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. Tel : +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 [2]. 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. Ould suggested that automation of test-data generation is vital to advance the state-of-the-art is software testing [9]. 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 [2].

Automated test-data generators for structural testing can be divided into three classes – random [2], static [3, 4] and dynamic. This work is based on the dynamic approach to test-data generation. This approach was first suggested in 1976 [8]. Korel's work [7] built on this, suggesting as a way forward the use of global optimisation techniques. More recently, some work has investigated the use of global-optimisation [13, 5] to overcome the problem of locally optimal solutions in the search space.

The research presented in this paper builds on these approaches. An optimisation-based framework has been developed and applied to a number of testing problems, such as specification conformance testing and worst-case execution time testing [10, 11]. This paper focuses on the application of the framework to a number of structural-testing problems.

2 Optimisation-Based Structural Test-Data Generation

In essence, the test-data generation problem for structural testing is finding a set of program inputs that achieves the desired coverage. The dynamic approach to this problem involves a search for program input which forces execution of the desired part of the software under test (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. 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.

The input domain of most programs 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. The size and complexity of the search space therefore limits the effectiveness of simple gradient-descent or neighbourhood searches as they are likely to get stuck in locally optimal solutions and hence fail to find the desired test-data.

Heuristic-global optimisation techniques are designed to find good approximations to the optimal solution in large complex search spaces. Simulated annealing is one such general purpose optimisation technique [6]. 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.

It is necessary to define the cost-function which will guide the simulated annealing search. The cost-function needs to indicate whether the desired statement (or branch, etc.) has been executed. 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.

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. This property gives an efficient stopping criteria for the search process. The cost-function is calculated as shown in table 1.

In order to evaluate the cost-function it is necessary to execute an instrumented version of the SUT. There are two types of procedure call added by the instrumenter – execution path monitor calls and branch evaluation calls. The monitor 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 branch evaluation calls replace the branch predicates in the SUT. They 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:

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 (cost (`a`), cost (`b`))`
$a \wedge b$	`cost (`a`) + cost (`b`)`
$\neg a$	Negation is moved inwards and propagated over a

Table 1: Cost-Function Calculation

- 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 branch predicate must be false then the cost of $\neg$ (branch predicate) is used.
- For loop predicates the desired number of iterations determines whether the cost of the loop predicate or $\neg$ (loop predicate) is used.
- 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.

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.

3 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, [10, 11].

3.1 Boundary Value Analysis

Boundary value analysis considers test-data which lies close the boundaries of a sub-domain to be of a higher adequacy than test-data which is further away.

To address the problems of boundary value analysis 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 example of $X < 42$, if the desired path requires this condition to be false, the test-data generator 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 reduced by a scaling factor to allow the cost of $X \geq 42$ to dominate. This removes the efficient stopping criterion defined earlier. This simply means that the search will continue until it freezes, at which point it will terminate.

For more complex branch predicates, the particular boundary may be more difficult to identify. The current tool-set will automatically search for 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.

3.2 Assertion/Run-Time Exception Testing

Assertions and run-time exceptions are a useful tool for automatic detection of run-time errors. 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.

To allow improved guidance the assertion or exception-condition is converted to disjunctive normal form (DNF). 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.

3.3 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, i.e. for all input-data in the component's domain the pre-condition holds true. 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 component is instrumented such that it evaluates the cost of meeting a given disjunct in the negated pre-condition. This value is added to the overall cost. Thus the search is guided to test-data which executes the call of the component and also breaks the pre-condition of the component.

4 Evaluation

A preliminary evaluation has used a collection of small programs written in Ada 95 (20 to 200 lines of code). In all but one case 100% branch coverage was achieved with search times of 2 to 35 seconds. In the failing case 48 of the 50 attempts gave 100% coverage. Tuning the search parameters to slow down the cooling overcame this problem. A number of the programs contained breakable assertions or could cause run-time exceptions. The simulated annealing search found test-data in call cases which was able to highlight the error. The test-data generation took between 0.5 and 17 seconds. Component re-use testing has been evaluated using a small number of programs with explicit pre-conditions specified as Spark Ada proof-context annotations [1]. Harness software was produced which used these components in an unsafe manner which for some inputs. For each of these simple examples suitable test-data was generated to highlight the unsafe reuse. As a comparison random test-data generaton has been used on the same problems and was found to be up to 4 orders of magnitude less efficient.

5 Conclusions

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. We view the tools 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. More details and full results can be found in the full version of this paper (located at http://www.cs.york.ac.uk/testsig in the publications section)

5.1 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 most important area requiring further work is that of gathering empirical evidence as to the effectiveness of the technique.

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 and genetic algorithms 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 [12].

6 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. *High Integrity Ada: The SPARK Approach*. Addison-Wesley, 1997.

[2] B. Beizer. *Software System Testing and Quality Assurance*. Thomson Computer Press, 1996.

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

[4] 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.

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

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

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

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

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

[10] 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.

[11] 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.

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

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