

Automated Program Flaw Finding using Simulated Annealing

Nigel Tracey

John Clark

Keith Mander

Department of Computer Science
University of York
Heslington, York
YO1 5DD, England.
+44 1904 432727
{njt, jac, mander}@cs.york.ac.uk

1 Abstract

One of the major costs in a software project is the construction of test-data. This paper outlines a generalised test-case data generation framework based on optimisation techniques. The framework can incorporate a number of testing criteria, for both functional and non-functional properties. Application of the optimisation framework to testing specification failures and exception conditions is illustrated. The results of a number of small case studies are presented and show the efficiency and effectiveness of this dynamic optimisation-base approach to generating test-data.

1.1 Keywords

Automatic test-case generation, software testing, formal specifications, exception conditions, optimisation techniques, simulated annealing.

2 Introduction

Software testing is an expensive process, typically consuming at least 50% of the total costs involved in developing software [3], while adding nothing to the functionality of the product. It remains, however, the primary method through which confidence in software is achieved. Automation of the testing process is desirable both to reduce development costs and also to improve the quality of (or at least confidence in) software. While automation of the testing process – the maintenance and execution of tests – is taking hold commercially, the automation of test-data generation has yet to find its way out of academia. Ould has suggested that it is this automation of test-data generation which is vital to advance the state-of-the-art in software testing [17].

Many techniques for generating test-data have been developed. These can broadly be classified into static and dynamic methods. Static techniques do not require the software under test to be executed. They generally use symbolic execution to obtain constraints

on input variables for the particular test criterion. Solutions to these constraints represent the test-data. Many of the limitations of the static techniques come from their use of symbolic execution. 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 intractable. However, it is often the case that the constraints which arise from *real* software are in a subset of general constraint systems which can be solved. In contrast, dynamic methods require that the software under test is executed. Dynamic methods generally involve a directed search for test-data which meets a desired criterion. However the computational expense involved in repeated execution can mean that obtaining test-data is computationally intractable for complex software with a large parameter space. Figure 1 shows the major approaches presented in the literature [5, 15, 12, 6, 22, 16, 13, 10]. The arrows in the figure indicate the progression of ideas and methods through the various approaches. While the 1980s saw the emergence of formal methods as a focus of academic research attention, the 1990s has seen the growth once more of software testing research. This is, possibly, partly due to the realisation that the use of formal methods does not obviate the need for good software testing and also the realisation that software testing can provide a cost-effective means of verification.

All of these previous approaches focus on generating test-case data in a narrow area; they all aim to generate test-data for testing functional properties, with most selecting test-data using structural testing criteria. The application of many of these techniques is limited by lack of generality. This lack of generality can be seen in the limited data-types or control-flow structures which some of the techniques can process and also in the limited testing criteria of the others. The aim of the present work is to develop a generalised framework for test-case data generation. The objectives are to automate the generation of test-case data to satisfy both black and white box testing of functional properties and also non-functional properties. This paper focuses on two applications of this general framework: the first relates to falsification of program specifications (functional testing), the second relates to exception condition testing (non-functional testing).

The generality of the approach comes from the use of optimisation techniques. Optimisation techniques are a flexible and powerful search method with an ability to find good results for many extremely difficult problems. The next two sections outline methods for guiding a search to select test-data to illustrate specification failures and exception conditions. Because of the size and complexities of the search space it is unlikely that exhaustive or simple neighbourhood search strategies would obtain good test-data. Therefore

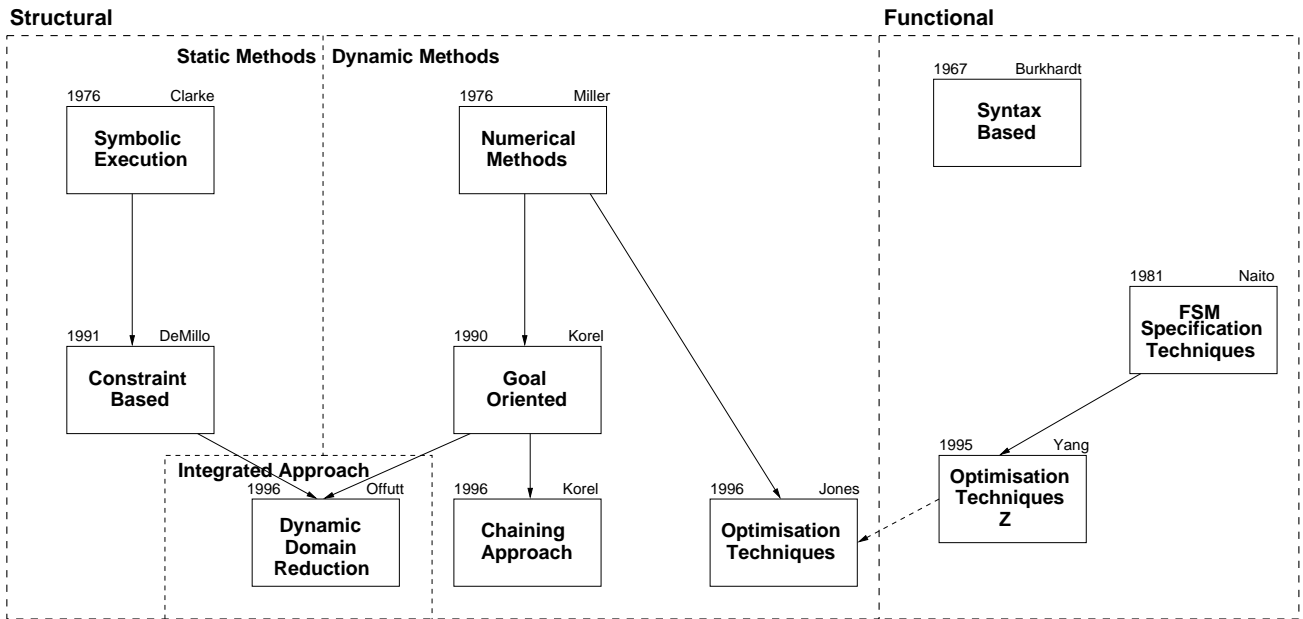


Figure 1: Some Approaches to Automating Test-Case Data Generation

optimisation techniques, specifically simulated annealing¹, are introduced. Some case-studies are then given to show how simulated annealing manages to generate good test-case data despite the complexities and size of the search space.

3 Testing Specification Failure

Dynamic testing of functional properties can be divided into two categories, functional and structural. Structural tests are considered white-box, the underlying code of the software being used to determine the test-data. In contrast functional testing is considered black-box, the test inputs and expected outputs are derived solely from the functional specification. To allow functional test-data to be generated automatically it is obviously necessary for the specification to be in a sufficiently formal notation. Testing against a formalised specification has a number of advantages – it has the potential to detect missing functionality, it is independent of the implementation language and it provides scope for an automated test oracle.

For highly safety-critical software system it is often the case that formal proofs must be developed. These proofs must show that the software is a refinement of the formal specification. Producing formal-proofs is a complex, time-consuming and expensive process. If the putative properties are simply untrue then attempting a proof (which will inevitably fail) is an extraordinarily expensive method to find errors in the software. To demonstrate that a proof will fail it is only necessary to find one test-case which shows the

¹Simulated annealing is a very simple optimisation algorithm to implement, it is for this reason that it has been used during the development and assessment of prototype tools. There is no suggestion that is the superior optimisation technique, an investigation as to the merits of other optimisation techniques (i.e. genetic algorithms, tabu-search, etc.) for generating test-data is something which is planned for the future.

specification is not satisfied. Indeed it is often stated that a successful test-case is one which illustrates software error (Dijkstra 1972). For the purposes of investigating an optimisation-based approach to testing specification failure the formal specifications have been specified in a SPARK-Ada proof contexts [18, 2] like notation. These proof contexts consist of a pre- and post-condition for the subprogram. These conditions consist of an Ada expression, extended to allow logical implication and equivalence. Figure 2 shows how the formal specification is expressed as part of the subprogram specification for a simple integer square-root subprogram.

```

procedure Int_Root (N : Integer; Root : out Integer);
--# pre N >= 0;
--# post (Root * Root <= N) and
--# (N < (Root + 1) * (Root + 1));

```

Figure 2: Formal Specification Notation

Variables in the pre-condition refer to input values (i.e. prior to execution). Variables in the post-condition can refer to either output values or input values (when decorated with a \sim), when a variable is both imported and exported from a routine. To show that a particular implementation does not fully meet its specification it is necessary to find a test-case which prior to execution satisfies the pre-condition but after execution does not satisfy the post-condition. Such a process can be difficult, time-consuming and hard to manage (as the process lacks a quantified measure of completeness). However if such a process could be fully automated then it could be routinely applied prior to attempting proofs or other traditional testing methods. The following describes a method for carrying out this automation using simulated annealing.

3.1 The Dynamic Approach

To allow a dynamic approach to be used it is necessary to devise an objective function which will guide the search. The solution we are searching for is a test-case which satisfies the pre-condition before execution of the subprogram and the negated post-condition² after execution. Obviously any particular test-case will either meet this criterion or it will not. However this is not sufficient to guide the search. We therefore need the objective function to return good values for those test-cases which *nearly* meet the criterion and worse values for those which are a long way away from meeting the criterion. For example, consider the constraint $X > 50$. If $X = 49$ then this constraint is a lot *nearer* to being true than when $X = 2$ (however it remains a fact that they are both false, we just consider one to be less false than the other!).

The objective function is calculated as follows. Firstly the pre-condition and negated post-condition are converted to Disjunctive Normal Form (DNF). For example the condition $A \rightarrow (B \vee (C \wedge D))$ would become $\neg A \vee B \vee (C \wedge D)$. A solution to any one disjunct (i.e. $\neg A$, B or $(C \wedge D)$) then represents a solution to the entire condition. All possible pairs of a single pre-condition disjunct and post-condition disjunct are then formed using conjunction. These pairs can be considered as an encoding of one of the possible ways in which the software can fail. The search process attempts to find a solution to each of these pairs in turn. Each term within the pair adds a value to the pairs overall cost according to the rules shown in Table 1 (the value K in the table refers to a failure constant which is always added if the term is not true). If all terms are true then it can be seen that the overall cost will be zero, this becomes the stop-condition for the simulated annealing search.

Term	Value
Boolean	if TRUE then 0 else K
$a = b$	if $\text{abs}(a - b) = 0$ then 0 else $\text{abs}(a - b) + K$
$a \neq b$	if $\text{abs}(a - b) \neq 0$ 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 $(b - a) + K$

Table 1: Term Values

Using this objective function simulated annealing can be used to search for test-data to illustrate flawed functionality.

3.2 Simple Example

A simple example will help illustrate how this guides the search process. Consider a simple wrap-round incrementer routine as specified in Figure 3. This simple routine should count from 0 to 10 and then wrap-around back to 0 again.

The first step is to convert the pre-condition and the negation of the

²Satisfying the negation of the post-condition is equivalent to falsifying the post-condition.

```

procedure Wrap_Inc (N : in out Integer);
--# pre  N >= 0 and N <= 10;
--# post (N< 10 -> N = N + 1) and
--#      (N = 10 -> N = 0);

```

Figure 3: Specification of Wrap-Round Increment Routine

post-condition to DNF. It is then necessary to form all possible pairs of pre-condition and post-condition disjunction. The pre-condition is already in DNF and consists of a single disjunct. The DNF of the negated post-condition consists of two disjuncts (shown in parenthesis below) hence there are two possible pairings as follows.

$$N \geq 0 \wedge N \leq 10 \wedge (N < 10 \wedge N \neq N + 1) \quad (1)$$

$$N \geq 0 \wedge N \leq 10 \wedge (N = 10 \wedge N \neq 0) \quad (2)$$

Let us consider the second of these pairs. If, say, $N = 2$ then each term contributes the following to the overall cost (assuming that N after execution will equal 3).

Term	Cost Contribution
$N \geq 0$	0
$N \leq 10$	0
$N = 10$	$\text{abs}(2 - 10) + K = 8 + K$
$N \neq 0$	0

It can be seen that the overall condition is false, however the cost gives a measure of how false. If we now consider $N = 10$ then each term contributes the following (assuming that the software is implemented correctly and that N after execution will equal 0).

Term	Cost Contribution
$N \geq 0$	0
$N \leq 10$	0
$N = 10$	0
$N \neq 0$	$0 - 0 + K = K$

This shows that while the overall condition remains false it is now *less* false than it was before. Indeed if the software had been implemented incorrectly (for example it wrapped-round at 11 not 10 possibly due to $>$ rather than \geq being used) then the cost would be reduced to 0, hence this would represent a test-case which illustrated a specification failure. This flawed implementation appears at the end of this paper.

Using this method of assigning a cost to a particular test-case through an objective function, guidance can be given to the search process. Due to the complexities of software systems it is extremely unlikely that this cost surface would be linear or continuous, for example a small change in input data can cause a different path to be traversed causing a radically different cost value. This fact limits the usefulness of simple hill-climbing or neighbourhood search strategies. It is also possible that the search space will be extremely large making exhaustive search strategies computationally infeasible. For these reasons heuristic global optimisation techniques have been used. The search aims to minimise the value of the objective-function for each of the pre- and post-condition pairs in turn. Whenever a cost of zero is found the input-data represents a test-case to show specification failure.

We therefore use simulated annealing [1] to find suitable test-cases, and here present a number of examples to demonstrate its effectiveness in uncovering implementation errors with respect to the specifications. It is also possible to extend these ideas to search for test-cases which meet other criteria. One such extension is to target the search at finding test-cases which cause exception conditions; this is discussed in the following section. Other extensions include finding violations of safeness conditions, boundary analysis, etc.

4 Testing Exception Conditions

As with specification testing, when testing exception conditions it is usually more interesting to focus on the failure case (i.e. the case where an exception condition arises). Perhaps the most common form of exception condition is that of numeric overflow or underflow (this is known as a `Constraint_Error` in Ada). Indeed in a SPARK Ada program a `Constraint_Error` is the only exception condition which can occur. The remaining exception conditions cannot arise because of the rules of SPARK Ada. It is possible to use the SPARK tools to prove that `Constraint_Errors` cannot arise (the run-time checks [2]). The SPARK Examiner simply inserts additional conditions at appropriate places to ensure that run-time exception conditions cannot occur, the developer then has to prove the resulting conditions. However, as with specification proofs, this can be a complex, time-consuming and expensive way to find errors.

The conditions which may cause exceptions to be raised can be specified in a similar way to the formal specification. These are the negations of the conditions inserted by the SPARK tools as part of the exception-free proof process. The conditions represent the constraints which must be satisfied at a particular point during the execution for an exception condition to occur. Figure 4 shows an example for a simple integer square routine.

```

procedure Square (N : in Integer; S : out Integer) is
begin
  --# exception not (N*N >= Integer'First and
  --#                    N*N <= Integer'Last);
  S := N * N;
end Square;

```

Figure 4: Exception Condition Notation

4.1 The Dynamic Approach

Again the search for test-case data requires guidance using an objective function. The same objective function as for specification failure testing can be used. However the values used for variables must be those intermediate values at the exact point in the code where the exception condition is stated. For example, converting the exception condition from figure 4 to DNF results in the following:

$$\neg(N * N \geq \text{Min_Int}) \vee \neg(N * N \leq \text{Max_Int})$$

Assuming the maximum integer is 2^{31} then we can see how this guides the search process for the second disjunct. If $N = 100$ then the objective function is equal to $2^{31} - (100 * 100) = 2147473648$, which is a long way from being true. However, if $N = 46340$ then the value of the objective function decrease to 88048 which is much *closer* to being true. Indeed, if $N = 50000$ then the objective function becomes zero indicating that the condition has been met and

also that an exception condition will occur. The objective function is further complicated because it also needs to guide the search to test-data which executes the desired statement. Details of this will be presented in a later paper, a summary can be found in [20].

The search space remains large, complex and unlikely to be linear and continuous. It is therefore necessary to use a search strategy which can obtain good result despite these problems. The next section introduces optimisation techniques and the specific technique of simulated annealing. Simulated annealing has been used both because of its simplicity and its ability to obtain good results. Following this some simple case studies are presented to show the effectiveness of the simulated annealing search in finding test-cases to cause exception conditions.

5 Optimisation and Simulated Annealing

Optimisation techniques make very few assumptions about the underlying problem which they are attempting to solve. It is this property which allows them to be applied to a wide variety of problems. Optimisation techniques are simply directed search method which attempts to find minimal (or maximal) values of a particular objective function. Thus to apply optimisation techniques to a problem it is only necessary to devise an objective function to direct the search. Simulated annealing is one such optimisation technique. It has been used to assess the feasibility of an optimisation-based approach to generalised test-case data generation during this work.

Simulated annealing is based on the idea of neighbourhood search [19]. The ideas were first published in 1953 [14], thirty years later Kirkpatrick [11] suggested a form of simulated annealing could be used to solve complex optimisation problems. The algorithm works by selecting candidate solutions which are in the neighbourhood of the given candidate solution. Better candidate solutions (i.e. with respects to an objective function) are always accepted, however worse candidate solutions are accepted in a controlled manner. The idea is that it is better to accept a short-term penalty in the hope of finding significant rewards longer term. In accepting an inferior solution the search aims to escape from locally optimal solutions. A control parameter (known as the temperature) is used to control the acceptance of inferior candidate solutions. Initially the temperature is high allowing almost unrestricted movement around the search space. The temperature is gradually reduced during the search constraining the acceptance of inferior candidate solutions. Eventually the process *freezes* and no inferior solutions can be accepted, reducing the search to simple hill climbing. Figure 5 shows the basic outline of the simulated annealing algorithm.

Simulated annealing is a general purpose search strategy. Much of the algorithm remains completely unchanged across problem domains. Dowsland [7] classifies the implementation decisions which must be made into two categories – generic and problem specific. The generic decisions represent the parameters of the algorithm itself. The main generic implementation decision is how the temperature will be reduced, this is known as the *cooling schedule*. The problem specific decisions include the representation of the solution space, the neighbourhood and how to quantify the cost (*goodness*) of a solution. As we are dealing with software test-data generation the representation of a candidate solution and neighbourhood can remain fixed across this domain. A candidate solution is obviously a collection of data values which represent a test-case. This leads to the natural representation of a candidate solution as a number of data items which are derived from the underlying data types of the programming language used for the software under test. The neighbourhood should represent the set of candidate solutions which are

```

procedure Simulated_Annealing is
  Current_Solution, New_Solution : Solution;
  Current_Cost, New_Cost : Float;
  Temperature : Float;
  Length : Integer;
begin
  INITIALISE (Current_Solution, Temperature, Length);
  -- Start with :-
  -- Random solution
  -- Temperature to allow almost free movement
  -- Length to allow some solutions to be accepted

  Current_Cost := COST (Current_Solution);
  -- Obtain cost using objective function

  loop
    for I in 1 .. Length loop
      New_Solution := GENERATE (Current_Solution);
      -- Generate new solution in the neighbourhood
      -- of the current solution

      New_Cost := COST (New_Solution);

      if New_Cost < Current_Cost then
        Current_Solution := New_Solution;
        Current_Cost := New_Cost;
        -- Accept Solution
      else
        if (exp ((Current_Cost - New_Cost) /
          Temperature) > Random (0, 1))
        then
          Current_Solution := New_Solution;
          Current_Cost := New_Cost;
          -- Accept Solution
        else
          -- Do not accept solution
          null;
        end if;
      end if;
    end loop;

    COOL (Length, Temperature);
    -- Reduce the temperature and set the number
    -- of solution to examine at the reduced temperature

  exit when STOP_CRITERION;
end loop;
end Simulated_Annealing;

```

Figure 5: Simulated Annealing Algorithm

in some respects *close* to a given candidate solution. Given the representation of candidate solutions, the neighbourhood can be defined as follows for the fundamental Ada types.

Basic Type	Neighbourhood
INTEGER	\pm Some proportion of allowed range
FLOAT	\pm Some proportion of allowed range
BOOLEAN	TRUE or FALSE
ENUMERATION	Any value from the enumeration ³

Methods for quantifying the cost of a candidate solution depend entirely upon the software testing problem being addressed. These were discussed in previous sections for the problems of finding specification failures and exception conditions.

The following section shows some simple case studies to illustrate the effectiveness of the optimisation based (using simulated annealing) approach in generating test-case data to illustrate specification failures or exception conditions. The results show that this can be done both effectively and efficiently.

6 Case Studies

6.1 Specification Testing

This section presents the results of using simulated annealing to find test-case data highlighting specification failures for three Ada programs – middle, bubble sort and tomorrow. The incorrect implementations of these three routines can be found at the end of this paper.

The specification for the middle routine is shown in Figure 6. It states that given three integer values this routine should return the middle numeric value. However if two of the input values are the same the return value should be the other input value. If all three input values are identical then any return value will satisfy the given specification. The implementation of the routine, however, always returns the first input value when any of the input values are the same. Hence, the desired test-case is one where two of the input values are the same.

```

function Middle (A, B, C : Integer) return Integer;
--# pre True;
--# return M => ((B<A and A<C) -> M=A) and
--# ((C<A and A<B) -> M=A) and
--# ((A<B and B<C) -> M=B) and
--# ((C<B and C<A) -> M=B) and
--# ((A<C and C<B) -> M=C) and
--# ((B<C and C<A) -> M=C) and
--# (B=C -> M=A) and (A=C -> M=B) and (A=B -> M=C);

```

Figure 6: Specification of Middle Routine

The specification for the bubble routine is shown in Figure 7. It states that given an array of input values the routine should sort them into ascending order⁴. The implementation of this routine, however, does not perform enough iterations to sort the array when the smallest input values appears as the last element in the input

³No neighbourhood range is used for enumeration types as the ordering of enumeration literals is not always significant.

⁴In fact, the specification is too weak because it only states that the values in the returned array should be ordered. It does not state that they must be a permutation of the input values.

array. Hence the desired test-case is one where the smallest value in the array is in the last element.

```

type My_Int is Integer range 0 .. 1000;
type My_Arr is array (1 .. 10) of My_Integer;

procedure Bubble_Sort (A : in out My_Arr);
--# pre True;
--# post X(1) <= X(2) and X(2) <= X(3) and
--# X(3) <= X(4) and X(4) <= X(5) and X(5) <= X(6) and
--# X(6) <= X(7) and X(7) <= X(8) and X(8) <= X(9) and
--# X(9) <= X(10);

```

Figure 7: Specification of Bubble Sort Routine

The specification for the tomorrow routine is shown in Figure 8. Given a date including the day of the week the routine should return the date of tomorrow, accounting for leap years. However the implementation is incorrect in that it considers all years divisible by 4 as leap years (rather than just years divisible by 400 or 4, but not 100). Hence the desired test-case is one where the input date is 28th Feb on a year which is divisible by 100 but not by 400.

The following table shows the results of using simulated annealing to search for test-data that illustrates a specification failure. The search process was repeated 50 times to give an indication of the robustness of the search. Each search started using a randomly generated initial candidate solution and the following information was recorded; whether the test-case data was a solution (i.e. was a specification failure illustrated), the number of test-cases examined during the search and the execution time for the search⁵. The figures below show the average percentage of test-cases examined during each search, the average execution time for each search and also the total number of failure illustrating test-cases produced over the 50 search attempts.

Size of Param. Space	Space Searched	Valid Test-Cases	Execution Time
Middle 100000	0.0445%	50	5 sec.
Bubble Sort 1e+30	8.7e-11%	50	23 sec.
Tomorrow 286440	0.2467%	50	31 sec.

These results demonstrate the effectiveness of the optimisation-based approach to generating test-case data for specification failure testing. For example consider the tomorrow program, there are only 9 test-cases which would highlight the flaw in the given implementation. This represents only 0.00314% of the search space. Considering only the disjunct pair which encodes the flaw in the implementation simple random testing requires on average more than 31,000 iterations to illustrate the flaw. In comparison the simulated annealing search requires on average only 3,000 iterations which is a significant improvement. Given that this test-generation and execution is entirely automated it could be used as a pre-proof phase to attempt to detect errors before complex proofs are attempted.

6.2 Exception Condition Testing

⁵The system executes on an Intel Pentium Pro 200MHz machine running Linux and is coded in Ada 95.

```

type Day_Type is
  (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

type Date_Type is range 1..31;

type Month_Type is
  (Jan, Feb, Mar, Apr, May, Jun,
   Jul, Aug, Sep, Oct, Nov, Dec);

type Year_Type is range 1900..3000;

procedure Tomorrow
  (Day : in Day_Type;
   Date : in Date_Type;
   Month : in Month_Type;
   Year : in Year_Type;
   Next_Day : out Day_Type;
   Next_Date : out Date_Type;
   Next_Month : out Month_Type;
   Next_Year : out Year_Type);
--# pre (Date = 31 ->
--# (Month = Jan or Month = Mar or
--# Month = May or Month = Jul or
--# Month = Aug or Month = Oct or
--# Month = Dec)) and
--# (Month = Feb -> Date <= 29);
--# post (Day = Sun -> Next_Day = Mon) and
--# (Date /= 31 -> Next_Date = Succ(Date)) and
--# (Date = 31 ->
--# (Next_Date = 1 and
--# (Month = Dec ->
--# (Next_Month = Jan and
--# Next_Year = Year + 1)) and
--# (Month /= Dec ->
--# Next_Month = Succ(Month)))) and
--# (((Date = 30 and
--# (Month = Apr or Month = Jun or
--# Month = Sep or Month = Nov)) or
--# (Date = 29 and Month = Feb)) ->
--# (Next_Date = 1 and
--# Next_Month = Succ(Month) and
--# Next_Year = Year)) and
--# (((Date = 28 and Month = Feb and
--# ((Year mod 4 = 0 and Year mod 100 /= 0) or
--# Year mod 400 = 0)) ->
--# (Next_Date = 29 and
--# Next_Month = Month and
--# Next_Year = Year)) and
--# ((Date = 28 and Month = Feb and not
--# ((Year mod 4 = 0 and Year mod 100 /= 0) or
--# Year mod 400 = 0)) ->
--# (Next_Date = 1 and Next_Month = Mar and
--# Next_Year = Year)))));

```

Figure 8: Specification of Tomorrow Routine

This section presents the results of using simulated annealing to find test-case data which highlights exception conditions for two Ada programs – square and integer square root.

The square routine can be seen in Figure 9. The exception condition under examination occurs when the value of $N \times N$ is out of bounds for the type `Large_Integer`. Hence the desired test-case is where the value of $N \times N$ overflows its type range.

```

type Small_Integer is range -10001 .. 10001;
type Large_Integer is range -100000000 .. 100000000;

procedure Square
  (N : in Small_Integer; S : out Large_Integer) is
begin
  --# exception (N*N) < Large_Integer'First or
  --#           (N*N) > Large_Integer'Last;
  S := Large_Integer(N) * Large_Integer(N);
end Square;

```

Figure 9: An Exception Condition for Square Routine

The integer square root routine can be seen in Figure 10. The particular exception condition specified in this routine is where the assignment of $S + T$ to S is out of bounds for the type `My_Natural`. Hence the desired test-case is where the input parameter N causes $S + T$ to overflow its type range.

```

type My_Natural is Natural range 0 .. 10_000;

procedure IntRoot (N : My_Natural; Root : out My_Natural) is
  R : My_Natural := 0; S, T : My_Natural := 1;
begin
  loop
    exit when S > N;
    R := R + 1; T := T + 2;
    --# exception (S+T) < My_Natural'First or
    --#           (S+T) > My_Natural'Last;
    S := S + T;
  end loop;
  Root := (My_Natural (R));
end IntRoot;

```

Figure 10: An Exception Condition for Integer Square Root Routine

The results of the simulated annealing search for test data are shown below. The system was set to search for test-cases which illustrated the specified exception conditions 50 times. The figures below show the average percentage of parameter space that was searched, the average search time and also the number of test-case which caused the desired exception that were found.

The following table shows the results of using simulated annealing to search for test-data that illustrates an exception condition. The search process was repeated 50 times to give an indication of the robustness of the search. Each search started using a randomly generated initial candidate solution and the following information was recorded; whether the test-case data was a solution (i.e. was an exception condition caused), the number of test-cases examined during the search and the execution time for the search. The figures below show the average percentage of test-cases examined during each search, the average execution time for each search and also the

total number of exception generating test-cases produced over the 50 search attempts.

Size of Param. Space	Space Searched	Valid Test-Cases	Execution Time
Square 20002	0.265%	50	<0.5 sec.
Integer Square Root 10000	2.07%	50	<0.5 sec.

These results once again demonstrate the effectiveness of the optimisation based approach to generating test-case data for finding exception conditions. Again the process is entirely automatic requiring very little user effort to initiate the search for test-case data. The above examples are for illustration only. The aim is to show how optimisation can be used for exception generation testing. The general use of optimisation is intended to allow us to generate test-data where non-linearity is exhibited. In these examples a simple hill climbing search would have succeeded.

7 Further Work and Generalisations

The results presented above show that it is possible to use optimisation to automatically generate test-case data both effectively and efficiently. The degree of effort required to find test-case data for the problems presented above is significantly lower than that required by manual test-data generation or tool-supported proof attempts. Unsuccessful proof attempts are an extremely expensive way in which to find errors in software. Targeting manual testing at flaw finding in specifications or in run-time conditions is an very difficult task to manage will little guidance on the degree of completion and when to stop. This approach can be applied in a completely automated manner (obviously for specification failure testing effort is required to derive the specification information in a suitable form⁶) and thus could be applied almost for *free* before other testing begins.

To further assess the optimisation-based approach to provide a useful, generalised framework for automated test-case data generation more work on several fronts is needed. 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-cases can be generated.

7.1 Optimisation Techniques

There are 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 [8] and genetic algorithms [9] 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 (see [21]). 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 re-

⁶Note: While this paper presents the idea of finding flaws in full specifications, the approach is applicable to checking simple safety conditions or any other conditions which must hold that may be significantly cheaper to derive from the software requirements.

sults⁷. 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.

7.2 Search Efficiency

From the results presented in the previous section it can be seen that the search space (even for simple routines) can be extremely large. It can be seen that the current simple simulated annealing search is more efficient in its search when the parameter space is large (i.e. a smaller proportion of the parameter space is examined). Methods to reduce the amount of the search space examined may still be useful and indeed may allow very complex or higher-level routines to be processed. A number of problem specific enhancements can be made to the various optimisation techniques. For example, the neighbourhood of simulated annealing can be restricted to only allow variables that contribute to a condition failure to be changed. Careful selection of tabu-move attributes will also increase the efficiency of the search.

7.3 Generalisation

For each testing criteria to be addressed it is necessary to consider how to represent it as an optimisation problem. This paper shows an approach for representing specification failure and exception condition generation in an optimisation framework. These two problems are simply a special case of constraint solving involving a transformation of the variable values (i.e. the execution of the software under test). Thus the same objective function can be used to guide the search for general constraint solving. Indeed a tool using the objective function developed for general constraint solving has been integrated into a proof-tool being developed at York [4]. Preliminary work has also been carried out on testing for worst-case execution time (WCET) and stack-usage (see [21], where the objective function is simply a measure of execution time or stack size). There remains a large number of test-data selection criteria worthy of consideration. For each criterion it is only necessary to devise an objective function which gives the search process sufficient guidance. This allows virtually any testing criteria to be incorporated into the same generalised framework. In particular structural testing criteria and coverage will form the focus for work in the near future.

8 Conclusions

Many of the approaches for automated software test-data generation presented in the literature are inflexible or have limited capacity. They are often limited to particular data-types or control-flow structures and can often only process the lowest level routines.

Optimisation techniques are a flexible and powerful approach to solving *difficult* problems. To allow the optimisation technique to generate test-data for a new testing criteria it is necessary only to devise a suitable objective function. Because of this it is hoped that it will be possible to build a general framework which can be used to generate test-data for a wide class of testing criteria.

The results presented in this paper are encouraging and justify further work in the field. The ability to obtain test-data illustrating

⁷For example, if optimisation fails to find test-data illustrating a specification failure that does not imply that the software must be correct, it simply means that the search failed to find any such failures.

specification failures or exception conditions automatically could save a significant amount of time and money (time and money which would have been spent on unsuccessful proof attempts) in the development software, particularly safety-critical software where such formal specifications are readily available.

As with all testing approaches, we can only show the presence of faults not their absence. Indeed the failure of the search to find flaw illustrating test-cases does not indicate that the software is correct, only that the search failed. However, given an intensive directed search for flaws, the failure to find flaws does increase confidence in the quality of the software which is after all the aim of testing. The tools we provide for falsification based on heuristic optimisation 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 as producing test-cases that can be checked by other means. This is important since the algorithms are stochastic and it is difficult to reason about their efficacy for application to arbitrary code. We envisage them being used within a wider and integrated set of verification and validation tools.

9 Acknowledgements

This work was funded by grant GR/K63702 from the Engineering and Physical Sciences Research Council (EPSRC) as part of the Realising Our Potential Awards (ROPA) scheme in the UK. The authors would also like to express their thanks to the anonymous referees for their constructive and helpful comments.

10 Implementations

This section gives the incorrect implementations for each of the programs presented in the specification failure section. The point at which the implementation is incorrect is highlighted and it is this error which the search process should illustrate by the generation of suitable test-data.

10.1 Wrap-Round Counter

```
procedure Wrap_Inc (N : in out Integer) is
begin
  -- Error in line below should be >=
  if N > 10 then
    N := 0;
  else
    N := N + 1;
  end if;
end Wrap_Inc;
```

10.2 Bubble Sort

```
procedure Bubble_Sort(A : in out My_Arr) is
begin
  for I in 1 .. 8 loop -- Error loop should go to 9
    for J in 1 .. 9 loop
      if X(J) > X(J+1) then
        Swap (X(J), X(J+1));
      end if;
    end loop;
  end loop;
end Bubble_Sort;
```

10.3 Middle

```
function Middle (A, B, C : My_Integer) return My_Integer is
begin
  if (A < B and B < C) or else (C < B and B < A)
  then
    return B;
  elsif (A < C and C < B) or else (B < C and C < A)
  then
    return C;
  elsif (B < A and A < C) or else (C < A and C < B)
  then
    return A;
  else
    -- Error should return unique value
    -- when two values are the same
    return A;
  end if;
end Middle;
```

10.4 Tomorrow

```
procedure Tomorrow
  (Day : in Day_Type; Date : in Date_Type;
   Month : in Month_Type; Year : in Year_Type;
   Next_Day : out Day_Type; Next_Date : out Date_Type;
   Next_Month : out Month_Type; Next_Year : out Year_Type)
is
begin
  Next_Day := Day; Next_Date := Date;
  Next_Month := Month; Next_Year := Year;
  if Day = Sun then
    Next_Day := Mon;
  else
    Next_Day := Day_Type'Succ (Day);
  end if;

  if (Month = Dec and Date = 31) then
    Next_Date := 1; Next_Month := Jan;
    Next_Year := Year + 1;
  elsif (Date = 28 and Month = Feb) then
    -- Error - should be
    -- ((Year mod 4 = 0 and Year mod 100 /= 0)
    -- or Year mod 400 = 0)
    if ((Year mod 4 = 0) or Year mod 400 = 0)
    then
      Next_Date := 29;
    else
      Next_Date := 1;
      Next_Month := Mar;
    end if;
  elsif (Date = 31 or (Date = 29 and Month = Feb) or
  (Date = 30 and
  (Month = Apr or Month = Jun or Month = Sep or
  Month = Nov)))
  then
    Next_Date := 1;
    Next_Month := Month_Type'Succ(Month);
  else
    Next_Date := Date + 1;
  end if;
end Tomorrow;
```

References

- [1] E. H. L. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines*. Wiley, 1988.
- [2] J. Barnes. *High Integrity Ada: The SPARK Approach*. Addison-Wesley, 1997.
- [3] B. Beizer. *Software Testing Techniques*. Thomson Computer Press, 2nd edition, 1990.
- [4] J. Clark and N. Tracey. Solving constraints in law 22117. Law/d5.1.1(e), European Commission - DG III Industry, 1997. Legacy Assessment Workence Feasibility Assessment.
- [5] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, September 1976.
- [6] R. Demillo and A. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.
- [7] K. A. Dowsland. *Modern Heuristic Techniques for Combinatorial Problems*, chapter 2 – Simulated Annealing, pages 20–69. McGraw Hill, 1993.
- [8] F. Glover and M. Laguna. *Modern Heuristic Techniques for Combinatorial Problems*, chapter 3 – Tabu Search, pages 70–150. McGraw Hill, 1993.
- [9] D. E. Goldberg. *Genetic Algorithms in Search, Optimisation and Machine Learning*. Addison-Wesley, 1989.
- [10] B. Jones, H. Sthamer, and D. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.
- [11] S. Kirkpatrick, J. C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.
- [12] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.
- [13] B. Korel. Automated test data generation for programs with procedures. *ACM ISSA*, pages 209–215, 1996.
- [14] N. Metropolis, A. W. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculation by fast computing machine. *Journal of Chem. Phys.*, 21:1087–1091, 1953.
- [15] W. Miller and D. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, SE-2(3):223–226, September 1976.
- [16] A. J. Offutt and J. Pan. The dynamic domain reduction procedure for test data generation. <http://www.isse.gmu.edu/faculty/ofut/rsrch/atdg.html>, 1996.
- [17] M. Ould. Tesintg - a challenge to method and tool developers. *Software Engineering Journal*, 6(2):59–64, March 1991.
- [18] Praxis Critical Systems. *Spark-Ada Documentation 2.0*, 1995.
- [19] V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith, editors. *Modern Heuristic Search Methods*. Wiley, 1996.
- [20] N. Tracey, J. Clark, and K. Mander. The way forward for unifying dynamic test case generation: The optimisation-based approach. In *Dependable Computing and Its Applications – To appear*. IFIP, 1998.
- [21] N. J. Tracey. Test-case data generation using optimisation techniques – first year dphil report. Department of Computer Science, University of York, 1997.
- [22] X. Yang. The automatic generation of software test data from z specifications. Technical report, Department of Computer Studies, University of Glamorgan, 1995.