

An Automated Framework for Structural Test-Data Generation

Nigel Tracey John Clark Keith Mander John McDermid

Department of Computer Science.

University of York,

Heslington, York.

YO1 5DD, England.

+44 1904 432749

{njt, jac, mander, jam}@cs.york.ac.uk

Abstract

Structural testing criteria are mandated in many software development standards and guidelines. The process of generating test-data to achieve 100% coverage of a given structural coverage metric is labour intensive and expensive. This paper presents an approach to automate the generation of such test-data. The test-data generation is based on the application of a dynamic optimisation-based search for the required test-data. The same approach can be generalised to solve other test-data generation problems. Three such applications are discussed – boundary value analysis, assertion/run-time exception testing and component re-use testing. A prototype tool-set has been developed to facilitate the automatic generation of test-data for these structural testing problems. The results of preliminary experiments using this technique and the prototype tool-set are presented and show the efficiency and effectiveness of this approach.

1 Introduction

Software testing is an expensive and time-consuming process, especially in safety-critical applications, typically consuming at least 50% of the total costs involved in developing software [3]. However, it is an imperative task as it remains the primary method through which confidence in software is gained. Automation of the testing process would allow both reduced development costs and an increase in the quality of (or at least confidence in) the software. Automation of the testing process – the maintenance and execution of tests – is well in advance of automation of test-case and test-data generation. Ould suggested that automation of test-data generation is vital to advance the state-of-the-art in software testing [25]. The

test-data generation problem is that of identifying program input data which satisfy selected testing criteria. Test-data selection methods can be divided into two distinct classes – functional and structural testing [4].

In functional testing, the test-data are chosen with reference only to the functional specification of the software. In contrast, the input test-data for structural testing are selected based on the structure of the software (however the expected output-data are still devised with reference to the functional specification). Structural testing is supported by tools which monitor the coverage of a set of test-cases with respect to executed statements, branches, linear-code sequences and jumps (LCSAJs), etc. [28]. The test-data generation process is to identify test-data which will cause previously uncovered parts of the software to be executed. This can be extremely labour-intensive and expensive [22].

Automated test-data generators for structural testing can be divided into three classes – random, static and dynamic. Random test-data generation is easy to automate, but problematic [3]. Firstly, it produces a statistically insignificant sample of the possible paths through the software under test (SUT). Secondly, random test-data generation may never give 100% coverage. Finally, it may be expensive to generate the expected output-data for the large number of test-cases generated. Static approaches to test-data generation generally use symbolic execution. Much of the previous work on test-data generation has used this [5, 7, 8, 9, 19, 26]. Symbolic execution works by traversing a control flow graph of the SUT and building up symbolic representations of the internal variables in terms of the input variables. Branches within the code introduce constraints on the variables. It is

the solution to these constraints which represents test-data for the path. Several problems exist with this approach. It is difficult to analyse recursion, dynamic data-structures, array indices which depend on input data and some loop structures using symbolic execution. Also, the problem of solving arbitrary constraint systems is known to be undecidable. In contrast, dynamic test-data generation involve execution of the SUT and a directed search for test-data which meets the desired criterion.

The dynamic approach to test-data generation was first suggested in 1976 [24]. This uses a straight-line version of the SUT and function minimisation techniques to generate floating-point test-data. Korel's work [21, 22, 11] built on this, removing the need for the straight-line programs and generating test-data for more data-types. The approach uses a local direct search technique, which for complex search spaces may often get stuck in only locally optimal solutions, thus failing to find the desired test-data. As a way forward, Korel has suggested the use of global optimisation-based search as a way of overcoming this weakness. Recently, some work has investigated the use of global-optimisation as a way of generating structural test-data [32, 18, 17] to overcome the problem of locally optimal solutions in the search space.

The research presented in this paper builds on these approaches. The foundation of the work is the use of an heuristic global-optimisation technique to build a general framework for generating test-data. This framework has been applied to a number of testing problems, such as specification conformance testing and worst-case execution time testing [29, 30]. This paper focuses on the application of the framework to structural-testing problems and is structured as follows. The next section will introduce the problem of test-data generation for structural testing criteria. Following this, optimisation techniques are introduced and a method for expressing the test-data generation problem as an optimisation problem is presented. Extensions are then presented for boundary value analysis, assertion/run-time exception testing and also for component re-use testing. The results of some preliminary evaluation and experiments are then presented to illustrate the efficacy of the approach. Finally some conclusions and ideas for further work are outlined.

2 Structural Test-Data Generation

A control flow-graph is a directed graph which represents the control structure of a program. It can be described as follows [10]: $G = (N, E, s, e)$, where N is a set of nodes, E is a set of edges of the form (n_i, n_j) and s and e are unique entry and exit nodes such that

$s, e \in N$. A node, $n \in N$, is a sequence of statements such that if any one statement of the block is executed, then all are executed. This is also known as a *basic-block*. An edge $(n_i, n_j) \in E$ represents a possible transfer of control from the basic block n_i to the basic block n_j . For branch instructions the edges will be associated with a *branch predicate*. This describes the conditions which must hold for the branch to be taken.

A program is driven down a path in the control flow graph by the values of its input variables and the global state. This can be described as the vector $I = \langle x_1, x_2, \dots, x_n \rangle$. Each variable will have an associated domain, D_{x_i} , which can be determined from the variable's type. The total input space can be defined as the cross product of each of these domains, D . This allows a program input to be defined as $x \in D$.

For structural testing we are concerned with finding a program input such that the path executed will cover (or execute) the desired statement, branch, LCSAJ, etc. So, in essence, the test-data generation problem for structural testing is finding a set of program inputs, X (such that $X \subset D$) that achieves the desired coverage. Thus an automated test-data generator would, given a particular node to execute or path to follow, return test-data which achieves this. The dynamic approach to this problem involves a search for program input which forces execution of the desired part of the SUT.

For the search to succeed, it needs to be given some guidance. This guidance is given in the form of a cost-function which relates a program input to a measure of how *good* it is, with respect to the objective of executing a given part of the SUT. The amount of guidance given by the cost-function is one of the key elements in determining the effectiveness of the test-data generation. The other key factor is, of course, the search technique itself.

3 Optimisation-Based Solution

The input domain of most programs, D , is likely to be very large. A cost-surface could be formed by applying the cost-function to every possible program input. It is this surface which is effectively being searched when attempting to generate test-data. Given the complexities of software systems it is extremely unlikely that this cost surface would be linear or continuous for any effective cost-function. For example, a small change in input data can cause a different path to be traversed causing a radically different cost. The size and complexity of the search space therefore limits the effectiveness of simple gradient-descent or neighbourhood searches. These techniques

are likely to get stuck in locally optimal solutions and hence fail to find the desired test-data [27].

Heuristic-global optimisation techniques are designed to find good approximations to the optimal solution in large complex search spaces. General-purpose optimisation techniques make very few assumptions about the underlying problem which they are attempting to solve. It is this property which allows a general test-data generation framework to be developed for solving a number of testing problems. Simulated annealing is one such general purpose optimisation technique [20]. Simulated annealing is based on a modified neighbourhood search, to overcome the problems of locally optimal solutions. Simulated annealing allows movements which worsen the value of the cost-function based on a control parameter known as the *temperature*. Early in the search inferior solutions are accepted with relative freedom, but as the search progresses (or cools) accepting inferior solutions becomes more and more restricted. Accepting these inferior solutions is based on the premise that it is better to accept a short term penalty in the hope of longer term rewards. Simulated annealing has been used to assess the effectiveness of an optimisation based approach to the test-data generation problem¹.

To apply simulated annealing to a given domain it is necessary to devise a representation of candidate solutions and neighbourhoods and to develop a cost-function which can measure the quality of candidate solutions. The neighbourhood should represent the set of solutions which is in some respect *close* to a given candidate solution. The neighbourhood can thus be defined as follows.

| Basic Type | Neighbourhood |
|------------|---|
| INTEGER | \pm Some proportion of allowed range |
| FLOAT | \pm Some proportion of allowed range |
| BOOLEAN | TRUE or FALSE |
| ENUM. | Any value from the enumeration ² |

It is now necessary to define the cost-function which will guide the simulated annealing search to the desired test-data. The cost-function needs to indicate whether the desired statement (or branch, etc.) has been executed. A particular test-case will either sat-

¹Simulated annealing is a very simple optimisation algorithm to implement. It is for this reason that it has been used during the development of the prototype tool for automatic test-data generation. Other optimisation techniques, such as genetic algorithms or tabu-search, may be more effective. This will require investigation in the future.

²No neighbourhood range is used for enumeration types as the ordering of enumeration literals is not always significant.

isfy this criterion or will not, however this provides very little guidance to the search. Thus the cost-function needs to return good values for test-data that *nearly* executes the desired statement and bad values for test-data that is a *long way* from executing the desired statement. The branch predicates determine the path followed and are therefore vital in determining an effective cost-function. Given some input-data, assume that a path $P = \langle n_i, n_j, n_l, n_n \dots, n_z \rangle$ is followed through the program control-flow graph shown in figure 1. Now assuming test-data was required to execute node n_m , the sub-path $\langle n_i, n_j \rangle$ was correct. The sub-goal becomes finding test-data which still executes the correct sub-path, but which changes the outcome of the branch predicate at node n_j . It is the cost function which must guide the search to this goal.

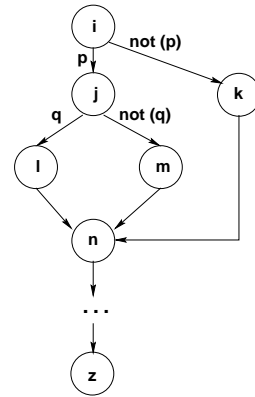


Figure 1: Sample Control Flow Graph

Branch predicates consist of relational expressions connected with logical operators. The cost-function has been designed such that it will evaluate to zero if the branch predicate evaluates to the desired condition and will be positive otherwise. The larger the positive value, the further the test-data is from achieving the desired outcome. This property gives an efficient stopping criteria for the search process, i.e. once the cost-function is zero. The cost-function is calculated as shown in table 1. In the table K represents a failure constant which is added to further punish incorrect test-data.

In order to evaluate the cost-function it is necessary to execute an instrumented version of the SUT. The instrumented SUT contains procedure calls which monitor the execution path taken and the evaluation of the branch predicates. A simple worked example will help to illustrate how the cost-function works in practice to provide guidance to the search. Figure 2 shows a simple program along with the instrumenta-

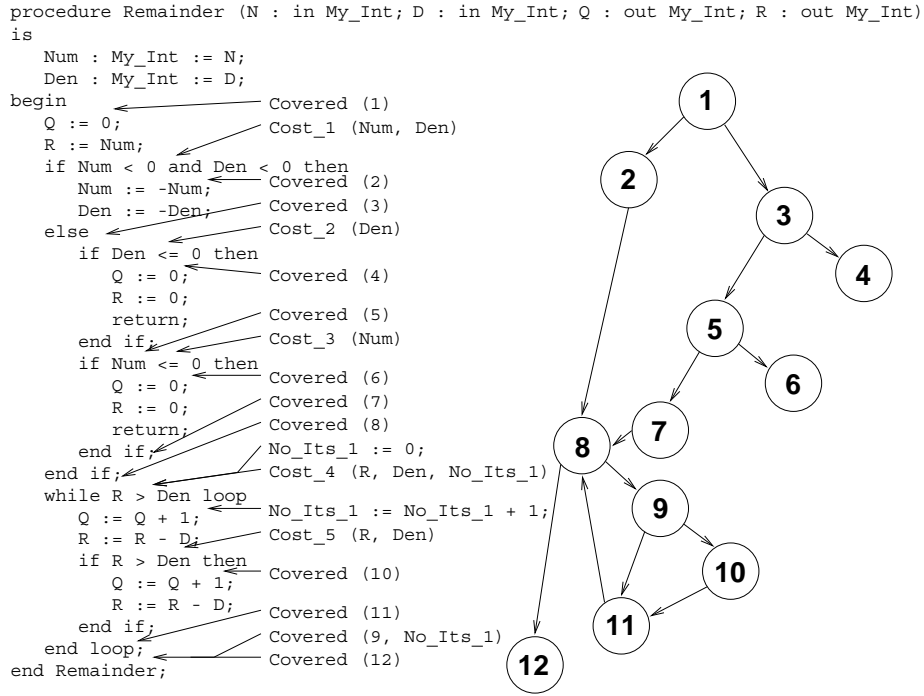


Figure 2: Example of Instrumented Software

| Element | Value |
|--------------|---|
| 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 $(b - a) + K$ |
| $a \vee b$ | $\min(\text{cost}(a), \text{cost}(b))$ |
| $a \wedge b$ | $\text{cost}(a) + \text{cost}(b)$ |
| $\neg a$ | Negation is moved inwards and propagated over a |

Table 1: Cost-Function Calculation

tion and its control-flow graph.

There are two types of procedure call added by the instrumenter – execution path monitor calls and branch evaluation calls. The Covered (X) calls work as follows:

- Record that node X has been executed.
- If X is current target node then move to next target node on target path.

The $\text{Cost}_N(\dots)$ calls replace the branch predicates in the SUT. These functions are responsible for adding to the overall cost the contribution made by each individual branch predicate which is executed and returning the boolean value of the predicate. They function as follows:

- If the target node is only reachable if the branch predicate is true then add cost of branch predicate to the overall cost for the current test-data.
- If the target node is only reachable if the branch predicate is false then add cost of \neg (branch predicate) to the overall cost for the current test-data.
- For loop conditionals, if the target number of iterations is greater than the actual number of iter-

ations then add cost of the conditional loop predicate to the overall cost for the current test-data. If the target and actual number of iterations are equal then add the cost of \neg (conditional loop predicate).

- If the current test-data causes an undesired branch to be taken (i.e. the true branch when the target node is on the false branch) then terminate the execution of the SUT and return the cost to the search procedure. This improves the performance of generating the desired test-data.
- Within loops, adding the cost of branch predicates is deferred until exit from the loop. At this point the minimum cost evaluated for that branch predicate is added to the overall cost. This prevents punishment of taking an undesirable branch until exit from a loop, as the desirable branch may be taken on subsequent iterations.

For example, consider the desired execution path $(1, 2, 9_{(2+)}, 10, -1), 9_{(2+)}$ can be read as execute the loop at node 9 two or more times and -1 can be read as execute any other nodes until SUT termination (i.e. don't care). The following illustrates how the cost-function works for this path and the SUT given in figure 2.

$$N = 2, D = -10$$

This test-data covers node 1 and thus moves the desired node to 2. Evaluating `Cost_1` gives the following: $Den < 0$ gives a cost of 0, $Num < 0$ gives a cost of $(2 - 0) + K$, giving a cost for $Num < 0 \wedge Den < 0$ of $(2 + K) + 0$. This indicates that the current test-data will not execute node 2, thus execution is terminated with an exception and the search procedure uses the calculated cost value to determine new test-data to try.

$$N = -5, D = -3$$

This test-data covers node 1 and thus moves the desired node to 2. Evaluating `Cost_1` results in zero being added to the cost as the condition is satisfied, covering node 2 and hence moving the desired node on to node $9_{(2+)}$. As the current number of iterations of node 9 is zero the desired path requires the loop conditional to evaluate to true. `Cost_4` evaluates the cost of meeting the loop conditional, which evaluates to zero (i.e. $R = 5, Den = 3$). `Cost_5` is now evaluated as follows: $R > Den$ gives $(3 - 2) + K$. This value is not directly added to the cost, but stored until exit from the enclosing loop where upon its

minimal value will be added. This prevents punishing not taking this branch on each iteration of the loop. At this point the number of iterations executed is one, so once more the desired path requires the loop conditional to be true. However, at this point `Cost_4` evaluates to $(3 - 2) + K$, indicating that the current test-data will not execute the loop with two or more iterations. Hence execution is terminated with an exception and the search procedure uses the calculated cost value to determine new test-data to try.

$$N = -15, D = -2$$

This test-data covers node 1 and thus moves the desired node to 2. `Cost_1` evaluates to zero, covering node 2 and hence moving the desired node on to node $9_{(2+)}$. `Cost_4` evaluates to zero on zero, one and two iterations, and from that point the evaluation of `Cost_4` contributes nothing to the overall cost as two or more iterations have already been executed, hence the outcome of the conditional does not matter. Within the loop, `Cost_5` evaluates to zero on the first, second and third iterations. This covers node 10 and moves the desired node on to -1 (i.e. the *don't care* node). On the fourth iteration, it evaluates to $(3 - 2) + K$. On exit from the loop, zero is added to the overall cost as this is the minimal value for `Cost_5`. Hence the overall cost at termination of the SUT is zero, indicating that valid test-data for the specified path has been found.

As can be seen from the above example, the cost-function gives a quantitative measure of the suitability of the generated test-data for the purpose of executing the specified node in (or path through) the SUT. The simulated annealing search uses this to guide its generation of test-data until either it has successfully found test-data with a zero cost or until the search freezes and no further progress can be made. At this point the system moves on to attempt generation of test-data for the next desired node or path. The entire process is supported by a prototype tool-set which supports the testing of Ada 95 [16] source.

Figure 3 shows the tools which make up this tool-set for the automatic test-data generation system. The tools are shown as rectangles. These tools communicate information using files with a defined syntax and semantics. The role of each of the tools is as follows.

Extractor This tool is responsible for processing the SUT. It extracts all the information necessary for the rest of the testing system and stores it in files. This tool is based on the GNAT [14] compiler

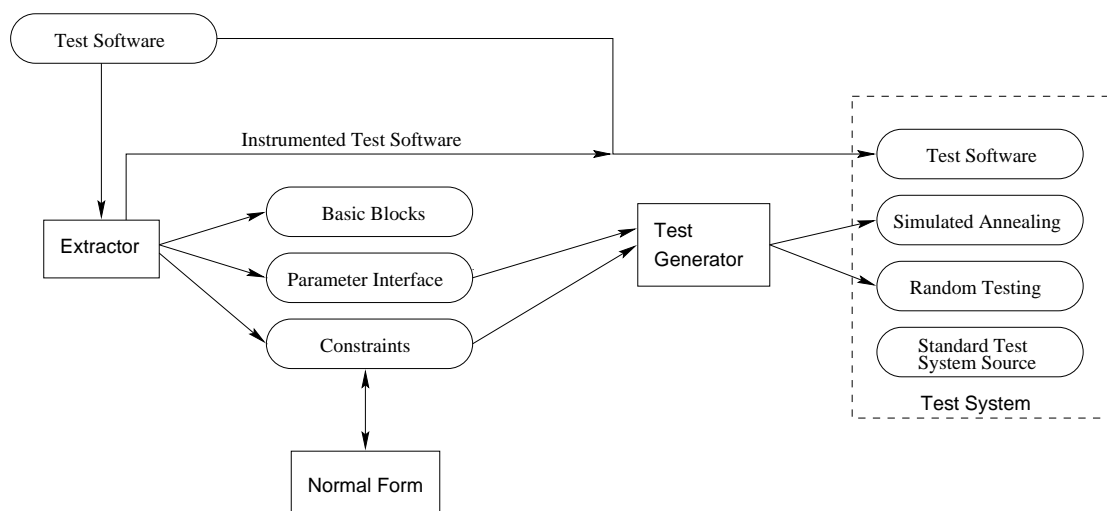


Figure 3: Prototype Tool-Set

front-end as this gives access to a full semantic tree³ for the SUT. This tool also generates the instrumented version of the SUT, when applicable to the software problem being addressed.

Normal Form This tool converts constraints into disjunctive normal form. This canonical form allows a simplified cost-function which gives more guidance to the search for a number of test-data generation problems – specification failures based on pre- and post-condition (see [29]), assertion/run-time exception testing and component re-use testing (see below).

Test-Generator This tool uses the information from the other tools to produce the test-system. When compiled and linked with the SUT this test-system will use the desired search strategy (currently only simulated annealing and random search⁴ are implemented) to generate test-data that meets the specified objectives. For structural testing this tool automatically generates a set of desired nodes which will give full conditional coverage for the SUT. This can be overridden by the user to allow generation of test-data to meet any structural objectives.

³This tool will be moved to an ASIS [15] implementation once a stable public release of ASIS-for-GNAT becomes available.

⁴Random search has been used as a benchmark for assessing the effectiveness of the simulated annealing search.

4 Other Structural Testing Problems

While it is useful to generate automatically test-data that meets structural objectives there are many other testing problems. For an automatic test-data generator to be generally useful it must be flexible. This section describes how the optimisation-based test-data generation framework which has been developed can be extended to address other structural testing problems. Previous work has illustrated how the framework may be extended to address other non-structural testing problems, [6, 29, 30].

4.1 Boundary Value Analysis

The input domain of a program, D , will be split into a number of sub-domains $D = \{d_1, d_2, \dots, d_n\}$. These correspond to the different paths through the program. Boundary value analysis considers test-data which lie close the boundaries of a sub-domain to be of a higher adequacy (meaning more likely to reveal program errors) than test-data which is further away. This is based on the assumption that a common coding error is to misplace the boundaries to these sub-domains. For example, consider the condition $X \leq 42$, if X is an integer type then the domain boundary will be at $X = 42$. However a simple coding error (i.e. ' $<$ ' instead of ' \leq ') would shift the boundary to $X = 41$.

To address the problems of boundary value analysis when generating test-data the cost-function needs to be biased towards lower costs for test-data at or near a boundary. This can be achieved with a simple extension to the previous cost function. Consider the above example of $X < 42$, if the desired path requires this condition to be false, the test-data genera-

tor needs to generate $X \geq 42$. To bias this towards the boundary case of $X = 42$ the previous cost-function is augmented with the cost of $X = 42$. The value of this is divided by a scaling factor. This allows the cost of $X \geq 42$ to dominate, but also allows a bias towards the boundary. For example, consider $X = 100$, the previous cost function would return zero for this test-data, however the new cost function would give $0 + \frac{(100-42)+K}{scale}$. $X = 42$ would give a cost of zero, thus it can be seen that the test-data generation is biased towards selecting boundary values. This extension means that the desired test-data will not always have a cost of zero. This removes the efficient stopping criterion defined earlier. However, it simply means that the search will continue until it freezes, at which point it will terminate, so the test-data generation may take a little longer.

For more complex branch predicates, the particular boundary may be more difficult to identify. The toolset will automatically try to generate a test-data set which is on the boundary of each of the individual relational expressions in the predicate. This can be overridden by the user to allow combined boundaries to be targeted. For example, for the predicate $X > 5 \wedge Y \leq 10$, it may be desirable to target the boundaries of both X and Y simultaneously, resulting in $X = 6$ and $Y = 10$.

4.2 Assertion/Run-Time Exception Testing

Assertions and run-time exceptions are a useful tool for automatic detection of run-time errors. Assertions express conditions which must be true at a given point in the program's execution. Run-time exceptions occur when rules or conditions of the language are violated [1]. For example, numeric under- or over-flow, divide-by-zero, out-of-range errors, etc. The goal of the test-data generator for assertion and run-time exception testing is to find test-data which breaks the assertion or causes the run-time exception. To meet this goal the cost-function must guide the search in two ways. First, the search must be guided to test-data that executes the statement associated with the assertion or possible exception. Secondly the search must be guided to test-data that causes the assertion to be false, or the exception condition to be true.

Korel has shown that assertion testing can effectively be reduced to the problem of executing a particular statement [23]. In effect, the assertion is expanded to a conditional branch, the goal then becomes that of generating test-data which executes the *else* part of this conditional branch. The generation of such test-data can therefore be achieved with exactly

the same cost-function as for structural testing. All that is required is an instrumented version of the SUT with the assertions converted in this manner. Figure 4 shows how the assertion is converted.

```

Assert (X >= 42);

-- Becomes

if X >= 42 then
  Assert (True);
else
  -- Assertion violated in this branch
  -- Find test-data to execute this node
  Assert (False);
end if;

```

Figure 4: Assertion Conversion

Exception conditions can be specified in a similar manner to assertions. Figure 5 illustrates this. As for assertion testing, the cost-function guidance is twofold – firstly to execute the statement associated with the exception condition and then to cause the exception condition itself. Again, the cost-function for finding test-data which executes the statement associated with the exception condition is the same as that for structural testing. However, this is now augmented with an additional call just before this statement is executed. This evaluates the cost of the exception-condition holding. To allow improved guidance the exception-condition is converted to Disjunctive Normal Form (DNF). For example, $A \wedge (B \vee C)$ would become $(A \wedge B) \vee (A \wedge C)$. A solution to any one disjunct then represents a solution to the entire condition. Each disjunct is considered in turn. The search process aims to find test-data which executes the desired statement and satisfies the current disjunct. The cost-function for the disjuncts is the same as that used for branch predicates. For example, consider the program and exception condition shown in figure 5.

```

function F (X : Integer) return Integer
is
  Tmp : Integer := X;
begin
  -- # exception when ((X-1) * (X-5)) = 0;
  Tmp := (X ** 4) / ((X - 1) * (X - 5));
  return Tmp;
end F;

```

Figure 5: Exception Example

When $X = 0$ the exception-condition cost-function will evaluate to $abs((-1 \times -5) - 0) + K$. However, when $X = 1$ or $X = 5$, this cost-function will evaluate to zero, indicating that test-data has been found which will cause the specified exception.

4.3 Component Re-use Testing

When re-using components it is vital that the environment within which they are re-used meets the assumptions that were made when the component was developed. This can be stated as $\forall x \in D \bullet pre(x) = \text{True}$ – that is for all input-data in the component’s domain the pre-condition holds true. Where this is not the case the component is being re-used in a unsafe manner. Thus the goal of re-use testing can be stated as finding test-data which causes the precondition of a re-usable component to be broken. The cost-function remains almost identical as for exception condition testing. However, in this case the tools must convert the *negated* pre-condition to DNF. The component is instrumented such that it evaluates the cost of meeting a given disjunct in this negated pre-condition. This value is added to the overall cost for the test-data. Thus the search is guided to test-data which executes the call of the component, but also breaks the pre-condition of the component.

As an example, consider a simple function that does integer division by repeated subtraction. The pre-condition to this function is that both the numerator and denominator are greater than zero. The program in Figure 6 uses this division function.

Figure 6: Re-use Example

When $X = 0$ the cost of the disjunct, numerator ≤ 0 , evaluates to $(0 + 5) * (0 + 2) - 0 = 10 + K$. This indicates that this pre-condition disjunct is not violated. However with $X = -5$ or $X = -2$ the cost evaluates to zero indicating that the pre-condition is violated. Thus this test-data illustrates a (re)use of the divide function which is unsafe. In general the components to be re-used are more complex, and so are the corresponding pre-conditions, but the same approach can be applied.

5 Evaluation

The purpose of the evaluation is to investigate the effectiveness of the automatic test-data generation framework. This preliminary evaluation has used a collection of small programs written in Ada 95. These have been tested for 100% branch coverage with a

boundary value analysis bias. A number of the programs contain assertions or could cause run-time exceptions. The assertion falsification/run-time exception test-data generation was used to attempt to find test-data to highlight these problems. Component re-use testing has been evaluated using a small number of programs with explicit pre-conditions specified as Spark Ada proof-context annotations [2]. Harness software was produced which used these components in a manner which for some inputs would violate the component’s pre-condition.

The evaluation was performed on a Pentium Pro 200MHz running Linux. The test-data generation system is written in Ada 95 and was compiled using the GNAT compiler. Each of the test-data generation attempts was repeated 50 times and the results reported are the mean values.

The following table shows the results of generating test-data for branch coverage for a number of programs (INP - size of input domain, TS - size of test-data set generated, C - coverage, T - search time). These programs vary in length from 20 to 200 lines of code. It can be seen that 100% branch coverage was achieved in all but one case. For this case on 48 of 50 attempts 100% coverage was achieved, on the other two attempts the search failed to cover one of the branches. This was overcome by tuning the simulated annealing parameters to slow down the cooling. While this lengthened the search time it meant that covering test-data was always found.

| Program Name | INP | TS | C | T |
|---------------|-----------|----|------|-------|
| Middle | $1e + 12$ | 7 | 100% | 1.3s |
| Find | $1e + 44$ | 6 | 100% | 2.0s |
| Remainder | $1e + 8$ | 5 | 100% | 2.4s |
| Tomorrow | 286,440 | 16 | 100% | 4.8s |
| Triangle | 8,120,601 | 14 | 100% | 7.2s |
| Bigint Divide | $1e + 50$ | 29 | 99% | 36.2s |

The table below shows the results of the error directed testing – assertion/exception testing and component re-use testing. In each case, a number of incorrect versions were produced and the results indicate the ability of the generated test-data to illustrate the problem.

| Program Name | Number. Incorrect | Found | Search Time |
|-----------------|-------------------|-------|-------------|
| Square | 1 | 1 | <0.5s |
| Int Square Root | 3 | 3 | 1.1s |
| Remainder | 2 | 2 | 1.3s |
| Convert | 7 | 7 | 17.2s |

The results are extremely encouraging. They show the optimisation-based test-data generation approach to be efficient and effective. However, these are only preliminary results. The software tested has been relatively small. Further evaluation is vital to assess the usefulness of the approach on more *realistic* software.

6 Conclusions

Many of the approaches for automated software test-data generation presented in the literature are inflexible or have limited capacity. Optimisation techniques in contrast are a flexible and efficient approach to solving *difficult* problems. To allow the optimisation based framework to generate test-data for new testing criteria it is necessary only to devise a suitable cost-function.

As with all testing approaches, we can only show the presence of faults not their absence. Indeed, the failure of the search to find exception conditions or re-use problems does not indicate that the software is correct, only that the search failed. However, given an intensive directed search for a specific problem, the failure to find the problem does allow increased confidence in the quality of the software which is after all the aim of testing. The tools we provide need not be of high integrity (that is not to say they need not be of a high quality!) even when testing safety critical code. We view them simply as generating test-data that can be checked by other means, i.e. use of a test-harness to check that the generated test-data does in fact cause the desired exception. This is important as the algorithms are stochastic and it is difficult to reason about their efficacy for application to arbitrary code.

6.1 Further Work

The results presented above show that it is possible to use optimisation techniques to develop a framework to automate the generation of test-data for a number of common software testing problems. This framework can be used to generate test-data efficiently. The complete automation of the generation process significantly reduces the effort required to obtain test-data compared with the traditional manual approach to generating test-data. More work on several fronts is needed to further assess the optimisation based approach to providing a useful, generalised framework for automated test-data generation. Perhaps the most important is that of gathering empirical evidence as to the effectiveness of the technique. It is essential to evaluate the approach using real, large-scale software to assess how effectively test-data can be generated. This evaluation will be carried out using real commercial aviation engine controller code shortly.

There is a variety of other optimisation techniques which could be examined. A detailed comparison of the various optimisation techniques to discover their relative strengths and weaknesses would be required. Tabu-search [12] and genetic algorithms [13] are two such optimisation techniques which are suitable for investigation. Some preliminary work on methods to apply tabu-search and genetic-algorithms has already been carried out [31]. However, integration into the prototype tool-set is required to allow a full assessment of their performance.

Optimisation techniques will never be able to guarantee their results. However, it may be possible to devise software metrics which can give guidance in a number of areas – to suggest which optimisation techniques will give the best results; to suggest suitable parameter values for the optimisation techniques; and also to give an indication as to the likely quality of the result.

7 Acknowledgements

This work was funded by grant GR/L42872 from the Engineering and Physical Sciences Research Council (EPSRC) in the UK as part of the CONVERSE project.

References

- [1] John Barnes. *Programming in Ada 95*. Addison-Wesley, 1995.
- [2] John Barnes. *High Integrity Ada: The SPARK Approach*. Addison-Wesley, 1997.
- [3] B. Beizer. *Software Testing Techniques*. Thomson Computer Press, 2nd edition, 1990.
- [4] B. Beizer. *Software System Testing and Quality Assurance*. Thomson Computer Press, 1996.
- [5] R. Boyer, B. Elspas, and K. Levitt. Select – a formal system for testing and debugging programs by symbolic execution. *Proceedings International Conference on Reliable Software*, pages 234–245, 1975.
- [6] John Clark and Nigel Tracey. Solving constraints in law. Law/d5.1.1(e), European Commission - DG III Industry, 1997. Legacy Assessment Workbook Feasibility Assessment.
- [7] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, September 1976.

- [8] R. Demillo and A. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.
- [9] R. Demillo and A. Offutt. Experimental results form an automatic test case generator. *ACM Transactions on Software Engineering and Methodology*, 2(2):109–127, April 1993.
- [10] Narsingh Deo. *Graph Theory with Applications to Engineering and Computer Science*. Prentice-Hall, 1974.
- [11] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, 1996.
- [12] Fred Glover and Manuel Laguna. *Modern Heuristic Techniques for Combinatorial Problems*, chapter 3 – Tabu Search, pages 70–150. McGraw Hill, 1993.
- [13] D. E. Goldberg. *Genetic Algorithms in Search, Optimisation and Machine Learning*. Addison-Wesley, 1989.
- [14] Ada Core Technologies Inc. The gnat ada-95 compiler, 1997. <http://www.gnat.com/>.
- [15] ISO/IEC 15291. *Ada Semantic Interface Specification - ASIS*, (p) edition, 1997.
- [16] ISO/IEC 8652:1995. *Ada 95 : Language Reference Manual*, 1995.
- [17] B. Jones, H. Sthamer, and D. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.
- [18] B. F. Jones, H. H. Sthamer, and D. E. Eyres. Generating test-data for Ada procedures using genetic algorithms. In *Genetic Algorithms in Engineering Systems: Innovations and Applications*, pages 65–70. IEEE, September 1995.
- [19] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [20] S. Kirkpatrick, Jr. C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.
- [21] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.
- [22] B. Korel. Automated test data generation for programs with procedures, 1996.
- [23] Bogdan Korel and Ali M. Al-Yami. Assertion oriented automated test data generation. In *18th International Conference on Software Engineering*, pages 71–80. IEEE, March 1996.
- [24] W. Miller and D. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, SE-2(3):223–226, September 1976.
- [25] M. Ould. Testing - a challenge to method and tool developers. *Software Engineering Journal*, 6(2):59–64, March 1991.
- [26] C. Ramamoorthy, F. Ho, and W. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, SE-2(4):293–300, 1976.
- [27] V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith, editors. *Modern Heuristic Search Methods*. Wiley, 1996.
- [28] Ian Sommerville. *Software Engineering*. Addison-Wesley, 4th edition, 1992.
- [29] Nigel Tracey, John Clark, and Keith Mander. Automated program flaw finding using simulated annealing. In *International Symposium on Software Testing and Analysis*, pages 73–81. ACM/SIGSOFT, 1998.
- [30] Nigel Tracey, John Clark, and Keith Mander. The way forward for unifying dynamic test case generation: The optimisation-based approach. In *International Workshop on Dependable Computing and Its Applications*, pages 169–180. IFIP, 1998.
- [31] Nigel J. Tracey. Test-case data generation using optimisation techniques – first year DPhil report. Department of Computer Science, University of York, 1997.
- [32] Alison Lachut Watkins. The automatic generation of test data using genetic algorithms. *Proceedings of the 4th Software Quality Conference*, 2:300–309, 1995.