The Way Forward for Unifying Dynamic Test Case Generation: The Optimisation-Based Approach

Nigel Tracey John Clark Keith Mander
Department of Computer Science
University of York
Heslington, York
YO1 5DD, England.
+44 1904 432712
{njt,jac,mander}@cs.york.ac.uk

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. This framework can incorporate a number of testing criteria unifying both functional and non-function testing. Application of the optimisation based approach are given for worst-case execution time, specification conformance, structural coverage and exception condition testing. The results of a number of small example case studies are presented and show the efficacy of the approach.

1 Introduction

Software testing is an expensive process, typically consuming at least 50% of the total costs involved in developing software [1]. Automation of the testing process is desirable both to reduce the 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 becoming commercially wide spread, 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 advancing the state-of-the-art in software testing [18].

Software testing can be divided into the testing of functional and non-functional properties. Testing non-functional properties is concerned with ensuring that the constraints imposed by process, product or external requirements are met [20]. Research into testing non-functional requirements is limited, with very little work on automation of the testing process [21]. The current industrial state-of-practice appears to simply reuse functional tests (which in practice is little better than random testing). The work presented in this paper aims to provide a unified framework which allows test-data to be generated for both functional and non-functional properties.

Many techniques for generating test-data for functional properties have been developed [2, 4, 5, 6, 10, 11, 13, 14, 16, 17, 23, 24]. 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 comes from their use of symbolic execution. In contrast, dynamic methods require the software under test to be executed. Current dynamic methods are briefly discussed in the following section. This is followed by details of the optimisation based framework for dynamic test-data generation. Some applications of this framework to specific testing criteria are then given to show how optimisation techniques can effectively generate good test-data.

2 Dynamic Approach

The first use of dynamic methods to automatically generate test-data is presented in [16]. The use of dynamic methods is based on the contention that test-data generation is best formulated as a numerical maximisation (or minimisation) problem. The results of Miller's work appear to be very encouraging. Suggestions for further work included attempting to apply global heuristic optimisation techniques to improve the search process. Later work by Korel [13] presented a technique which monitored the execution of the software under test. When a undesirable branch was taken the execution was suspended and function minimisation techniques used to alter the execution path at the branch point. More recently, work by Jones [10] and Watkins [23] has begun to explore Miller's recommendation of using global optimisation techniques. A more detailed review of existing work on test data generation can be found in [9, 22].

The application of many of the existing techniques is limited by the lack of generality. This lack of generality can be seen in the limited data-types or control-flow structures which can be processed and also in the limited testing criteria addressed. The aim of the present work is to develop a unified framework for test-case data generation. The objectives are to automate the generation of test-data to satisfy *black-box* or *white-box* testing of functional properties and also non-functional properties. This paper focuses on four applications of this framework – worst-case execution time (WCET) testing, specification-conformance testing, exception condition testing and structural coverage.

3 Optimisation

General-purpose optimisation techniques make very few assumptions about the underlying problem which they are attempting to solve. It is this that we wish to exploit to allow a general test-data generation framework to be devised. Optimisation techniques are simply directed search methods which attempt to find minimal (or maximal) values of a particular objective function. Simulated annealing is one such optimisation technique. It has been used to assess the feasibility of an optimisation-based approach to generalise test-data generation during this work[1].

Simulated annealing is based on the idea of neighbourhood search [19]. Thirty years after the original algorithm was presented [15], Kirkpatrick suggested a form of simulated annealing could be used to solve complex optimisation problems [12]. Simulated annealing works by selecting candidate solutions which are in the neighbourhood of the given candidate solution. Better candidate solutions are always accepted, however worse candidate solutions are accepted in a controlled manner. The idea is that it may be 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 (often called the temperature, due to the analogy with the physical annealing process) is used to control the acceptance of inferior candidate solutions. At the start of a search 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.

Simulated annealing is a general purpose search strategy. Much of the algorithm remains completely unchanged across problem domains. All that is required to apply simulated annealing to a given domain is a representation of candidate solutions and neighbourhoods and an objective function which measure quality of candidate solutions. 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 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`	± Some proportion of allowed range
`FLOAT`	± Some proportion of allowed range
`BOOLEAN`	`TRUE` or `FALSE`
`ENUMERATION`	Any value from the enumeration

This framework can be used to address any testing problem where the quality of test-data can be quantified. Methods for quantifying the cost of a candidate solution depend entirely upon the software testing problem being addressed. These are detailed below for a number of software-testing problems.

4 Applications

4.1 Worst-Case Execution Time Testing

The problem of WCET testing is to find test-data which causes the worst-case path through the software-under-test. To allow this to be addressed within the optimisation framework a measure of actual dynamic execution time is used as the objective function. The optimisation, in particular simulated annealing, approach works as follows. An initial random set of input-data is generated and the software-under-test is executed and timed. The simulated annealing search then generates test-data in the neighbourhood of this initial solution and again executes and times the software-under-test. This solution will either be accepted or rejected (depending on the relative execution times and the current temperature of the simulated annealing algorithm), and the search continued using the (possibly) new test-data. This continues

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

until the simulated annealing search terminates, which occurs when none of the test-data generated for a number of iterations of the algorithm is accepted.

While testing can never give an absolute guarantee of WCET it does give a lower bound on the WCET. When combined with static analysis techniques (which give an upper bound) a confidence interval can be obtained, within which the WCET is guaranteed to fall. Testing and/or analysis can continue until this interval is sufficiently small. Some examples have been used to assess the ability to find good WCET test-case data using the optimisation framework. The following programs have been used in these tests.

Conditional Blocks – This software unit takes in two parameters – one integer and one enumeration. It has several conditional blocks, with many interdependencies between the decision taken. This demonstrates the system's ability to find valid test-cases in a situation where simple syntax-based analysis would fail.

Simple Loop – This is a simple software unit which has only one integer parameter. This unit iterates around a loop the number of times specified in the parameter. The expected result is the maximal value for the parameter type.

Binary Integer Square Root – This software unit calculates the integer square root of its input parameter using a binary search. Thus the expected result is a value which causes the maximum number of iterations in the binary search. In this case, with the input parameter range from 0 to 10,000, the maximum number of iterations is 14. This demonstrates the system's ability to cope with a still more complex cost surface.

Insertion Sort – This software unit performs an insertion sort on an array of 10 integer values. The worst case complexity for insertion sort is for reverse sorted data, thus the expected results is an ordered, decreasing array of values. This demonstrates the system's ability to cope with a large parameter/search space.

The results of the preliminary tests are shown in table 1. The system was set to search for the WCET of each of the above programs. The search was repeated 50 times for each program. For all test programs each search resulted in a valid test case which exercised a worst-case path.

Program	Parameter Space	WCET Space	Parameter Space Searched
Conditional Blocks	$1,407$	50	353.1%
Simple Loop	$10,000$	1	47.4%
Binary Square Root	$10,000$	321	52.4%
Insertion Sort	$9.765625e+16$	$1.027227e+10$	$7.198e-28\%$

Table 1: Preliminary WCET Results

As can be seen from the results the optimisation based search is more effective with larger, more complex, parameter spaces. With small parameter spaces to search, the optimisation based search examined a very large number of candidate solutions before the algorithm cooled and stabilised. The search for test-data took between 35 seconds and 3 minutes on an Intel Pentium Pro-200MHz.

4.2 Specification Conformance

A software unit's specification can be represented as a pre-condition constraint on input values and a post-condition constraint on input/output values. The most interesting test-case is one which shows an example of when the specification has not been conformed to (i.e. illustrates an error in the software-under-test). To allow this to be addressed within the optimisation framework the objective function must measure how close a test-case is to illustrating a specification falsification. This can be achieved as follows. Firstly the pre-condition and negated post-condition are converted to disjunctive normal form (DNF). All possible pairs of single pre-condition disjuncts and post-condition disjuncts are then formed using conjunction. Each pair can be considered as an encoding of one of the possible ways in which the software can fail. By generating input test-data and executing the software-under-test to calculate output data, the optimisation process uses these values to attempt to find a solution to each of the pairs in turn. Each term within the pair adds a value to the pair's overall cost according to the rules shown in Table 2 (the value K in the table refers to a positive 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 a stop-condition for the simulated annealing search. The search process will also stop once the algorithm has cooled sufficiently and is no longer making any progress.

Term	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` $(b - a) + K$

Table 2: Term Values

The following programs have been used to assess the effectiveness of optimisation to find specification failures.

Middle – The specification 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. In the case with three values the same this value should be returned. 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.

Bubble Sort – The specification states that given an array of input values the routine should sort them into ascending order. 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 array. Hence the desired test-case is one where the smallest value in the array is in the last element.

Tomorrow – The specification states that 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 specification of this program is shown in figure 1.

```
type Day_Name_Type is
  (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
type Day_Of_Month_Type is range 1..31;
type Month_Name_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_Name_Type; Date : in Day_Of_Month_Type;
   Month : in Month_Name_Type; Year : in Year_Type;
   Next_Day : out Day_Name_Type; Next_Date : out Day_Of_Month_Type;
   Next_Month : out Month_Name_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
--#    (Day /= Sun -> Next_Day = Succ(Day)) 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 1: Specification of Tomorrow Routine

The results of the preliminary tests are shown in table 3. The system was set to search for a specification failure of each of the above programs. The search was repeated 50 times for each program. For all test programs each search resulted in a valid test case which illustrated the error in the implementation of the software-under-test.

Program	Parameter Space	Parameter Space Giving Error	Parameter Space Searched
Middle	$1e+9$	$2e-6\%$	0.0445%
Bubble Sort	$1e+30$	9.95%	$8.7e-11\%$
Tomorrow	$286,440$	$2.79e-5\%$	0.2467%

Table 3: Preliminary WCET Results

4.3 Structural Coverage

The approach taken is to search for test-data which covers each statement in turn. Thus the objective function indicates whether the desired statement has been executed or not. Obviously any particular test-data will either satisfy this criterion or it will not. We therefore need the objective function to return good values for test-data which *nearly* executes the desired statement and worse values for those cases which are a long way from executing the desired statement. The objective function used involves instrumenting the software under test with procedure calls. These procedures monitor the execution path taken and each contribute to the total value of the objective function. The instrumentation works as follows. A call to `covered_node(x)` is made at the start of each *basic block*[2]. This monitors which basic blocks are executed by a given test, to allow the system to decide which basic blocks are uncovered and thus where the subsequent search should be targeted. The calls to `cost_node_x` routines are placed at each point where there is a decision statement (i.e. if statements, loop statements, etc ...). These functions add to the objective function a value which is related to how close the current execution is to taking the desired branch at the given decision. This is calculated in the same manner as measuring how close to a specification failure a test-case is.

If the condition is true then the constraint is satisfied, however a small value is still added to the objective function to bias the search towards boundary values. The `SF` constant makes sure that this value is sufficiently small so that the search is biased towards finding test-data that satisfies the constraint before finding boundary values. If the condition is not meet then the constraint is not satisfied in this case a failure constant, `K`, and a measure of how *untrue* the condition is are added to the objective function. A simple example will help illustrate how this works in practice. Figure 2 shows a simple integer division routine instrumented for statement coverage. In this example the loop is instrumented such that it requires three sets of test-data to cover it completely; zero times, once and two or more times round the loop are considered separate tests (note: the instrumentation lines appear as comments simple so that they can be easily identified, obviously they are normally inserted as uncommented code).

The call to `coverage.cost_node_2`, for example, works as follows. If the search is attempting to find test-data to cover basic-block 2 then the following value is added to the cost.

$$\texttt{if}(numerator - 0) \leq 0 \texttt{ then } \frac{abs(a-b)}{SF} \texttt{ else } (Numerator - 0) + K$$

Assuming we are attempting to find test-data to cover basic-block 2 and we try a test case with `Numerator` set to 50 then this adds $50 - 0 + K$ to the objective function. However, if `Numerator` is set to 1 then this adds only $1 - 0 + K$ to the objective function (i.e. this test-case is *closer* to covering

[2]A basic block is a set of statements such that if any one statement of the set is executed then they all are.

```
subtype My_Integer is Integer range 0 .. 1000;
procedure Remainder
  (Numerator      : in Integer; Denominator : in Integer;
   Quotient_Val   : out My_Integer; Remainder_Val : out My_Integer)
is
   --  No_Its : Integer;
begin
   --  Coverage.Covered_Node (1);
   Quotient_Val := 0;
   Remainder_Val := Numerator;

   --  Coverage.Cost_Node_2 (Numerator);
   --  Coverage.Cost_Others_A (Numerator);
   if Numerator <= 0 then
      --  Coverage.Covered_Node (2);
      Quotient_Val := 0;
      Remainder_Val := 0;
      return;
   end if;

   --  Coverage.Cost_Node_3 (Denominator);
   --  Coverage.Cost_Others_B (Denominator);
   if Denominator <= 0 then
      --  Coverage.Covered_Node (3);
      Quotient_Val := 0;
      Remainder_Val := 0;
      return;
   end if;

   --  No_Its := 0;
   --  Coverage.Cost_Node_4 (Remainder_Val, Denominator, No_Its);
   --  Coverage.Cost_Node_5 (Remainder_Val, Denominator, No_Its);
   --  Coverage.Cost_Node_6 (Remainder_Val, Denominator, No_Its);
   while Remainder_Val >= Denominator loop
      Quotient_Val := Quotient_Val + 1;
      Remainder_Val := Remainder_Val - Denominator;

      --  No_Its := No_Its + 1;
      --  Coverage.Cost_Node_4 (Remainder_Val, Denominator, No_Its);
      --  Coverage.Cost_Node_5 (Remainder_Val, Denominator, No_Its);
      --  Coverage.Cost_Node_6 (Remainder_Val, Denominator, No_Its);
   end loop;
   --  Coverage.Covered_Node (4, No_Its);
end Remainder;
```

Figure 2: Instrumented Division Routine for Coverage

basic-block 2). Similarly with `Numerator` set to -50 then $\frac{abs(50-0)}{SF}$ is added to the objective function. However, if `Numerator` is set to 0 then this adds only $\frac{abs(0-0)}{SF}$ to the objective function (i.e. this test-case is *closer* to the boundary). Even though both these last two cases cover the basic-block the one closer to the boundary is biased so it will be selected by the optimisation search.

The search proceeds looking for test-data to cover each basic-block in turn. When test-data is found which covers the desired basic-block the system monitors which other basic-blocks (if any) the test-data also covered. The search then moves on to look for test-data for the next uncovered basic-block. The following table shows the test-data set return by the system for this routine.

Basic Blocks Covered	Test-Data
1, 6	N = 47, D = 21
1, 2	N = 0, D = -100
1, 3	N = 1, D = 0
1, 4	N = 81, D = 82
1, 5	N = 3, D = 2

4.4 Safeness/Exception Conditions and Assertion Check

Safeness conditions represent constraints on program variables which must hold to maintain safe operation of the system in which the software is embedded. Exception conditions are assertions which must hold to ensure no exception is raised (for example, numeric overflow or type-bound errors). Assertion checks are constraints which the developer has specified must hold at a particular point in the code. Targeting testing at causing specific exceptions may be very important. In many production systems it is assumed that exceptions will not occur (confidence in this is achieved often through reuse of functional tests). Thus in the released system run-time checks are often disabled to gain a performance increase. Specific testing for exception conditions may allow increased confidence that unwanted exception conditions will not occur. Using a similar objective function as above for specification conformance it is possible to search for test-cases which illustrate a unsafe or exception condition or an assertion failure. The objective function must guide the search to the statement of interest in the software under test, this can be achieved using the method describe for structural coverage. However, the objective function must also guide the search to find test-data which causes the exception condition. This can be achieve by combining the objective introduced for specification conformance.

Preliminary assessment has shown this technique to be effective in generating test-cases to cause constraint exceptions in a number of numeric routines. With parameter spaces of around $10,000$ and with only a handful of values which cause the exception condition the system has consistently managed to find test-data to illustrate the exceptions by examining only 5% of the search space. However, further work on larger-scale realistic samples is required to fully assess the effectiveness of the system in this area.

5 Further Work

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, given that it is almost completely automated. However, 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.

5.1 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 [7] and genetic algorithms [8] 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 [22]). 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[3]. 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.

5.2 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). Indeed the search time for the case-studies presented ranges from 3 seconds to 6 minutes on a Pentium-Pro based machine, which is far quicker than the test-data could have been manually generated. 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.

5.3 Generalisation

For each testing criterion to be addressed it is necessary to consider how to represent it as an optimisation problem. This paper shows an approach for representing a number of testing problems in an optimisation framework. The specification problems is 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 [3]. 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.

6 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.

[3]For example, if optimisation fails to find test-data illustrating a specification failure that does no imply that the software must be correct, it simply means that the search failed to find any such failures.

The results presented in this paper are extremely encouraging and justify further work in the field. The ability to obtain test-data illustrating 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 more likely to be available. WCET testing and covering test-sets would also save money allowing resources to be used in other verification and validation efforts.

As with general dynamic test approaches, we can only show the presence of faults not their absence. However, 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.

7 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, and also grant GR/L42872 from the EPSRC as part of the CONVERSE project.

References

[1] B. Beizer. *Software Testing Techniques*. Thomson Computer Press, 2nd edition, 1990.

[2] R. Boyer, B. Elspas, and K. Levitt. Select – a formal system for testing and debugging programs by symbolic execution. *Preceedings International Conference of Reliable Software*, pages 234–245, 1975.

[3] J. Clark and N. Tracey. Solving constraints in law 22117. Law/d5.1.1(annex e), European Commission - DG III Industry, 1997. Legacey Assessment Worbence Feasibility Assessment - Final Deliverable.

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

[5] R. Demillo and A. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[6] 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.

[7] F. Glover and M. Laguna. *Modern Heuristic Techniques for Combinatorial Problems*, chapter 3 – Tabu Search, pages 70–150. McGraw Hill, 1993.

[8] D. E. Goldberg. *Genetic Algorithms in Search, Optimisation and Machine Learing*. Addison-Wesley, 1989.

[9] D. Ince. The automatic generation of test data. *Computer Journal*, 30(1):63–69, 1987.

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

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

[12] S. Kirkpatrick, J. C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

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

[14] B. Korel. Automated test data generation for programs with procedures. *ACM ISSTA*, pages 209–215, 1996.

[15] 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.

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

[17] 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.

[18] M. Ould. Tesintg - a challenge to method and tool developers. *Software Engineering Journal*, 6(2):59–64, March 1991.

[19] V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith, editors. *Modern Heuristic Search Methods*. Wiley, 1996.

[20] I. Sommerville. *Software Engineering*. Addison-Wesley, forth edition edition, 1992.

[21] E. H. Spafford. Extending mutation testing to find environmental bugs. *Software – Practice and Experience*, 20(2):181–189, February 1990.

[22] N. J. Tracey. Test-case data generation using optimisation techniques – first year dphil report. Department of Computer Science, University of York, 1997.

[23] A. L. Watkins. The automatic generation of test data using genetic algorithms. *Proceedings of the 4th Software Quality Conference*, 2:300–309, 1995.

[24] X. Yang. The automatic generation of software test data from z specifications. Technical report, Department of Computer Studies, University of Glamorgan, 1995.