirement. This is achieved through the classification of branches in the CFG of the program with respect to a target node as either *critical*, *semi-critical* or *non-essential*. This can be performed automatically on the basis of the program's CFG.

For branches leaving a node on which the target is control dependent, a *critical branch* is the edge which leads the execution path away from the target node. If control flow is driven down a critical branch, there is no prospect of the target being reached. Therefore, an objective function, of the form outlined in the previous section, is associated with the branch predicate of the alternative branch. The alternating variable search method is then employed to seek inputs so the alternative branch is taken instead. If the required inputs cannot be found, the overall process terminates, with the target remaining unexecuted.

A *semi-critical* branch is one which leads to the target node, but only via the backward edge of a loop. The alternative branch from the same branching node leads directly to the target node. In the case where the execution is driven down a semi-critical branch, the alternating variable method is again invoked to seek inputs for the execution of the alternative branch. If suitable input values cannot be found, however, the process does not terminate. Execution is allowed to flow down the semi-critical branch, in the hope of taking the alternative branch in the next iteration of the loop.

Finally, a *non-essential* branch is neither critical or semi-critical. Non-essential branches do not determine whether the target will be reached, regardless of their position in the CFG. Therefore, execution is allowed to proceed unhindered through these branches.

Take the example of Figure 7, with the target being the execution of node 5. The classification of each branch can be seen from the CFG in Figure 8. The false branches from nodes 1 and 3 are critical since node 5 cannot be reached if they are executed. The false branch from condition 4 is semi-critical because, although control flow diverges away from the target at this point, the target may still be reached in the next iteration of the loop. If the input vector is `(a=0)` the false branch from condition 1 is taken, and so the search procedure is invoked to change the value of `a`. Control flow proceeds down through the true branch from node 1, but from node 4 the false branch is taken. However, the search cannot change the outcome at this branch, and so the flow of control is allowed to continue around the loop a further nine times upon which the true branch from node 4 is taken, and the target is reached.

```
CFG
Node
  s     void goal_oriented_example(int a)
        {
  1         if (a > 0)
            {
  2             int b = 10;
  3             while (b > 0)
                {
  4                 if (b == 1)
                    {
  5                     // target
                    }

  6                 b --;
                }
            }

  e         return;
        }
```

Figure 7. Example for the goal-oriented approach.

As the goal-oriented method also employs the alternating variable local search, it suffers from similar problems to those of Korel's original approach. The removal of the requirement to select a path, although relieving some effort on behalf of the tester, introduces new ways in which the test data search can fail. Take the example of Figure 9 and the execution of the true branch from node 4. The true branch is only taken for objective values less than or equal to zero. Consider what happens when the initial input vector is selected so that `a` is less than zero (approximately half of the input domain). With such a starting point, the critical false branch from node 4 is taken. The search will fail, since small exploratory moves of `a` will have no effect on the objective function associated with this condition, which is concerned only with the value of `b`. The landscape of the objective function in this region of the search space is flat (Figure 10).

In this example, one could attribute the failure to the use of a local search technique. A global search technique such as a genetic algorithm is likely to sample the input domain more thoroughly and find the required value of `a`. The local search could incorporate a series of restarts. However, it may be that the required path up to the target node is found with some very low probability. Even genetic algorithms will have trouble with these search spaces (see Section 3.5.4). Korel noted that this situation could be avoided if the data dependencies of the test goal were also taken into account by the search, and attempted to address this issue with the *chaining approach*.

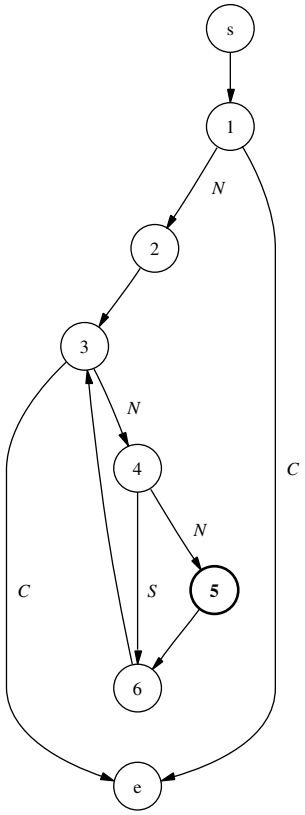


Figure 8. CFG and branch classification for program in Figure 7. Node 5 is the target. C represents a critical branch; S, a semi-critical branch; and N, a non-essential branch.

3.3.4. *The chaining approach*

The chaining approach [34,35] uses the concept of an *event sequence* as an intermediate means of deciding the type of path required for execution up to the target node. An event sequence is basically a succession of program nodes that are to be executed. The initial event sequence consists of just the start node and target node. Extra nodes are then inserted into this event sequence when the test data search encounters difficulties.

An event sequence can be formally described as a sequence of events $\langle e_1, e_2, \ldots, e_k \rangle$ where each event is a tuple $e_i = (n_i, C_i)$ where $n_i \in N$ is a program node and C_i is a set of variables referred to as a constraint set [35]. For every two adjacent events $e_i = (n_i, C_i)$ and $e_{i+1} = (n_{i+1}, C_{i+1})$, no variables in the constraint set should be modified. That is to say, a definition-clear path must be taken from n_i to n_{i+1} with respect to each variable in C_i.

CFG Node	
s	`void chaining_approach_example(int a)`
	`{`
1	`int b = 0;`
2	`if (a > 0)`
	`{`
3	`b = a;`
	`}`
4	`if (b >= 10)`
	`{`
5	`// target`
	`}`
	`// ...`
	`}`

Figure 9. Example for the chaining approach.

For the example in Figure 9, the target is the execution of node 5. The initial event sequence is therefore:

$$\langle (s, \phi), (5, \phi) \rangle$$

For every two adjacent events $e_i = (n_i, C_i)$ and $e_{i+1} = (n_{i+1}, C_{i+1})$ in each event sequence E, the branches of the program are classed as either critical, semi-critical or non-essential. If there does not exist a definition-clear path with respect to the variables in C_i from n_i to n_{i+1} through branch (p, q), where p and q are program nodes, yet such a path does exist from the alternate branch from p, the branch (p, q) is declared critical. A branch (p, q) is semi-critical if it is not critical, n_{i+1} is control dependent on p, and there does not exist an acyclic definition-clear path from p to n_{i+1} with respect to C_i though (p, q). All other branches are declared as non-essential. As with the goal-oriented approach, the flow of control should not take a critical branch. If one is taken, the alternating variable method is used to try and change the execution at the branching node. Semi-critical branches are preferably avoided, and non-essential branches are ignored.

Recall from the last section, the search for inputs to execute the branching node 4 as true for the program of Figure 9 can fail when the value of `a` is negative, e.g. `-10`. In executing the initial event sequence, the false branch from node 4 is critical. However, the local search is unable to find an input value of `a` so that the alternative true branch is taken, since exploratory moves from `-10` yield no change in values of the objective function associated with this branch. When inputs cannot be found to change the flow of control so that a critical branch (p, q) is avoided, p is 'declared' as a problem

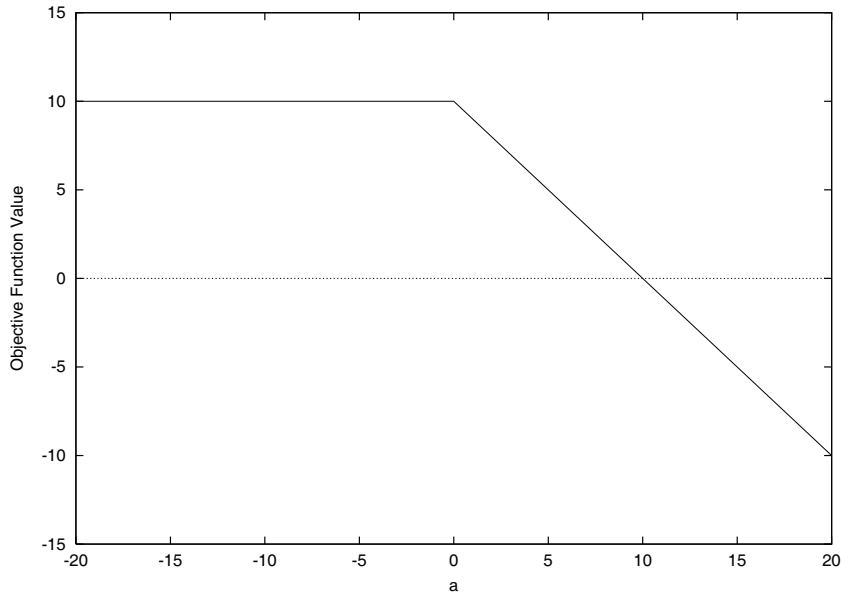


Figure 10. Objective function landscape for execution of node 4 as true for Figure 9.

node, for which new event sequences can be generated. In such instances the chaining approach looks for last definition statements of variables used at the problem node. In the example, the variable used at node 4 is the variable `b`. This variable is defined at nodes 1 and 3. Therefore, two different event sequences are generated, one inserting an event where node 1 should be executed and one where node 3 should be executed, i.e.

$$(1)\ \langle (s, \phi), (1, \{b\}), (4, \phi), (5, \phi)\rangle \quad \text{and}$$
$$(2)\ \langle (s, \phi), (3, \{b\}), (4, \phi), (5, \phi)\rangle$$

The constraint set for both events includes the variable `b`, since a reassignment to `b` before node 4 would destroy the effect of the inserted event node.

The first event sequence executes exactly the same path for which inputs could not be found. The outcome, however, is different for the second sequence. Assume the input vector is still `(a = -10)`. Control flow is driven down the critical false branch at node 2. The alternating variable method is used to try and amend this. Increments in `a` have a positive effect on the objective function associated with the true branch. Eventually, the input `(a = 1)` is found. Flow of control is now driven down the critical false branch at node 4. However, exploratory moves of `a` now have an effect on the objective function associated with this branch. An increment of `a` leads to an improvement in the cost function, until eventually the vector `(a = 10)` can be found.

The chaining approach organizes the generated event sequences in a tree. At the root of the tree is the initial event sequence. The first level contains the event sequences generated as a result of the

Table II. Tracey's objective functions for relational predicates. The value K, $K > 0$, refers to a constant which is always added if the term is not true.

Relational predicate	Objective function *obj*
Boolean	if *TRUE* then 0 else K
$a = b$	if $abs(a - b) = 0$ then 0 else $abs(a - b) + K$
$a \neq b$	if $abs(a - b) \neq 0$ then 0 else K
$a < b$	if $a - b < 0$ then 0 else $(a - b) + K$
$a \leq b$	if $a - b \leq 0$ then 0 else $(a - b) + K$
$a > b$	if $b - a < 0$ then 0 else $(b - a) + K$
$a \geq b$	if $b - a \leq 0$ then 0 else $(b - a) + K$
$\neg a$	Negation is moved inwards and propagated over a

first problem node. In more complicated examples, further problem nodes could be encountered on route to executing some last definition node inserted into the sequence. In such instances, the chaining approach backtracks further and looks for last definition statements for variables used at these new problem nodes. These additional event sequences are added to the tree. The tree is explored in a depth-first fashion, to some specified depth limit.

The chaining approach can generate test data for a larger class of programs than the goal-oriented approach. However, search times increase, and the local search employed can still become trapped in difficult search spaces.

3.4. Applying simulated annealing

The work of Tracey and co-authors [36,37] applies simulated annealing to structural test data generation in the hope of overcoming some of the problems associated with the application of local search. In this work, test data can be generated for specific paths, or for specific statements or branches.

In order to apply simulated annealing, a neighbourhood structure has to be defined for the various different input variable types. For integer and real variables, the neighbourhood is simply a defined range of values around each individual value. Since the ordering of values is not significant for Boolean and enumerated types, all values for these variables are considered as neighbours.

The objective function is simply the branch distance of the required branch when control flow diverges away from the intended path, or away from the target structure down a critical branch. The objective functions used (Table II) are in principle identical to those employed by Korel, except the use of a non-zero positive failure constant K—which is always added if the branch predicate evaluates to false—removes the need to use a relation *rel* within the function. In this way, the objective function always returns a value above zero if the predicate is false, and zero when it is true.

In order to reduce the chances of the search becoming stuck in local optima, Tracey drops the constraint employed by Korel that the newly generated solution must conform to an already successful sub-path. However, the means of doing this results in the search losing some information about its progress. This is because solutions which diverge away from the target down earlier critical branches

CFG Node	
s	`void landscape_example(int i, int j)`
	`{`
1	`if (i >= 10 && i <= 20)`
	`{`
2	`if (j >= 0 && j <= 10)`
	`{`
3	`// target statement`
	`// ...`
	`}`
	`}`
	`}`

Figure 11. Example for comparing objective functions.

are assigned similar objective values to those diverging away at a later stage. This can be demonstrated with the example of Figure 11. For the target statement at node 3, the false branches from nodes 1 and 2 are critical. Under Korel's scheme, if the current solution is `(i=10, j=-1)`, diverging down the critical branch from node 2, the vector `(i=9, j=-1)` would not be given consideration, because the already successful sub-path up to node 2 is violated. This is due to the fact that this input vector takes the earlier critical branch at node 1. However, in Tracey's method, a move can take place between solutions and, furthermore, the solutions are rewarded identical objective values, since the distance values taken at the different branching nodes are the same.

3.5. Applying evolutionary algorithms

The application of evolutionary algorithms to test data generation is often referred to in the literature as *evolutionary testing* (for example, references [38–40]). The first work applying evolutionary algorithms to generate structural test data is that of Xanthakis *et al.* [41]. Up until this point, work on structural test data generation had largely focused on finding input data for specific paths or individual structures with programs, such as branches or statements. Initially, however, techniques using genetic algorithms took slightly different directions.

3.5.1. A classification of techniques

Different techniques applying evolutionary algorithms to structural test data generation can be categorized on the basis of objective function construction (Figure 12).

Coverage-oriented approaches reward individuals on the basis of covered program structures. In the work of Roper [42], an individual is rewarded on the basis of the number of structures executed

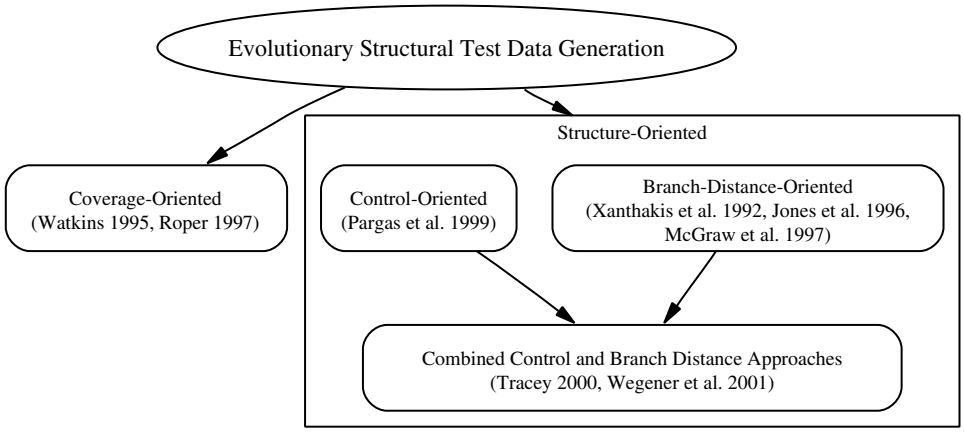


Figure 12. Classification of dynamic structural test data generation techniques using evolutionary algorithms.

in accordance with the coverage criterion. Under this scheme, however, the search tends to reward individuals that execute the longest paths through the test object. Guidance is not given for structures that are unlikely to be covered by chance, for example deeply nested structures, or branch predicates that are only true when an input variable has to be a specific value from a large domain.

The work of Watkins [43] attempts to obtain full path coverage for programs. The objective function penalizes individuals that follow already covered paths, by assigning a value that is the inverse of the number of times the path has already been executed during the search. The direction of the search, therefore, is under constant adaptation. However, the penalization of covered paths, in itself, provides little guidance to the discovery of new, previously unfound paths. The results show that in comparison with random testing, the genetic algorithm approach requires an order of magnitude fewer tests to achieve path coverage for two experimental programs. However, both of these programs are of a simple nature, containing no loops. Furthermore, in this work the input domains were artificially restricted for the search.

In general, the problem with coverage-oriented approaches is the lack of guidance provided for structures which are only executed with values from a small portion of the overall input domain. Therefore, it is difficult to expect full coverage with these techniques for any non-trivial program.

Structure-oriented approaches follow similar lines to the earlier work of Korel, and take a 'divide and conquer' approach to obtaining full coverage. A separate search is undertaken for each uncovered structure required by the coverage criterion. Structure-oriented techniques differ in the type of information used by the objective function. These can be categorized as either *branch-distance-oriented*, *control-oriented*, or *combined* approaches.

Branch-distance-oriented approaches exploit information from branch predicates in a similar style to earlier work by Miller and Spooner, and later Korel. In the work of Xanthakis *et al.* [41], genetic

algorithms are employed to generate test data for structures not covered by random search. A path is chosen, and the relevant branch predicates are extracted from the program. The genetic algorithm is then used to find input data that satisfy all the branch predicates at once, with the objective function summing branch distance values. However, this scheme suffers from similar problems to those encountered by Miller and Spooner. Furthermore, the need to select a path is a burden on the tester. In the work of Jones *et al.* [44] for obtaining branch coverage, a path does not need to be selected. The objective function is simply formed from the branch distance of the required branch. However, no guidance is provided so that the branch is actually reached within the program structure in the first place. McGraw *et al.* [45] alleviate this problem for condition coverage by delaying an attempt to satisfy a condition within a branching expression until previous individuals have already been found which reach the branching node in question. The initial generation for the target condition is then seeded with these individuals. This scheme, however, is inefficient if test data are required for the coverage of one, specific condition.

The earlier work of Korel had already removed the need for the tester to select a path. Since new test data considered by the search had to conform to the successful sub-path already found, explicit control-oriented information regarding the target did not need to be included in the objective function. However, such rigid constraints increase the chances of the search becoming stuck in local optima, and it would be better if more feedback could be provided via the objective function. This is the problem addressed by *control-oriented* approaches.

With *control-oriented* approaches, the objective function considers the branching nodes that need to be executed in some desired way in order to bring about execution of the desired structure. The approach of Jones *et al.* [44] to loop testing falls into this category. Here, the objective function is simply the difference between the actual and desired number of iterations. In the work of Pargas *et al.* [46], for statement and branch coverage, the control dependence graph of the test object is used. The sequence of control-dependent nodes is identified for each structure. These are the branching nodes that must be executed with a specific outcome in order for the structure to be reached. The objective value of an individual is simply assigned as the number of control-dependent nodes executed as intended. Recall that the branch leading away from the target at a control-dependent node is identified as a *critical branch* in Korel's work. The measure used by Pargas *et al.* is therefore equivalent to the number of critical branches successfully avoided by the individual.

The problem with using control information only for the purposes of the objective function are the plateaux that form on the objective function landscape. The objective function gives no guidance as to how to change the flow of execution at control-dependent nodes, since no distance information is exploited from branch predicates. Take the simple example of Figure 11. The target is node 3, which is control dependent on node 2, which in turn is control dependent on node 1. Let *dependent* be the number of control-dependent nodes for the current target, and *executed* the number of control-dependent nodes successfully executed in the required manner. A minimizing version of the objective function of Pargas *et al.*, can be computed as (*dependent* − *executed*). However, in this scheme, *every* individual diverging away from the target at node 1 receives an objective value of 2, with *every* individual diverging at node 2 receiving a value of 1. The landscape for the minimizing version of the objective function for the example is seen in Figure 13. This landscape has three plateaux. For individuals not satisfying one or more of the branch predicates, no guidance is given as to how to descend down the landscape to solutions that are closer to executing the target. Along these horizontal planes, the search becomes random.

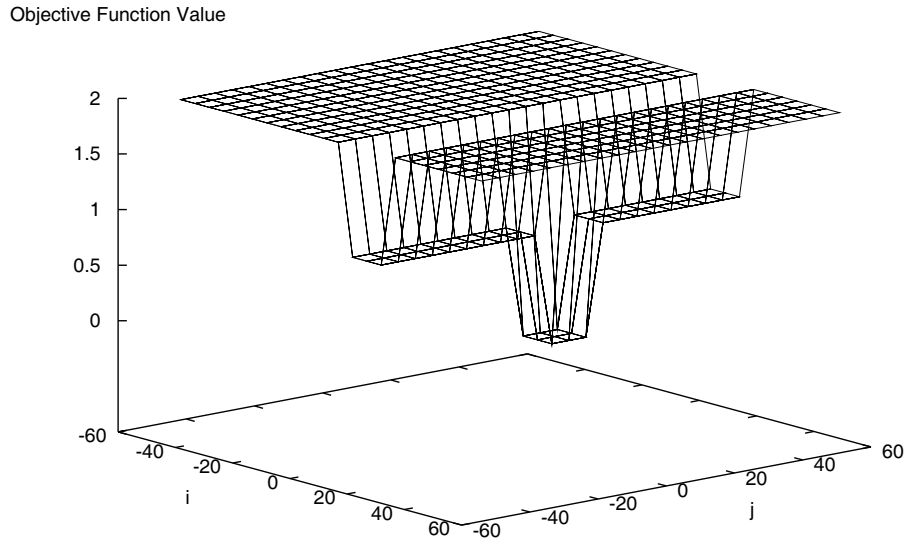


Figure 13. Objective function landscape of Pargas *et al.* [46] for example given in Figure 11.

Combined approaches make use of both branch distance and control information for the objective function. The work of Tracey [47] builds on previous work which used simulated annealing. The strategy for combining both techniques is as follows. The control-dependent nodes for the target structure are identified. If an individual takes a critical branch from one of these nodes, a distance calculation is performed using the branch predicate of the required, alternative branch. This is computed using the functions of Table II (and Table III for *and* and *or* logical connectives). Tracey then uses the number of successfully executed control-dependent nodes to scale branch distance values. Let *branch_dist* be the branch distance calculation performed at the branching node where a critical branch was taken. The formula used by Tracey for computing the objective function is

$$\left(\frac{executed}{dependent}\right) \times branch\_dist$$

Unfortunately, this scheme can lead to unnecessary local optima in the objective function landscape. For the example of Figure 11, this is evident by the valleys in the objective function landscape along $i = 9$ and $i = 21$ where $-3 \leq j$ and $j \geq 13$, as seen in Figure 14.

Wegener *et al.* [38,48] map branch distance values *branch_dist* logarithmically into the range [0, 1] (call this *m_branch_dist*). The minimizing objective function is zero if the target structure is executed, otherwise, the objective value is computed as

$$(dependent - executed - 1) + m\_branch\_dist$$

The $(dependent - executed - 1)$ sub-calculation is referred to as the *approximation level* or, perhaps more appropriately, the *approach* level attained by the individual [38,48]. The resulting objective

Table III. Tracey's cost functions for logical connectives, where $obj(c)$ is the individual cost of connective c.

Connective	Objective function *obj*
$a \wedge b$	$obj(a) + obj(b)$
$a \vee b$	$min(obj(a), obj(b))$
$a \Rightarrow b$	$obj(\neg a \vee b)$
	$\equiv min(obj(\neg a), obj(b))$
$a \Leftrightarrow b$	$obj((a \Rightarrow b) \wedge (b \Rightarrow a))$
	$\equiv obj((a \wedge b) \vee (\neg a \wedge \neg b))$
	$\equiv min((obj(a) + obj(b)), (obj(\neg a) + obj(\neg b)))$
a xor b	$obj((a \wedge \neg b) \vee (\neg a \wedge b))$
	$\equiv min((obj(a) + obj(\neg b)), (obj(\neg a) + obj(b)))$

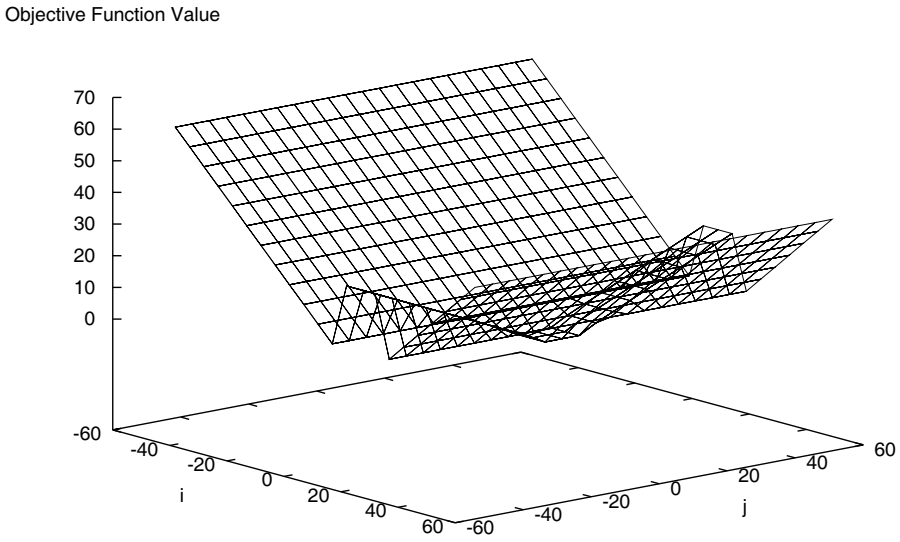


Figure 14. Objective function landscape of Tracey [47] for example given in Figure 11.

function landscape has a similar form to that of Pargas *et al.* (Figure 15). However, the extra information provided by the branch distance calculation prevents the formation of plateaux at each approach level. For the example, the result is a sweeping landscape from each level to the next level downwards.

3.5.2. *Objective functions for different structural coverage criteria*

The work detailed so far for structural test data generation has mainly addressed statement, branch or condition coverage. In the work of Wegener *et al.* [48], several new structure-oriented objective

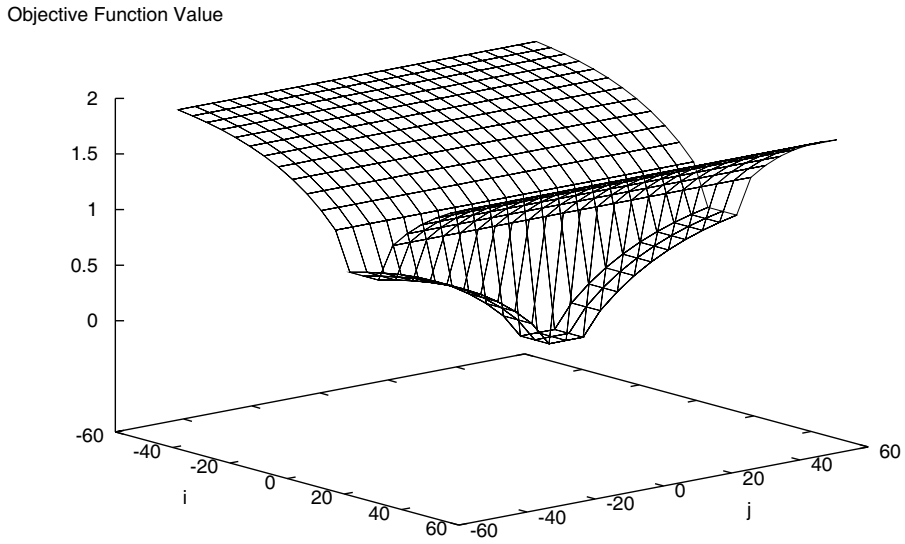


Figure 15. Objective function landscape of Wegener *et al.* [48] for example given in Figure 11.

functions were introduced for previously unexplored coverage types. For this purpose, structural criteria are divided into four categories:

- node-oriented;
- path-oriented;
- node–path-oriented;
- node–node-oriented.

The basic form of the (minimizing) objective function is

$$approach\_level + m\_branch\_dist$$

The strategy in which *approach_level* and *m_branch_dist* are computed varies according to the coverage type in question.

Node-oriented criteria aim to cover specific nodes of the CFG, for example statement coverage. The strategy for node-oriented methods was discussed in the last section. The approach level is calculated on the basis of the number of control-dependent nodes for the target lying between nodes covered by the individual and the target node itself. At the point where control flow diverges down a critical branch, the branch distance is calculated using the predicate of the alternative branch.

Path-oriented criteria require the execution of specific paths through the CFG. There are two possible ways to calculate the objective function. One method is to calculate the approach level on the basis of the length of identical initial path section, with the branch distance calculation performed using the predicate at the first diverging branch. An alternative strategy considers all identical path sections for

CFG Node	
s	`void loop_example(int i)`
	`{`
	`int n;`
1	`for (n=0; n <= 10; n++)`
	`{`
2	`if (n == 10 && i == 0)`
	`{`
3	`// target statement`
	`}`
	`}`
e	`}`

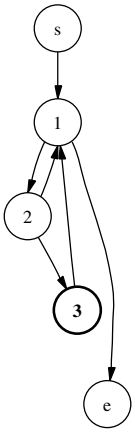


Figure 16. Loop example, with CFG.

the approach level, with the branch distance calculation an accumulation of distance calculations made at each point of divergence from the intended path. Wegener *et al.* report superior results with the latter method [48].

Node–path-oriented criteria include branch coverage and LCSAJ (linear code sequence and jump) coverage, where a node and a specific subsequent path must be executed. The objective function is a combined node-oriented and path-oriented calculation. Calculations for individuals not reaching the initial node are treated as for node-oriented criteria. For individuals reaching the initial node, a path-oriented calculation is additionally applied.

Node–node-oriented criteria aim to execute a certain sequence of nodes through the CFG, without the specification of a concrete path between each node. This includes data-flow-oriented coverage types such as *all-defs* and *all-uses* criteria. In this case, the objective function is a cumulative node-oriented strategy. Calculations for individuals failing to reach the first node are carried out as for node-oriented methods, with individuals reaching the subsequent node having additional calculations carried out at these further nodes.

3.5.3. Control-related problems for objective functions

The provision of guidance to structures nested within loops presents a problem which can be demonstrated with Figure 16. The target is the execution of node 3. However, node 3 is not control dependent on node 2, because paths taking the false branch from node 2 can still execute node 3 in subsequent iterations of the loop. Consequently, the search does not receive guidance regarding the fact that the true branch from node 2 needs to be taken for the target statement to be reached. This results in poor search performance. The approach taken by Baresel *et al.* [49] is to treat branches that miss the target in iterations of the loop as if they were critical branches (recall that these branches are classed as *semi-critical* in Korel's work). Thus, node 3 is treated as if it were control dependent on

CFG Node	
	`switch(a)`
	`{`
1	`case 1:`
2	`if (cond_1)`
	`return;`
3	`if (cond_2)`
	`break;`
4	`case 2:`
5	`if (cond_3)`
	`break;`
	`return;`
	`}`
6	`// target statement`

Figure 17. Example demonstrating problems with unstructured control flow.

node 2. This also appears to be the approach taken by Tracey [47]. However, this leads to penalization of individuals in the first iteration of the loop. In the example, if the input variable `i` is `1`, the objective value is taken in the first iteration, when `n` is `0`. However, the individual is closest to executing the target statement in the last iteration of the loop, when `n` is `10`. Furthermore, when the input value of `i` is `0`, the individual will be deemed to have missed the target, when the target is actually executed in the last iteration of the loop. In order to circumvent this problem, Tracey [47] examines the branch distance during each iteration of the loop and uses the minimum branch distance obtained for the purposes of computing the final objective value.

A further problem is the assignment of approach levels for some classes of program with unstructured control flow. Baresel *et al.* [49] present the example of Figure 17. The target of the search is node 6. However, there are three different control-dependent paths through to node 6 from node 1 (Figure 18), and two control-dependent paths from node 2. Consequently, there are two approach-level possibilities for node 1 (since two of the paths are of the same length), and two possibilities for node 2. Two plausible solutions to this problem include *optimistic* and *pessimistic* approach-level allocation strategies. In an optimistic strategy, a control-dependent branching node is allocated its approach level on the basis of the shortest control-dependent path from itself to the target node. In this way node 2 is assigned an approach level of 1 on the basis of the direct path through to 6, thereby receiving the same level as node 5. In a pessimistic strategy, a branching node is allocated its approach level on the basis of the longest control-dependent path to the target node. In this scheme, node 2 would be assigned an approach level of 3 on the basis of the path through nodes 3 and 5. Both optimistic and pessimistic schemes were put to the test in initial experiments by Baresel *et al.* [49]. Whilst they show that the

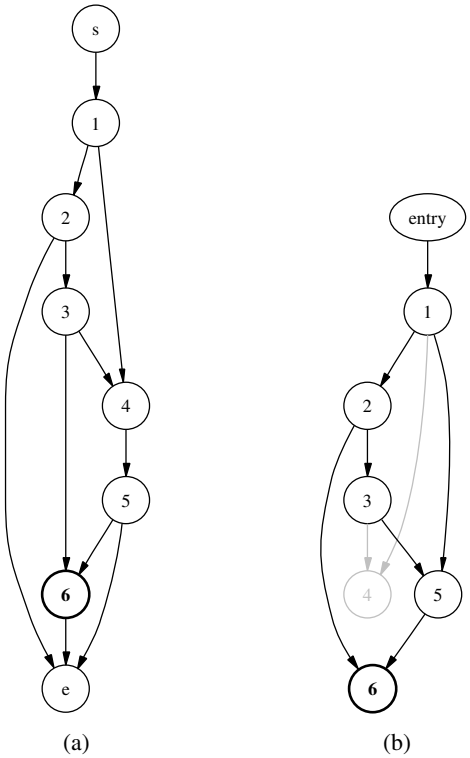


Figure 18. (a) CFG and (b) control dependence graph for example given in Figure 17.

different schemes have different effects on the progress of the search, they were unable to conclude from the experiments which strategy works best in general. Thus, this problem is still open to question.

3.5.4. Branch-distance-related problems for objective functions

Although global search techniques are more robust than local searches in objective function landscapes containing local optima and plateaux, they will still struggle in hostile search landscapes containing large plateaux or several local optima. In particular, plateaux can be induced on the search space through the use of 'flag' variables in branch predicates. A flag is simply a Boolean variable. When flag variables are involved in branch predicates, the resulting objective function landscape consists of two plateaux—one for the true value and one for the false value. In such situations, the evolutionary search performs no better than a random search.

Figure 19 demonstrates this with an example. For the true branch to be executed, the flag must be true. However, the objective function gives no guidance on how the true value is brought about. The plateau induced on the objective function landscape can be seen in Figure 20.

```
flag = (d == 0);

if (flag)
    result = 0;
else
    result = n / d;
```

Figure 19. Flag example.

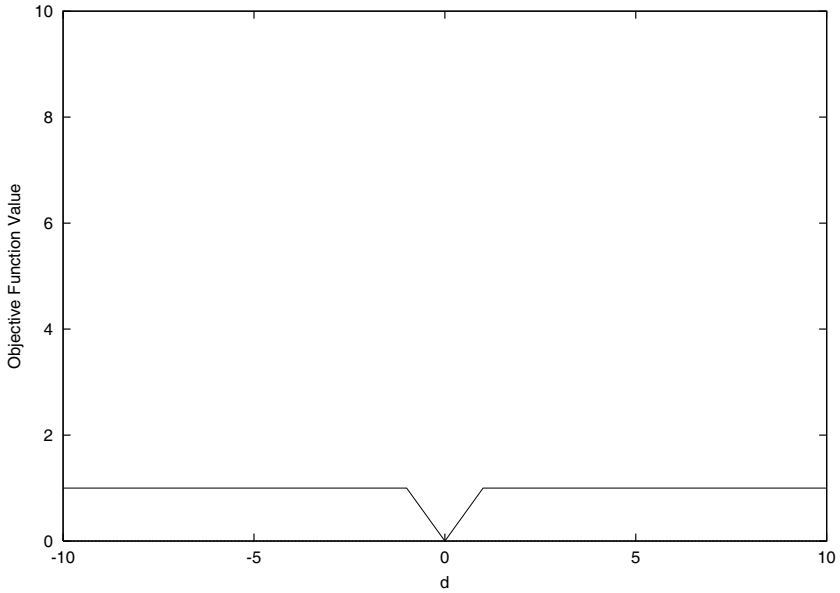


Figure 20. Objective function landscape for the flag example.

Bottaci [50] proposes a solution for a special case of flag problems similar in form to the example of Figure 19, where the value of the flag is determined by a predicate. In this work it is suggested that the predicate used for the distance calculation is substituted by the predicate used in assigning the flag value. Essentially the objective function landscape becomes that of Figure 21, which provides more guidance to the required test data. However, flags are more commonly assigned constant true or false values, as seen in Figure 22. In this case the expression leading to the true assignment is used to control the assignment. (Note that the true branch from node 4 would have already been executed if test data had already been found to execute the preceding true branch from node 2. However, for simplicity, this possibility is ignored for the purposes of this example, and others in this section.)

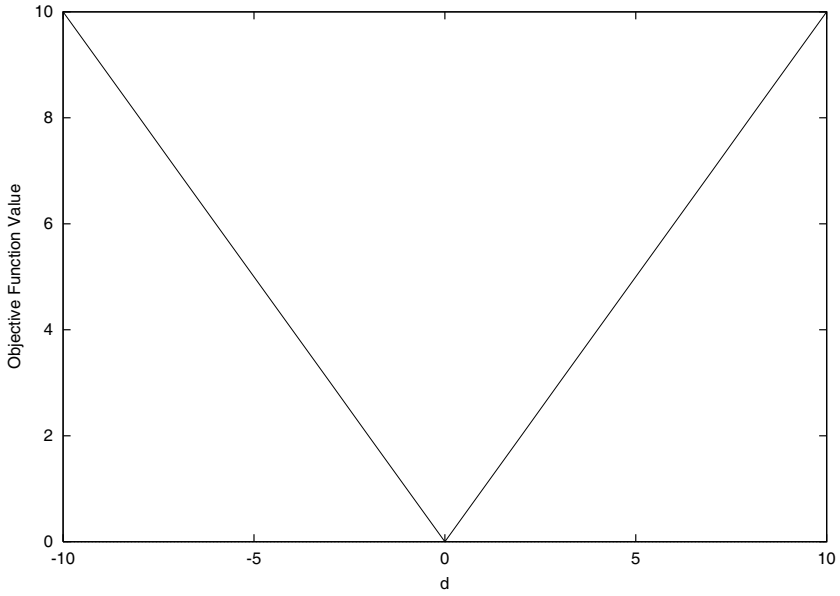


Figure 21. Objective function landscape for the predicate $d = 0$.

CFG Node	
1	`flag = false;`
2	`if (d == 0)`
3	`    flag = true;`
4	`if (flag)`
5	`    result = 0;`
	`else`
6	`    result = n / d;`

Figure 22. Alternative version of the flag example.

```
flag = false;
if (d == 0)
    flag = true;

if (d == 0)
    result = 0;
else
    result = n / d;
```

Figure 23. Flag removed from branch predicates of Figure 22 via program transformation.

Harman *et al.* [39] suggest the use of a program transformation to remove flag variables from branch predicates, replacing them with the expression that led to their determination. In the transformed version of the program, the branch predicate is flag-free, and consequently plateaux induced by the flag are also removed. Figure 23 shows a possible transformation of the program of Figure 22. Note that although the flag is removed from the branch predicate, it otherwise remains present in the program, in case it has a future purpose in a later statement. The objective function at the new branch predicate now has the more useful landscape of that of Figure 21. The transformed program is merely a means to an end, and can be discarded once the required test data have been found. A disadvantage of the approach is that it cannot yet transform programs where flags are involved in loops.

The approach of Baresel and Sthamer [51] is to identify a sequence of nodes to be executed prior to the branch predicate containing the flag. For the example of Figure 22 where the true branch from node 4 is required, it is clear that node 3 needs to be executed before node 4 is reached. The sequence of nodes to be executed is performed via data-flow analysis of the flags involved. The flag used at node 4 is defined at nodes 1 and 3, with node 1 assigning a false value and node 3 assigning the required true value. The required sequence is therefore $\langle 3, 4 \rangle$. Further guidance is now provided to the search in the form of the predicate of the true branch from node 2, which is required for the execution of node 3. The approach also handles nodes that should not be executed, for example if the flag was reassigned as false in a nested statement between nodes 3 and 4. This solution is not dissimilar to a static version of the chaining approach. The objective functions for executing the node sequences are not too dissimilar to *node–node*-oriented functions, which were discussed in Section 3.5.2. However, the approach has problems avoiding unrequired assignments to flags within loop bodies [51].

Aside from problems of local optima and plateaux appearing in the objective function landscape, it is entirely possible for the branch distance calculation to deceive the search. Consider the example of Figure 24. The goal is to execute the true branch of the final branching node, whose branch predicate is `r == 0`. However, unless `d` is zero, `r` will not be zero. The objective function works to guide the search away from `d` being equal to zero, since increasing values of `d` decrease values of `r` deceiving the search into believing it is getting closer and closer to zero, as depicted by the objective function landscape (Figure 25).

A further problem can occur with nested branch predicates as seen with the example of Figure 6. In this example, input data must be found satisfying `a == b` before the solution to `b == c` and


```
if (d == 0)
    r = 0;
else
    r = 1 / d;

if (r == 0)
    // target branch
```

Figure 24. Deceptive objective function example.

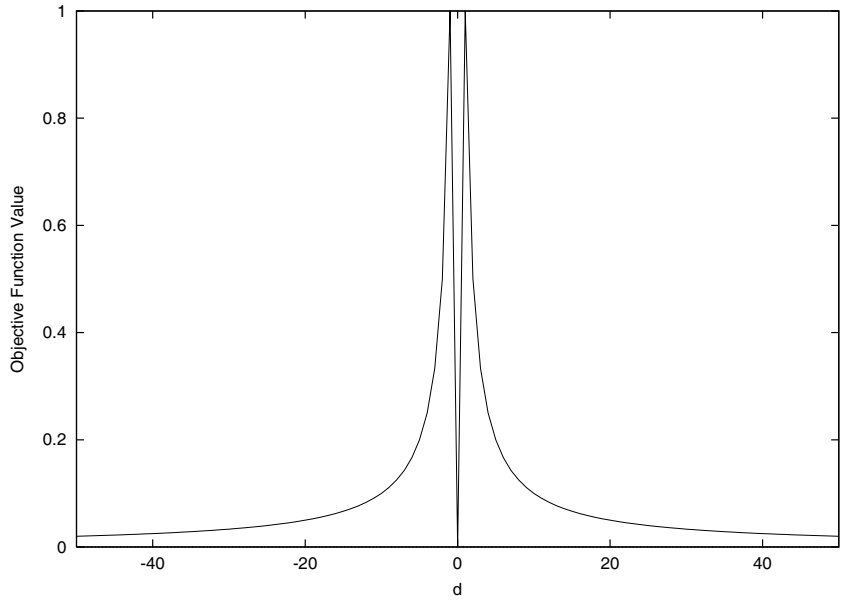


Figure 25. Landscape of the deceptive objective function.

`c < 0` can be attempted. Once input data are found for one or more of the conditions, the chances of finding input data that also fit subsequent conditions decrease. This is because a solution for subsequent conditions must be found without violating any of the earlier conditions. This leads to poor search performance. Ideally, all of the conditions should be evaluated at once. Here, none of the values `b`, `c` or `d` are modified between the branching statements, and so all predicates could be evaluated at the first branching statement. Such a situation could be established through the use of data dependency analysis [49].

A similar problem occurs with the use of short circuit evaluation of atomic conditions with branch predicates using operators such as `&&` and `||` in C. In such situations the evaluation of the overall predicate breaks off early if the end result has already been determined. Therefore, during the process of searching for test data, the individual conditions have to be attempted one after the other. For example,

```
if (a == b && b == c && c < 0)
{
    // ...
}
```

Again, it would be preferable to evaluate all of the conditions at once. In this situation, care needs to be taken when side effects appear in any of the conditions. A solution here might be to apply a side-effect removal program transformation [52,53] first. Alternatively, variable values could be saved into temporary variables inserted immediately before the branching statement, and restored after the side effect if the condition would not normally have been evaluated.

3.5.5. Applying variable dependence analysis

Harman *et al.* [54] apply variable dependence analysis to determine the subset of input variables that cannot affect the outcome at a branch predicate. In this way, the search space can be reduced, increasing the chances of finding a solution—and potentially finding it faster. Take the triangle example of Figure 4 once more. For branching node 1, only the input variables `a` and `b` are relevant. Variable `c` cannot affect the outcome at this node, and, as such, does not need to be included in the search. For branching node 5, all input variables are relevant, because `b` may have determined the outcome of `a` during the prior nodes 1–4. These ideas are similar to Korel's influences graph [31] (see Section 3.3.2), except the information is statically computed for each structural target. The variable dependence analysis information can also be used to compute a slice of the program with respect to the structural target. A program slice [55] is a smaller version of the original program which only contains the statements of interest according to some slicing criterion. In this case the criterion involves the removal of all statements that cannot affect the attainment of the desired structure. Such slices are potentially useful since they can cut down the time required to execute the program and evaluate individuals of the search.

3.5.6. Generating input sequences

A further problem for structural test data generation is caused by test objects with internal states. In these situations an input sequence is required to cover certain structures. Take the example of Figure 26. The variable `counter` is declared as `static`. This means that the value of `counter` will be retained at the end of the function call until the next time it is executed. Therefore, branching node 2 requires at least five executions of the function for the true branch to become feasible. Baresel *et al.* [56] aim to circumvent this for branch coverage by encoding individuals as sequences of input vectors. The sequence is of length n, in order for n calls to the function to be performed. Since the function is now called many times, the individual has many chances to execute the desired branch. The objective of the individual is calculated using the approach level and branch distance at the closest point of executing the branch. For the input sequence `<(6, 6, 5), (2, 2, 3), (2, 2, 3), (6, 6, 4), (6, 6, 5)>`, the individual is

```
const int THRESHOLD = 5;

int sequence_example(int a, int b, int c)
{
    static int counter = 0;
    if (((a + b) / 2) > c)      // branching node 1
        counter ++;

    if (counter >= THRESHOLD)   // branching node 2
        return 1;

    return 0;
}
```

Figure 26. Test sequence generation example.

closest to executing the true branch of the branching node in the last call, where the branch distance is $5 - 3 + K = 2 + K$. It is not required that the individual must execute the target structure during the last call to the function. One drawback to the scheme is that the tester must have some idea of how long the sequence is going to be. If the maximum sequence length is too short, the target structure will be unreachable. If it is too long, the search will take longer or fail to find test data. The use of a variable length encoding might solve this problem. Another problem is that the scheme only works for states within individual functions. In the general case, state behaviour can be exhibited by modules, abstract data types and objects. The example of Figure 27 demonstrates this. For the true branch in the function under test to be executed (part (a)), the `tick()` function in the dependent module must first be executed a number of times. An extension to the scheme of Baresel *et al.* could incorporate a function identification number and an extension to deal with different type functions, in order to generate a test script for the execution of the target structure. A further problem with state-based systems is their tendency to make use of flag and enumeration variables to control the current state. McMinn and Holcombe [57] suggest an approach combining the evolutionary search for test data with the construction of event sequences in a similar style to the chaining approach. The construction of an event sequence can be used to infer the function call sequence required, as well as solving flag problems. However, this will incur performance penalties, as an evolutionary search must take place at every step of the construction of event sequences. Another problem could potentially occur when the chaining tree of event sequences becomes too large to search exhaustively. In such cases McMinn and Holcombe suggest the use of further heuristics to pursue the exploration of the more 'promising' sequences.

3.5.7. Use of evolutionary algorithms: encodings and operators

Early work in applying genetic algorithms to structural test data generation used binary encodings. Jones *et al.* [44] found improvement in the use of a Gray code.

```
void check_time()
{
  if (get_time() == 60)
  {
    // target branch
  }
}
```

(a)

```
static time = 0;

void tick()
{
  time ++;
}

int get_time()
{
  return time;
}
```

(b)

Figure 27. Test sequence generation example with multiple functions. (a) Function under test and (b) dependent module.

However, it is common that variables will often only have valid values within a subset of the possible bit patterns at the binary level. In addition to the range imposed on an ordinal type by a compiler, input variables are often restricted to a certain range by the context of its application. One problem that can occur with binary encodings is the corruption that can occur with restricted types through the actions of the crossover and mutation operators. This problem was raised by Tracey [47]. The following shows two chromosomes (26, 81) and (56, 43) representing two integer variables restricted between 1 and 100. Crossover at locus 8 yields two offspring: (26, 107) and (56, 17).

$$\begin{array}{l|l} 00110101 & 010001 \\ 01110000 & 101011 \end{array} \quad \rightarrowtail \quad \begin{array}{l} 00110101101011 \\ 01110000010001 \end{array}$$

The final variable of the former chromosome is now out of range. One solution might be to restrict the crossover points to the boundaries of each variable, making it impossible for a variable value to go out of range. However, the chromosome can still be damaged by the mutation operator. A possible solution is to repair or penalize invalid individuals. An alternative is to use a real-valued encoding. This is the decision taken by Tracey [47] and Wegener *et al.* [48]. For real-valued encodings, crossover is naturally restricted to the boundaries of each variable. For example,

$$\begin{array}{l|l} 26 & 81 \\ 56 & 43 \end{array} \quad \rightarrowtail \quad \begin{array}{l} 26\ 43 \\ 56\ 81 \end{array}$$

The mutation operator can also be based on number creep (introduced in Section 2.3.2), taking care to ensure that each value is not shifted out of its required range. The use of a real-valued encoding also removes the need to encode and decode the input vector into and out of a binary format.

3.6. Future directions for search-based structural testing

For search-based structural testing, there are still problems involving flag and enumeration variables; unstructured control flow; and state behaviour, as have been described. Furthermore, there may be a variety of other reasons as to why test data cannot be found with ease for program structures using search-based techniques. Insights or metrics found from research in this area could be used to tune existing techniques.

Furthermore, search-based structural testing has been limited to programs of a numerical nature. Programs involving strings and dynamic data structures such as lists or trees are problematic when it is necessary to determine their required size and 'shape' for the execution of some program structure. The shape of a tree, for example, is determined by its branches and the number of nodes at each level. Some initial work by Korel in this area utilizes local search to adapt an inputted dynamic data structure so that it matches the requirements of the path to be executed [31]. It may also be necessary to find special values in special orders within these data structures, for example a string specifying a date. Some initial work in this area includes that of Baresel *et al.* [49]. Further problems with dynamic types include the comparison of pointer locations. Here, the traditional distance approach will compare memory locations; however, this information is not of real use in guiding the search to appropriate test data.

Extensions to search-based structural test data generation for object-oriented systems are complicated by problems of internal states, since objects are inherently state-based. Further issues include the use of polymorphic types. If a method is called with a reference to an object which could be of several different types, the test data generation system needs to decide which version of the interface needs to be instantiated.

Further possible areas of research include programs using information from files and sockets. Some initial work on structural testing of distributed systems includes that of Ferguson and Korel [58].

4. FUNCTIONAL (BLACK-BOX) TESTING

This section discusses the application of metaheuristic search techniques to the testing of the logical behaviour of a system, as described by some form of specification.

4.1. Generating test data from a Z specification

Jones *et al.* [59] generate test data for the triangle classification program, using a Z specification [60]. The state space of the system is described in a schema named *Triangle0*, which declares three input integer variables to represent the three sides of the triangle ($x?$, $y?$ and $z?$). This schema also describes invariants over the inputs to check that the lengths are within a specified range, and that the side lengths represent a valid triangle. These checks are also included in two other operations declared as *NumError* and *TriangleError*. Four further operations decide if the triangle is scalene (*ScalTri*), equilateral (*EquiTri*), isosceles (*IsosTri*) or right-angled (*RightTri*).

Using these schema, the whole system can be declared as

$$\begin{aligned} Triangle ::= &(Triangle0 \wedge EquiTri) \vee (Triangle0 \wedge IsosTri) \vee \\ &(Triangle0 \wedge ScalTri) \vee (Triangle0 \wedge RightTri) \vee \\ &NumError \vee TriangleError \end{aligned}$$

```
int wrapping_counter(int n)
{
    int r;
    if (n >= 10)
        r = 0;
    else
        r = n + 1;

    return r;
}
```

Figure 28. Wrapping counter example.

For the purposes of test data generation, each disjunct is considered as a *route* through the system. Genetic algorithms are used to search for test data that satisfy each route.

The fitness function rewards individuals that come close to satisfying the conjuncts in each route. In the case of an equilateral triangle, the predicates to be satisfied include invariants from the state space schema conjuncted with those of the *EquiTri* schema $((x? = y?) \wedge (y? = z?))$. Each conjunct is evaluated using a distance-based approach, in a similar fashion to the branch distance calculations used in structural testing. The overall fitness of the route is the summation of the distances for each of its conjuncts.

The results report successful test data generation by the genetic algorithm for each of the routes under examination, namely *ScalTri*, *EquiTri*, *IsosTri* and *RightTri*. However, the example is small and not general enough to establish its usefulness. Furthermore, only a small subset of Z is used, and this is limited to the use of relational operators only.

4.2. Testing specification conformance

The last section showed how test data could be generated from a formal specification. The work of Tracey *et al.* [47,61] extends this idea. In their technique, the conformance of the implementation to its specification is checked by executing the test object with the generated test data, and then validating the output against the specification.

The specification of the implementation is represented as a pre-condition, which defines valid inputs, and a post-condition, which defines the output. A failure is found when an input situation is discovered that satisfies the pre-condition of the function, but for which the outputs violate the post-condition. An objective function is derived which describes the 'closeness' of the test data to uncovering such a situation, and metaheuristic search techniques are then employed to seek failures in the implementation.

As a simple example, take the wrapping counter function of Figure 28, originally presented by Tracey [47]. This function implements a counter, which takes an integer value between 0 and 10, and returns the increment. If the input is 10, the counter wraps round to 0. The pre-condition for this function is simply

$$n \geq 0 \wedge n \leq 10$$

The post-condition is

$$(n < 10 \rightarrow r = n + 1) \vee (n = 10 \rightarrow r = 0)$$

where n is the input value and r is the return value.

A constraint system is then derived to describe conditions of implementation non-conformance by taking the pre-condition in conjunction with the negated post-condition:

$$n \geq 0 \wedge n \leq 10 \wedge \neg((n < 10 \rightarrow r = n + 1) \vee (n = 10 \rightarrow r = 0)) \quad (1)$$

An objective function is derived to indicate how 'close' failure is. This is constructed from the above constraint system using the rules in Tables II and III:

$$obj(n \geq 0) + obj(n \leq 10) + min((obj(n < 10) + obj(r \neq n + 1)), (obj(n = 10) + obj(r \neq 0))) \quad (2)$$

It was found that the landscapes of the objective functions derived from such constraint systems contained areas of plateaux. Figure 29 shows the objective function landscape for a faulty version of the program, where the branch predicate `n >= 10` is replaced by `n > 10`. The objective function is zero when $n = 10$, indicating a fault. However, a plateau forms for values of n between 0 and 9. This results from the use of the min operator in the objective function. For $n < 10$, the objective value of the first operand, $obj(n < 10) + obj(r \neq n + 1)$, is always K, which is always smaller than the objective value of the second operand $obj(n = 10) + obj(r \neq 0)$. It was found that guidance to the search could be improved by converting the constraint system to disjunctive normal form, and then using each disjunct as the basis of a separate search.

Conversion of the original constraint system (Equation (1)) to disjunctive normal form gives two disjuncts:

$$\begin{aligned} &\text{Disjunct 1:} \quad n \geq 0 \wedge n \leq 10 \wedge n < 10 \wedge r \neq n + 1 \\ &\text{Disjunct 2:} \quad n \geq 0 \wedge n \leq 10 \wedge n = 10 \wedge r \neq 0 \end{aligned}$$

The objective functions for each disjunct, are, respectively

$$\begin{aligned} &\text{Disjunct 1:} \quad obj(n \geq 0) + obj(n \leq 10) + obj(n < 10) + obj(r \neq n + 1) \\ &\text{Disjunct 2:} \quad obj(n \geq 0) + obj(n \leq 10) + obj(n = 10) + obj(r \neq 0) \end{aligned}$$

Figure 30 shows the landscape for the faulty branch predicate `n >= 10` for the objective functions of disjuncts 1 and 2, respectively. As can be seen, the landscape for the second disjunct in the range $0 \leq n < 10$ gives more guidance to the failure point when the objective value is zero.

Tracey [47] applied this technique to the testing of a safety-critical nuclear primary protection system, written in Pascal. Two sub-systems were available for this evaluation. The first consisted of 36 pages of formal VDM-SL specification and the second 54 pages, with approximately 2000 lines of executable code. The pre- and post-conditions for each function of each sub-system were manually derived from the specification, with 733 different disjuncts obtained. A mutation testing tool was then used to generate mutant implementations of the code. Simulated annealing and genetic algorithms were then used as metaheuristic searches for the technique. Both searches killed 100% of approximately 170 non-equivalent mutants, outperforming hill climbing and random searches, which still achieved overall scores of over 90%.

Buehler and Wegener [40] use evolutionary algorithms to test specification conformance of an early version of an automated vehicle parking system. This system aims to automate parking of

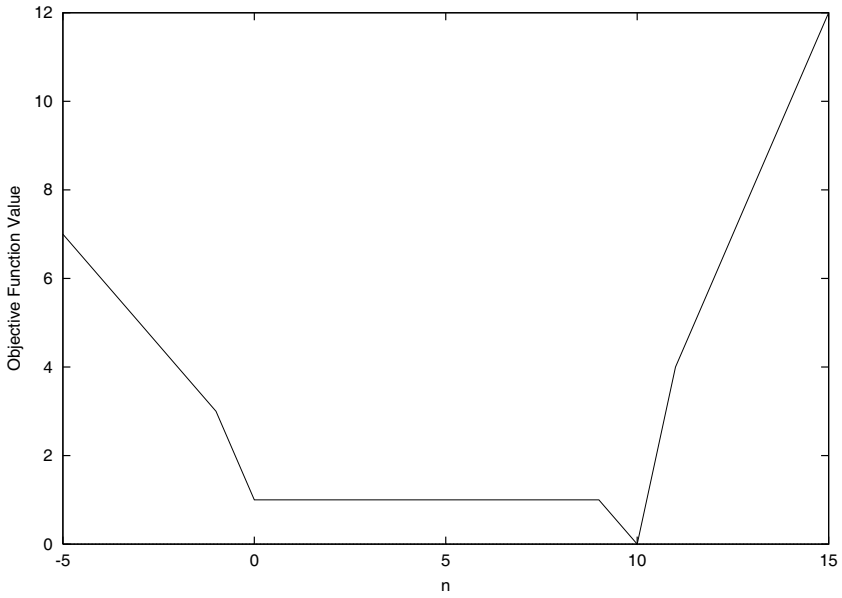
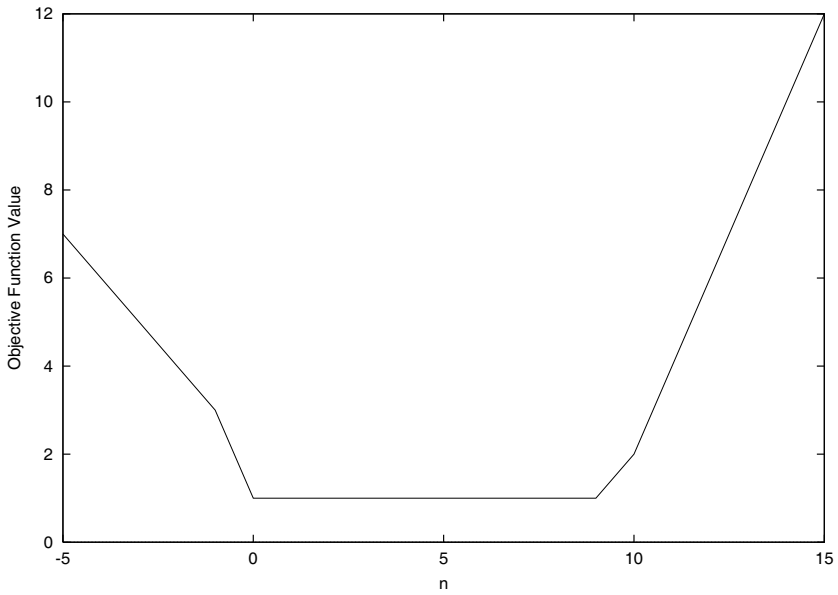


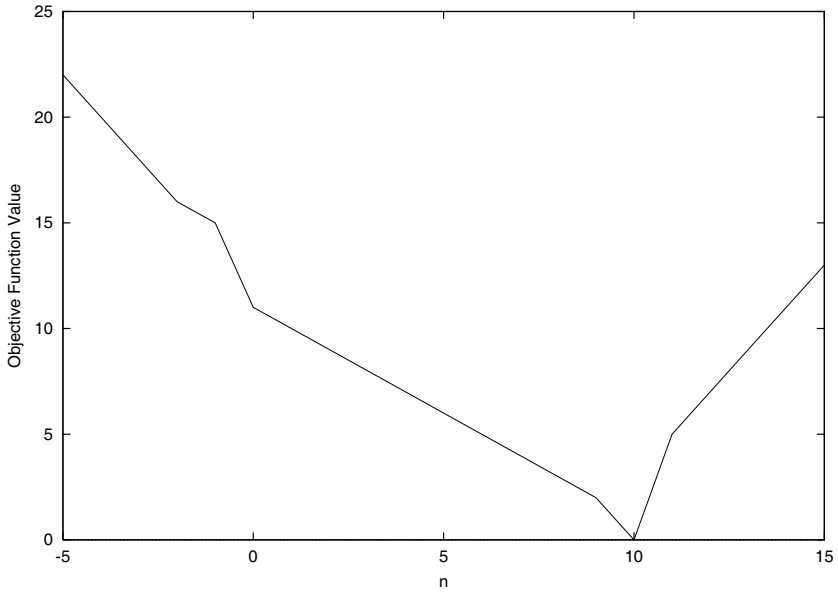
Figure 29. Objective function landscape for wrapping counter example, where $K = 1$.

a vehicle lengthways into a parking space, using information from environmental sensors, which register surrounding objects. The individuals of the search are simply parking scenarios which describe the dimensions of a parking space, including collision areas, and the starting position of the car. The parking control unit is called with these data, and a parking manoeuvre is simulated. With a successful test being one which causes a collision, the objective function is simply the value of the smallest distance between the car and the collision area recorded during the simulation. In the experiment undertaken, approximately 900 scenarios were simulated, with more than 25 scenarios found leading to collisions. After analysis of these scenarios, it was discovered that the controller had difficulties with scenarios where the parking space was some distance away and the starting position was already near to the collision area on one side. A fault was also detected with the simulation environment, where it was found that calculations involving the position of the car were too imprecise. This led to further simulated impacts with the collision area.

Baresel *et al.* [56] test Simulink and Stateflow models which require input signal sequences to be generated. One problem in this domain is the generation of realistic signals, and their potential length, which could result in a very large search space. Baresel *et al.* propose a novel solution by building the overall signal from a series of simple signal types, for example sine, spline and linear curves. The search space then becomes the set of parameters used to construct a signal section built from a base signal, for example its amplitude and length. This guarantees the generation of realistic input signals, as well as keeping the size of the search space relatively compact. The Distronic cruise control system was tested using this technique. This system senses the approach to slower vehicles and automatically slows the car down to maintain a safe following distance. The objective function checks for violations of the



(a)



(b)

Figure 30. Objective function landscapes for individual disjuncts of the wrapping counter example, where $K = 1$. (a) Disjunct 1 and (b) disjunct 2.

requirements, by checking dependencies between output signals, checking for output signal boundary violations and checking signal maximal overshoot and settlement time. For Distronic, tests revealed that the system broke a maximal speed violation under certain input conditions.

4.3. Future directions for search-based functional testing

There has been less activity in the area of search-based functional testing compared with structural testing. Functional tests can be derived from different forms of specification. For tests derived in this way, a present barrier to complete automation is the fact that a mapping needs to be provided from the abstract model of the specification to the concrete form of the implementation. For system tests, a potential problem is the size of the search space. A possible solution is the use of innovative encodings, such as the afore-mentioned scheme of Baresel *et al.* [56] for generating input signals from base signal types. Further potential problems include the existence of internal states. Test sequences may need to be generated to put the system into some valid state in order for the property of interest to be tested. Thus, the search may need additional information regarding the state structure of the system.

5. GREY-BOX TESTING

Grey-box testing combines both structural and functional information for the purposes of testing.

5.1. Assertion testing

The work of Korel and Al-Yami [62] attempts to find test cases that violate assertion conditions, which can be embedded by the programmer into the program code. Assertions specify constraints that apply to some state of a computation. When an assertion evaluates to false, an error has been found in the program. Assertions can be embedded within comment regions, either as Boolean conditions, for example

```
/*@ i > 0 and i <= 10 @*/  // assertion
i ++;                      // program statement
```

or as executable code. When assertions are embedded as blocks of executable code, a special variable `assert` is used. This is assigned true or false values to denote the correct or incorrect state of the assertion. For example, the following assertion checks that the elements of an array are sorted in ascending order:

```
/*@
assert = true;
for (i = 0; i < len-1; i++)
{
    if (a[i] > a[i+1])
        assert = false;
}
@*/
// ... normal program code ...
```


Korel and Al-Yami showed how the search for test data to falsify an assertion reduced to the problem of executing a specific statement in the program. First, assertions are stripped out of the code. For Boolean conditions, code is generated and placed in the assertion's original position. The assertion condition is then negated. This new condition is the condition which represents a violation and, therefore, the finding of a fault. This is then converted to disjunctive normal form. A series of nested `if` statements are then generated for each condition within each individual disjunct. If each `if` statement is evaluated as true, the violation is reported. For example, take the assertion condition $(a < b \wedge \neg(b = c \wedge c = d))$. The negated form of the assertion is $(a \geq b \vee (b = c \wedge c = d))$. The following code is generated for this negated condition (which is already in disjunctive normal form):

```
if (a >= b)
    report_violation();
if (b == c)
    if (c == d)
        report_violation();
```

The goal of the search is then to execute one of the `report_violation()` statements.

For assertions appearing as code, the assertion code is formed into a function, with the original assertion comment region replaced with a call to that function. The goal is then to execute a false assignment to the `assert` variable statement within the function, and thereafter avoiding all true assignments to the variable.

The process of test data generation is performed using the chaining approach (Section 3.3.4). In addition to programmer-embedded assertions, Korel and Al-Yami's tool automatically generates assertions for runtime errors such as division by zero errors, array boundary violations and overflow errors. The tool also tries to find input data to stimulate error conditions where variables are uninitialized, yet used in some following program statement.

In initial experiments, nine original Pascal programs were embedded with assertions. Twenty-five faulty versions were then produced. With these experiments, it was found that inputs could be found to violate an assertion—and thereby reveal a fault—92% of the time.

5.2. Exception condition testing

Tracey *et al.* [47,63] built on the ideas of Korel and Al-Yami using genetic algorithms and simulated annealing to generate input data to test the handling of runtime error conditions in code. In many languages, such as C++, Java and Ada, these runtime errors are known as *exceptions*. These languages provide explicit exception-handling constructs so that exception-related code can be separated from the main logic of the program. Tracey *et al.* generate test data for the raising of the exception, and then for the structural coverage of the exception handler. As with the work of Korel, both problems reduce to the problem of the execution of a certain statement (i.e. the statement which triggers the exception via a `throw` or `raise` statement), or a sequence of statements through the code (the raising of the exception followed by coverage of some structural element within the exception handler). Experiments were undertaken with seven simple programs of no more than 200 lines of code. It was found that metaheuristic techniques could generate test data to raise almost all the exception conditions contained within the code, and full branch coverage of exception handlers where they existed. An industrial

experiment was also undertaken on an engine controller. Here, test data were generated which raised a variety of exception conditions. However, it was found that these exceptions could not be raised in practice, since input situations had been generated which were not possible during actual operation of the system.

5.3. Future directions for search-based grey-box testing

The ability to embed arbitrary assertions within programs and be able to search for test data in order to check their violation is a very powerful concept. Future applications of assertion-based testing have been suggested by Tracey [36,47]. One idea includes component-reuse testing. This amounts to searching for test data that cause the component to be called where its usage assumptions (as described by assertions) are broken. Another application is the checking of outputs from structural tests. Presently, these have to be checked manually. Black-box assertions could be used as an oracle for the tests, offering further automation to the overall process.

6. NON-FUNCTIONAL TESTING

To date, search-based testing effort in the area of non-functional testing has concentrated on checking the best-case and worst-case execution times of real-time systems.

6.1. Execution time testing

The correct operation of a real-time system not only depends on its logical behaviour, but also its timing behaviour. In general, incorrect timing behaviour of a real-time system occurs when outputs are produced too early or too late. Execution time testing, therefore, involves attempting to find the worst-case execution time (WCET) or the best-case execution time (BCET) of a system in order to determine whether it is compliant with its timing constraints. This task is extremely difficult to achieve, since the timing behaviour of a piece of software is not only dependent on its internal structure, but also the characteristics of the target hardware. At the software level, timing is dependent on the instructions used and their corresponding data items. The compiler can also introduce effects not apparent at source code level. At the hardware level, accounting for the actions of the target processor is extremely difficult when caching and pipelining operations need to be considered. As a consequence, the longest or shortest path through the program will not necessarily yield the longest or shortest execution time.

6.1.1. Static analysis

Static analysis can be used to derive upper and lower bounds on WCET and BCET, respectively, in order to try and ensure that timing schedules will be met. This is performed by examining the possible execution paths and then modelling timing behaviour at the hardware level. The primary step needs assistance from the programmer, since information is required regarding infeasible paths, and the maximum number of iterations for each loop appearing in the code. Unfortunately, the possibility of simulation errors and the need for human involvement make this an error-prone process [64,65]. The result produced can also be extremely pessimistic in the case of WCET and optimistic in the case

of BCET. Sometimes the estimates can vary from those observed in practice by a magnitude of ten times[‡].

Consequently, the calculations produced still need to be tested. Of course, tests derived to expose flaws in the logical behaviour are generally of little benefit in this domain.

6.1.2. *Search-based execution time testing*

Search-based execution time testing seeks input situations which invoke extreme execution times. The objective function is simply the execution time of the system as executed with some input. The search attempts to maximize the objective function in the case of WCET, and minimize it in the case of BCET. If a test case is found that violates the timing constraints, the search can be terminated.

Wegener *et al.* [66] were the first to apply genetic algorithms to temporal testing. In their experiments [67,68] it is shown that genetic algorithms yield better results than random testing. A number of experiments with industrial test objects were carried out. A further experiment investigated six time-critical tasks in an engine control system [69]. Genetic algorithms were again found to outperform random search, and also tests constructed by the developers themselves. The developers' tests never found the longest execution times, and in three cases the developers' tests were worse than the random tests. Since the developers had internal knowledge of the system, these results were met with some surprise. Wegener *et al.* suggest this may be down to the use of system calls, linkage and compiler optimization, the effects of which on temporal behaviour could only be guessed with difficulty by the developers. Additional work by O'Sullivan *et al.* [70] applies cluster analysis to determine when the search should be terminated. This technique decides if the search is converging on the basis of the distribution of individuals in the search space.

Puschner and Nossal [64] apply genetic algorithms to find WCET for seven programs with differing execution-time behaviour. The results are compared with those obtained by random search, upper WCET bounds found by static analysis, as well as 'best effort' times, which were the researchers' own efforts to find input data to yield the WCET. The genetic algorithm was found to match or find longer times than the random search. The superiority of the genetic algorithm was particularly evident in large input domains. The genetic algorithm found similar times to the best effort analysis, in one case finding a longer time. Whilst upper bound times found by static analysis were never broken, they were matched on several occasions. In practice, this is unusual since the times provided by static analysis are generally too pessimistic or too optimistic for WCET and BCET, respectively.

Tracey employs simulated annealing and genetic algorithms for finding the WCET of a handful of small, well-understood programs written in Ada, with known WCET behaviour [37,47]. Each experiment was deemed to be a success if the technique executed the path through the program which yielded the already known WCET. It was found that genetic algorithms were more successful than simulated annealing, both of which outperformed hill climbing and random search. Overall, the genetic algorithm achieved success in fewer trials than simulated annealing. In this particular study, it was found that varying the parameters of the optimization techniques had little effect on the end result,

[‡]J. Wegener, private communication, 2003.

apart from when the initial temperature was set too low for simulated annealing, where dependency on the starting position could not be lost.

Unfortunately, if a branch in the program is executed only with a low probability, the chance of a search technique executing it is low. If this branch is involved in a path leading to an extreme execution time, then the extreme execution time will not be found. Gross [71] identifies a number of properties of programs which lead to low probability branches, for example high levels of nesting, branches that are only executed if an input variable is a specific value, and so on. However, just because these features exist in some source code, it does not necessarily mean that an extreme execution time will not be found. Therefore, Gross conducted an empirical study based on a handful of test objects to establish a system which could predict the testability of test objects, based on their source code. However, the empirical study was very small, consisting of only 15 test objects. The type of test objects used was not characterized in any particular way, and the effects of the underlying hardware were not accounted for. Furthermore, the dependence of the prediction system on the setting of the genetic algorithm parameters was not established.

Wegener *et al.* [65] investigated the objective function landscape for timing behaviour. They found that due to the fact that the execution times for several input vectors executing the same program path can be identical, plateaux are common in the landscape. Discontinuities were also formed by significant differences in execution time for slightly different input vectors leading to the execution of different paths. These findings help explain why little improvement could be obtained by using local search to improve times found by genetic algorithms in the work of Wegener *et al.* [67] and Tracey [47].

The experiments performed show the superiority of search-based approaches over random testing. Whilst search-based techniques cannot guarantee that the actual WCET or BCET will be found, the best result obtained can be used to form an interval with the time obtained from static analysis within which the *actual* extreme execution time most probably lies.

6.1.3. Future directions for search-based execution time testing

The use of search techniques has been shown to bind execution times from opposite ends to static analysis; however, it is interesting that no work has been published which attempts to combine the relative benefits of both, in order to yield tighter bounds. For example, there is the potential to incorporate domain knowledge from static analysis. It is surprising that no cues from either the source code, machine code or details regarding the hardware to be used have been included for the benefit of the search. Gross also suggests the input of human knowledge [72]. Conversely, search-based techniques could be used to verify path feasibility for static analysis.

Other strategies, such as guaranteeing survival of a path for a number of generations, have been suggested in the literature [68]. However, to the author's knowledge no work has been published reporting the effectiveness of these ideas. In particular, search strategies could be adapted to give 'low probability' paths special treatment, by directing the search into these untried areas (for example by using branch distance calculations) and then ensuring that the genetic algorithm maintains a certain level of diversity so that the proportion of the population utilizing these paths is not instantly 'killed' off. Another possibility is to combine the objective function with those used by structural test data generation to ensure that timing behaviour involving all branches is explored. In such instances care needs to be taken to avoid probe effects—deviations in the actual runtime behaviour due to the effects of instrumentation.

Finally, the work on search-based execution time testing has so far been largely restricted to programs of a procedural nature. However, some work extending these techniques to object-oriented software is beginning to appear [73].

6.2. Future directions for search-based non-functional testing

Work in non-functional testing has been largely restricted to execution time testing. However, there are many more possibilities for automating non-functional tests with search-based approaches. For example, the resource usage of software could be tested by searching for input situations that cause constraints on memory or storage requirements to be broken. In a similar fashion, a search-based approach to the automatic detection of memory leaks may also be possible. Other possibilities for search-based automation include stress testing, security testing and so on.

7. CONCLUSIONS

This paper has surveyed the application of metaheuristic search techniques to software test data generation. Search-based software test data generation is just one example of search-based software engineering.

For structural test data generation, metaheuristic dynamic approaches were compared against static techniques based on symbolic execution. Techniques using symbolic execution evaluate program code in order to build up a system of constraints describing the test goal. However, this is problematic in the presence of loops and in cases where computed storage locations need to be determined. Instead of trying to formulate a constraint system, dynamic approaches merely execute the program with some input, and examine the effects via some form of program instrumentation. This helps circumvent some problems associated with static techniques, since dynamic information—for example pointer locations—are known at runtime. Metaheuristic techniques are then used to search for test data. The use of a metaheuristic technique requires the definition of an objective function which 'rewards' test data solutions on the basis of how close they were to fulfilling the required test goal. *Coverage-oriented* objective functions reward input data on the basis of the number of program structures executed. It was argued, however, that *structure-oriented* approaches represent a more successful strategy. This is because each individual uncovered structure receives specific attention in the form of an individual search. Each individual search is provided explicit guidance to the coverage of the structure in question by an automatically tailored objective function. Without this guidance, nested structures only executed under special circumstances are unlikely to be exercised.

Search-based test data generation approaches to functional testing have largely focused on seeking input situations which demonstrate that an implementation does not conform to its specification. Executions of the test object are monitored, with input data solutions rewarded on the basis of how close they are to discovering a failure, as decided using the specification.

Grey-box test data generation approaches combine methods used in generating structural and functional testing. It was shown, in the work of Korel and Al-Yami, how the violation of a program-embedded assertion reduces to the problem of executing a program statement. Therefore, structure-oriented white-box testing techniques can be used to attempt to induce violations of these assertions.

The paper has discussed the results obtained in each of the testing areas, with many successful experiments undertaken using real-world examples drawn from industry. However, there are still many problems that need to be solved in each area, and directions for future research have been outlined at the end of each section.

ACKNOWLEDGEMENTS

This work is sponsored by DaimlerChrysler Research and Technology. The author would like to thank Mike Holcombe, Joachim Wegener, André Baresel and various anonymous referees for their comments on earlier drafts and sections of this paper. The deceptive function example in Section 3.5.4 is due to Mark Harman, and was presented at the *Search-based Software Engineering Workshop* in Windsor, September 2002.

REFERENCES

1. Harman M, Jones B. Search-based software engineering. *Information and Software Technology* 2001; **43**(14):833–839.
2. Clark J, Dolado JJ, Harman M, Hierons R, Jones B, Lumkin M, Mitchell B, Mancoridis S, Rees K, Roper M, Shepperd M. Reformulating software engineering as a search problem. *IEE Proceedings—Software* 2003; **150**(3):161–175.
3. Reeves CR (ed.). *Modern Heuristic Techniques for Combinatorial Problems*. McGraw-Hill: New York, 1995.
4. Corne D, Dorigo M, Glover F (eds.). *New Ideas in Optimization*. McGraw-Hill: New York, 1999.
5. Metropolis N, Rosenbluth A, Rosenbluth M, Teller A, Teller E. Equation of state calculations by fast computing machines. *Journal of Chemical Physics* 1953; **21**(6):1087–1092.
6. Kirkpatrick S, Gellat CD, Vecchi MP. Optimization by simulated annealing. *Science* 1983; **220**(4598):671–680.
7. Bäck T, Hoffmeister F, Schwefel H. A survey of evolution strategies. *Proceedings of the 4th International Conference on Genetic Algorithms*, San Diego, CA, 1991, Booker L, Belew R (eds.). Morgan Kaufmann: San Francisco, CA, 1991; 2–9.
8. Bäck T. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press: New York, 1996.
9. Whitley D. An overview of evolutionary algorithms: Practical issues and common pitfalls. *Information and Software Technology* 2001; **43**(14):817–831.
10. Holland JH. *Adaptation in Natural and Artificial Systems*. University of Michigan Press: Ann Arbor, MI, 1975.
11. Whitley D. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. *Proceedings of the 3rd International Conference on Genetic Algorithms*, San Mateo, CA, 1989, Schaffer JD (ed.). Morgan Kaufmann: San Francisco, CA, 1989; 116–121.
12. Deb K, Goldberg D. A comparative analysis of selection schemes used in genetic algorithms. *Foundations of Genetic Algorithms*, Rawlins GJ (ed.). Morgan Kaufmann: San Francisco, CA, 1991; 69–93.
13. Whitley D. A free lunch proof for gray versus binary encodings. *Proceedings of the Genetic and Evolutionary Computation Conference*, Orlando, FL, 1999. Morgan Kaufmann: San Francisco, CA, 1999; 726–733.
14. Whitley D, Rana SB, Dzubera J, Mathias KE. Evaluating evolutionary algorithms. *Artificial Intelligence* 1996; **85**(1–2):245–276.
15. Goldberg D. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley: Boston, MA, 1989.
16. Antonisse J. A new interpretation of schema notation that overturns the binary encoding constraint. *Proceedings of the 3rd International Conference on Genetic Algorithms and Their Applications*, San Mateo, CA, 1989. Morgan Kaufmann: San Francisco, CA, 1989; 86–91.
17. Davis L. *Handbook of Genetic Algorithms*. International Thomson Computer Press: London, 1996.
18. Mitchell M. *An Introduction to Genetic Algorithms*. MIT Press: Cambridge, MA, 1996.
19. Srinivas M, Patnaik LM. Genetic algorithms: A survey. *IEEE Computer* 1994; **27**(6):17–26.
20. Whitley D. A genetic algorithm tutorial. *Statistics and Computing* 1994; **4**:65–85.
21. Ferrante J, Ottenstein K, Warren JD. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems* 1987; **9**(3):319–349.
22. King J. A new approach to program testing. *Proceedings of the International Conference on Reliable Software*, Los Angeles, CA, 1975. ACM Press: New York, 1975; 228–233.
23. King J. Symbolic execution and program testing. *Communications of the ACM* 1976; **19**(7):385–394.
24. Clarke L. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering* 1976; **2**(3):215–222.
25. Boyer RS, Elspas B, Levitt KN. SELECT—A formal system for testing and debugging programs by symbolic execution. *Proceedings of the International Conference on Reliable Software*, Los Angeles, CA, 1975. ACM Press: New York, 1975; 234–244.

26. Ramamoorthy CV, Ho SF, Chen WT. On the automated generation of program test data. *IEEE Transactions on Software Engineering* 1976; **2**(4):293–300.
27. Garey MR, Johnson DS. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman: New York, 1979.
28. DeMillo RA, Offutt AJ. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering* 1991; **17**(9):900–909.
29. Offutt AJ, Jin Z, Pan J. The dynamic domain reduction procedure for test data generation. *Software—Practice and Experience* 1999; **29**(2):167–193.
30. Miller W, Spooner D. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering* 1976; **2**(3):223–226.
31. Korel B. Automated software test data generation. *IEEE Transactions on Software Engineering* 1990; **16**(8):870–879.
32. Gallagher MJ, Narasimhan VL. ADTEST: A test data generation suite for Ada software systems. *IEEE Transactions on Software Engineering* 1997; **23**(8):473–484.
33. Korel B. Dynamic method for software test data generation. *Software Testing, Verification and Reliability* 1992; **2**(4):203–213.
34. Korel B. Automated test generation for programs with procedures. *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA 1996)*, San Diego, CA, 1996. ACM Press: New York, 1996; 209–215.
35. Ferguson R, Korel B. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology* 1996; **5**(1):63–86.
36. Tracey N, Clark J, Mander K, McDermid J. An automated framework for structural test-data generation. *Proceedings of the International Conference on Automated Software Engineering*, Hawaii, 1998. IEEE Computer Society Press, 1998; 285–288.
37. Tracey N, Clark J, Mander K. The way forward for unifying dynamic test-case generation: The optimisation-based approach. *Proceedings of the International Workshop on Dependable Computing and its Applications*, 1998. Department of Computer Science, University of Witwatersrand: Johannesburg, South Africa, 1998; 169–180.
38. Wegener J, Buhr K, Pohlheim H. Automatic test data generation for structural testing of embedded software systems by evolutionary testing. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, New York, 2002. Morgan Kaufmann: San Francisco, CA, 2002; 1233–1240.
39. Harman M, Hu L, Hierons R, Baresel A, Sthamer H. Improving evolutionary testing by flag removal. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, New York, 2002. Morgan Kaufmann: San Francisco, CA, 2002; 1359–1366.
40. Buehler O, Wegener J. Evolutionary functional testing of an automated parking system. *International Conference on Computer, Communication and Control Technologies and the 9th International Conference on Information Systems Analysis and Synthesis*, Orlando, FL, 2003.
41. Xanthakis S, Ellis C, Skourlas C, Le Gall A, Katsikas S, Karapoulios K. Application of genetic algorithms to software testing (Application des algorithmes génétiques au test des logiciels). *Proceedings of the 5th International Conference on Software Engineering and its Applications*, Toulouse, France, 1992; 625–636.
42. Roper M. Computer aided software testing using genetic algorithms. *Proceedings of the 10th International Software Quality Week*, San Francisco, CA, 1997.
43. Watkins A. The automatic generation of test data using genetic algorithms. *Proceedings of the Fourth Software Quality Conference*, 1995; 300–309.
44. Jones B, Sthamer H, Eyres D. Automatic structural testing using genetic algorithms. *Software Engineering Journal* 1996; **11**(5):299–306.
45. McGraw G, Michael C, Schatz M. Generating software test data by evolution. *IEEE Transactions on Software Engineering* 2001; **27**(12):1085–1110.
46. Pargas R, Harrold M, Peck R. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability* 1999; **9**(4):263–282.
47. Tracey N. A search-based automated test-data generation framework for safety critical software. *PhD Thesis*, University of York, 2000.
48. Wegener J, Baresel A, Sthamer H. Evolutionary test environment for automatic structural testing. *Information and Software Technology* 2001; **43**(14):841–854.
49. Baresel A, Sthamer H, Schmidt M. Fitness function design to improve evolutionary structural testing. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, New York, 2002. Morgan Kaufmann: San Francisco, CA, 2002; 1329–1336.
50. Bottaci L. Instrumenting programs with flag variables for test data search by genetic algorithm. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, New York, 2002. Morgan Kaufmann: San Francisco, CA, 2002; 1337–1342.

51. Baresel A, Sthamer H. Evolutionary testing of flag conditions. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003)*, Chicago, IL, 2003 (*Lecture Notes in Computer Science*, vol. 2724). Springer: Berlin, 2003; 2442–2454.
52. Harman M, Hu L, Zhang X, Munro M. Side-effect removal transformation. *Proceedings of the 9th IEEE International Workshop on Program Comprehension (IWPC 2001)*, Toronto, Canada, 2001. IEEE Computer Society Press: Los Alamitos, CA, 2001, 310–319.
53. Harman M, Hu L, Zhang X, Munro M, Dolado JJ, Otero MC, Wegener J. A post-placement side-effect removal algorithm. *Proceedings of the 18th IEEE International Conference on Software Maintenance (ICSM 2002)*, Montreal, Canada, 2002. IEEE Computer Society Press: Los Alamitos, CA, 2002, 2–11.
54. Harman M, Fox C, Hierons R, Hu L, Danicic S, Wegener J. VADA: A transformation-based system for variable dependence analysis. *Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, Montreal, Canada, 2002. IEEE Computer Society Press: Los Alamitos, CA, 2002, 55–64.
55. Weiser M. Program slicing. *IEEE Transactions on Software Engineering* 1984; **10**(4):352–357.
56. Baresel A, Pohlheim H, Sadeghipour S. Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003)*, Chicago, IL, 2003 (*Lecture Notes in Computer Science*, vol. 2724). Springer: Berlin, 2003; 2428–2441.
57. McMinn P, Holcombe M. The state problem for evolutionary testing. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003)*, Chicago, IL, 2003 (*Lecture Notes in Computer Science*, vol. 2724). Springer: Berlin, 2003; 2488–2497.
58. Ferguson R, Korel B. Generating test data for distributed software using the chaining approach. *Information and Software Technology* 1996; **38**(5):343–353.
59. Jones B, Sthamer H, Yang X, Eyres D. The automatic generation of software test data sets using adaptive search techniques. *Proceedings of the 3rd International Conference on Software Quality Management*, Seville, Spain, 1995; 435–444.
60. Spivey JM. *The Z notation: A Reference Manual* (2nd edn) (*International Series in Computer Science*). Prentice-Hall: Englewood Cliffs, NJ, 1992.
61. Tracey N, Clark J, Mander K. Automated program flaw finding using simulated annealing. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 1998)*, Clearwater Beach, FL, 1998. *Software Engineering Notes* 1998; **23**(2):73–81.
62. Korel B, Al-Yami AM. Assertion-oriented automated test data generation. *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, Berlin, Germany, 1996. IEEE Computer Society Press: Los Alamitos, CA, 1996, 71–80.
63. Tracey N, Clark J, Mander K, McDermid J. Automated test data generation for exception conditions. *Software—Practice and Experience* 2000; **30**(1):61–79.
64. Puschner P, Nossal R. Testing the results of static worst-case execution-time analysis. *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, 1998. IEEE Computer Society Press, 1998; 134–143.
65. Wegener J, Pohlheim H, Sthamer H. Testing the temporal behavior of real-time tasks using extended evolutionary algorithms. *Proceedings of the 7th European Conference on Software Testing, Analysis and Review (EuroSTAR 1999)*, Barcelona, Spain, 1999.
66. Wegener J, Grimm K, Grochtmann M, Sthamer H, Jones B. Systematic testing of real-time systems. *Proceedings of the 4th European Conference on Software Testing, Analysis and Review (EuroSTAR 1996)*, Amsterdam, The Netherlands, 1996.
67. Wegener J, Sthamer H, Jones BF, Eyres DE. Testing real-time systems using genetic algorithms. *Software Quality Journal* 1997; **6**(2):127–135.
68. Wegener J, Grochtmann M. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems* 1998; **15**(3):275–298.
69. Wegener J, Pitschinetz R, Sthamer H. Automated testing of real-time tasks. *Proceedings of the First International Workshop on Automated Program Analysis, Testing and Verification*, Limerick, Ireland, 2000.
70. O'Sullivan M, Vössner S, Wegener J. Testing temporal correctness of real-time systems—a new approach using genetic algorithms and cluster analysis. *Proceedings of the 6th European Conference on Software Testing, Analysis and Review (EuroSTAR 1998)*, Munich, Germany, 1998.
71. Gross H-G. A prediction system for evolutionary testability applied to dynamic execution time analysis. *Information and Software Technology* 2001; **43**(14):855–862.
72. Gross H-G. An evaluation of dynamic, optimisation-based worst-case execution time analysis. *Proceedings of the International Conference on Information Technology: Prospects and Challenges in the 21st Century*, Kathmandu, Nepal, 2003.
73. Gross H-G. Evolutionary testing in component-based real-time system construction. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002) Late Breaking Papers*, New York, 2002; 207–214.

