Automatic Test Data Generation for Multiple Condition and MCDC Coverage

Kamran Ghani and John A. Clark
Department of Computer Science
University of York, York
YO10 5DD, UK
{kamran, jac}@cs.york.ac.uk

Abstract

Recently Search Based Software Engineering (SBSE) has evolved as a major research field in the software engineering community. SBSE has been applied successfully to many software engineering activities ranging from requirement engineering to software maintenance and quality assessment. One area where SBSE has seen much application is test data generation. Search based test data generation techniques have been applied to automatically generate data for testing functional and non-functional properties of softwares. For structural testing, most of the time, the criterion used, is branch coverage. However, this is not enough. For the wider acceptance of search based test data generation techniques, much stronger criteria are needed. In this paper we have proposed an automatic framework that extend search based testing techniques to more stronger criteria such as multiple condition and MCDC coverage.

1. Introduction

1.1. Dynamic Testing

Dynamic testing — "the dynamic verification of the behaviour of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behaviour" [1] — is used to gain confidence in almost all developed software. Various static approaches can be used to gain further confidence but it is generally felt that only dynamic testing can provide confidence in the correct functioning of the software in its intended environment.

We cannot perform exhaustive testing because the domain of program inputs is usually too large and also there are too many possible execution paths. Therefore, the software is tested using a suitably selected set of test cases. A variety of coverage criteria have been proposed to assess how effective test sets are likely to be. Historically, criteria exercising aspects of control flow, such as statement and branch coverage [2], have been the most common. Further criteria, such as data flow [3], or else more sophisticated flow criteria such as MC/DC coverage [4] have been adopted for specific application domains. Many of these criteria are motivated by general principles (e.g., you cannot have much confidence in the correctness of a statement without exercising it); others target specific commonly occurring fault types (e.g., boundary value coverage).

Finding a set of test data to achieve identified coverage criteria is typically a labour-intensive activity consuming a good part of the resources of the software development process. Automation of this process can greatly reduce the cost of testing and hence the overall cost of the system. Many automated test data generation techniques have been proposed by researchers. We can broadly classify these techniques into three categories: random, static and dynamic [5] [6].

Random approaches, though simple but are not very efficient and may simply fail to generate test data in any reasonable time-frame for more complex software (or more sophisticated criteria). Static approaches, on the other hand still struggle with some important code elements such as loops, pointers and arrays. Dynamic approaches, to a great extent, overcome these limitations. One such approach that has become more popular recently is the use of search techniques to generate test data. It has been briefly described in the following section.

1.2. Search Based Test Data Generation

In search based test data generation (SBTDG) achieving a test requirement is modeled as a numerical function optimisation problem and some heuristic is used to solve it. The techniques typically rely on the provision of "guidance" to the search process via feedback from program executions. For example, suppose we seek test data to satisfy the condition $X <= 20$. We can associate with this predicate a cost that measures how close we are to satisfying it, e.g., $cost(X <= 20) = max(X - 20, 0)$. The value $X == 25$ clearly comes closer to satisfying the condition than does $X == 50$, and this is reflected in the lower cost value associated with the former. The problem can be seen as minimising the cost function $cost(X <= 20)$ over the range of possible values of X.

SBTDG for functional testing generally employs a search/ optimisation technique with the aim of causing assertions

at one or more points in the program to be satisfied. We may require each of a succession of branch predicates to be satisfied (or not) to achieve an identified execution path; we may require the program preconditions to be satisfied but the postconditions to be falsified (i.e., falsification testing — finding test data that breaks the specification) [7]; or else we may simply require a proposed invariant to be falsified (e.g., breaking some safety condition, or causing a function to be exercised outside its precondition) [8].

There has been a growing interest in such techniques and we see more and more applications of these techniques to software testing. To evaluate the applicability of such approaches many tools were also developed in the past. For example, TESTGEN [9], [10], QUEST [11], ADTEST [12], GADGET [5], tools by Jones et al. [13] and Tracey [14] and recently IGUANA [15]. However, in most of these tools the target coverage criterion is branch coverage. We know that this criterion is not very 'strong'. To have more confidence in the testing process and hence wide acceptance of the search approaches, more specifically in the critical system domain, we need the application of search techniques to stronger criteria. In this paper we propose the application of search techniques to generate test data, to achieve multiple condition and MC/DC coverage.

1.3. Optimisation Techniques

Some of the optimisation techniques that have been successfully applied to test data generation are Hill Climbing, Simulated Annealing (SA), Genetic Algorithms (GAs), Tabu Search (TS), Ant Colony Optimisation (ACO), Artificial Immune Systems (AIS), Estimation of Distribution Algorithms (EDAs), Scatter Search (SS) and Evolutionary strategies (ESs). Our framework can incorporate any of these techniques. We evaluate our approach using Simulated Annealing, one of the most widely used optimisation techniques for search based testing. In the following paragraphs we give a brief introduction to this technique.

Simulated annealing is a global optimization heuristic that is based on the local descent search strategy. The original algorithm was given by Metropolis et al in 1953 [16]. It was 30 years later that Kirkpatrick [17] suggested its application to optimisation problems. Since then SA has been applied in many forms successfully to solve many problems more efficiently. Our Framework uses the standard SA as shown in Table 1.

The technique itself, as the name suggests, is inspired by the annealing process in metals, where the variations in temperature affect their mechanical properties. In analogy, each move in the search process, which is a random solution in the neighbourhood of current solution, is accepted with a probability depending on a parameter called 'Temperature'. Initially the temperature is high and almost all moves are accepted. Gradually the temperature is lowered. Improving moves are always accepted. Non-improving moves are accepted probabilistically. This facilitates escape form local optima. The worse a move is the less likely it is to be accepted. Similarly the lower the temperature the less likely a worsening move will be accepted. As the temperate heads to zero the search effectively becomes a local hill-climb.

Select an initial solution $testData_0$;
Select an initial temperature $t_0 > 0$;
Select a temperature reduction function α (=0.9 here);
Repeat

Repeat

Generate a move $testData \in N(testData_0)$;

Where $N(testData_0)$ defines the neighbourhood of $testData_0$

$\delta = f(testData) - f(testData_0)$;

If $\delta < 0$

Then $testData_0 = testData$;

Else

Generate random x uniformly in the range (0, 1);

If $x < e^{-\delta/t}$ then $testData_0$ = testData;

Until $innerLpCount = maxInnerLpNo$ or $f(testData_0)$satisfies the requirement;

Set t = $\alpha(t)$;

Until $outerLpCount = maxOuterLpNo$ or $nonAcceptCount = maxNonAcceptNo$ or $f(testData_0)$ satisfies the requirement.

$testData_0$ is the desired test-data
if $f(testData_0)$ satisfies the requirement.

Table 1. Standard SA algorithm

1.4. Control Flow Coverage Criteria

The most common control flow coverage criteria are statement, branch or decision coverage, condition/decision coverage, MCDC and multiple condition coverage. We restrict ourselves here to the discussion of multiple condition and MCDC, the subject of this paper.

In the control flow based coverage criteria subsumption hierarchy as shown in Figure 1 [18], multiple condition coverage is the strongest criterion. It requires test cases that cover all the conditions in a decision. For example, consider the truth table in Table 2. For a decision containing two conditions as $C_1 \wedge C_2$, we need test cases to exercise all

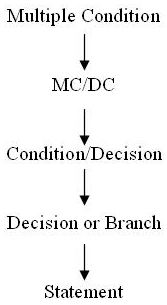

Figure 1. Control Flow Subsumption Hierarchy

Test Case No	$C_1 \wedge C_2$	outcome
1	TT	T
2	TF	F
3	FT	F
4	FF	F

Table 2. Test Case Sequences for Multiple Condition Coverage.

'true' and 'false' combination of C_1 and C_2 i.e., TT, TF, FT, and FF. In general, if a $decision\ D$ contains n number of $conditions\ C$, we require at least 2^n test cases to satisfy multiple condition coverage.

Since the number of tests required to satisfy multiple decision coverage increases exponentially with the number of conditions, it can become very expensive for decisions with large number of conditions. There may also be infeasible combinations of conditions. Filtering out such combinations further increases the cost of this criterion and hence it may not be practical to apply it for large and complex systems.

MC/DC on the other hand is a more practical criterion and hence usually a testing requirement for critical systems such as those developed in the avionics domain. It is satisfied when (i) every condition in a decision in a program has taken all possible outcomes at least once, and (ii) each condition has been shown to independently affect the decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions [4]. Consider Table 3; requirement (i) is satisfied by test cases 2 and 3 for condition C_1 and by test cases 1 and 2 for condition C_2. Requirement (ii) is satisfied by test cases 1 and 2 for C_1 and by 1 and 3 for C_2. We can satisfy MCDC for a decision with a minimal set of $n+1$ test cases, where n is the number of conditions in the decision.

Test case No	$C_1 \wedge C_2$	outcome
1	TT	T
2	TF	F
3	FT	F

Table 3. Test Case Sequence for MCDC.

2. Framework for Refined Test Data Generation

This section describes how search based techniques have been applied to generate test data for multiple condition coverage and MCDC. We propose a framework. Figure 2 shows the high level architecture of the framework. The framework can be broadly classified into three parts: the Instrumentor; Fitness Function Calculator; and Optimisation rig.

The Instrumentor takes the class under test (CUT) as input and produces an instrumented version. Paths and branches, for which data are to be generated, are identified. Based on coverage criteria, we select a fitness function and then using the selected optimisation technique, we search for the desired data.

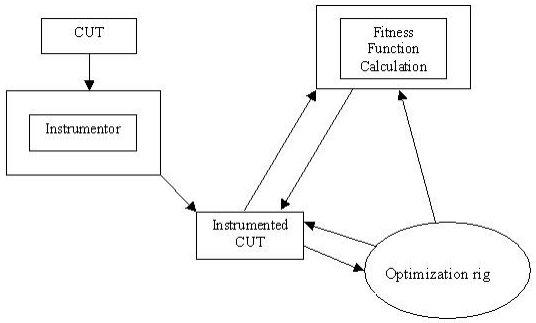

Figure 2. High Level Architecture of the Framework

The Instrumentor is an important component of the framework. We used ANTLR [19] for Java parser generation using the Java grammar [20]. The Abstract Syntax Tree (AST) is generated, walked and instrumented at desired locations in the code. We used the Java emitter by [21], which walks the modified AST to generate the instrumented code. The instrumentation scheme is devised in such away to capture the required information preserving the semantics of the program. The following lines explain the instrumentation scheme. Our tools assume that all conditions are in disjunctive normal form (DNF).

If during parsing a control structure of the form

$$c_{00} \wedge c_{01}..c_{0n} \vee c_{10} \wedge c_{11}..c_{1n} \vee ..c_{n0} \wedge c_{n1}..c_{nn}$$

is encountered, the instrumentor annotates the code with the respective decision number, conjunct number and clause number.

Here

c_{ij} represents jth conjunct of ith clause of the form

$(expr)relop(expr)$.

$relop$ can be any of the relational operators: $\leq, \geq, <, >, ==, !=$

For example consider the following if-branch, the condition portion of which consists of two clauses and each clause consists of two conjuncts

$$if((x_1 < 15 \wedge x_2 > 10) \vee (y_1 > z_1 \wedge z_1 < 35))$$

The instrumentor replaces it by the following structure.

$$if(data.complex(dec\#,$$
$$data.basic(x_1, " < ", 15, dec\#, clause_0, conjunct_0) \wedge$$
$$data.basic(x_2, " > ", 10, dec\#, clause_0, conjunt_1) \vee$$
$$data.basic(y_1, " < ", z_1, dec\#, clause_1, conjunct_0) \wedge$$
$$data.basic(z_1, " < ", 35, dec\#, clause_1, conjunct_0)))$$

The instrumented code can then be compiled by any standard Java compiler. $data.basic()$ and $data.complex()$ are method calls to the object of a Data class which contains data structures to gather information from all instrumented branches during execution of program.

The advantage of this kind of instrumentation is that we can store the information about input decision variables as well as the decision itself separately. This help us to manipulate it in a flexible way for our purpose i.e., we can incorporate different cost functions and hence coverage criteria. One problem that usually occurs with traditional source code instrumentation is that of undesirable side effects, which can alter the program's intended behaviour during run time. However, the suggested instrumentation scheme also avoids this problem.

3. Cost Function

The proposed framework can use any of the cost functions for test data generation, however, to start with, the cost function that we chose is similar to the one proposed by Tracey et al [14]. The cost function has been shown in Table 4. This is a modified form of the work proposed by Korel [9]. The cost function is based on evaluating branch predicates. It gives a value of 0 if the branch predicate evaluates to the desired value and a positive value otherwise. The lower the value, the better is the solution.

The framework can calculate cost function for many different criteria. Currently it provides cost calculation for:

- For any decision it reaches.
- For any conjunct in the decision.
- For a series of specific conjunct assignments for each decision and
- For any identified series of branches (a path).

In evolutionary testing literature, as stated earlier, we usually find test data generation for branch coverage only. However, The ability of our framework to calculate cost functions for a series of specific conjunct assignment in a flexible way helped us to achieve stronger coverage criteria such as MCDC and decision/condition coverage. The framework automatically generates test case sequences for multiple condition coverage and MCDC. In the case of MCDC, we generate minimal set of test case sequences, using the technique described by Mathure [22]. We modified the technique slightly to generate test sequences for compound decisions containing both conjuncts and disjuncts.

We have chosen the approach level with branch distance strategy to reach the desired goal [15]. In this strategy, the cost function is devised in such a way to consider the path leading to the target. Thus the cost function is given as;

$$CurrentBranchCost+$$
$$K * NumberOfRemainingBranches$$

Where K is constant and can be given any value just greater than the highest branch cost of any of the branches. The advantage of this value is that, it helps the search to be guided for a complete path.

When the current branch is the one directly leading to the the target without any other branches in between then the $BranchCost$ is the summation of costs of individual conditions in the MCDC test case under consideration. For all other branches, leading to this branch, only the cost for branch coverage is calculated. e.g., Consider the following branch statement;

$$if\ (a == b) \wedge (b == c)$$
{
statement/s;
target reached
}

The Minimal MCDC sequence require three test cases i.e., TT, TF and FT. Consider the test case under consideration is TT. Let the current values of variables be $a = 551$, $b = 382$ and $c = 168$, then the cost of the current branch is calculated as $abs(551 - 382) + abs(382 - 168)$.

During the execution of instrumented program, the framework also stores the minimum and maximum value of cost function for a condition. This is particularly helpful for storing information about loop predicates as well as the branches inside a loop. For example, consider the following code segment;

```
public void loopTest(int x){
    for (int i=0;i<10; i++){
      if (x>50){

       target statement;
      }
      else{
```

```
            statements;
        }
      x++;
    }
  }
```

The target is executed only when the value of x is greater than 40. When the value is less than 40, the minimum cost value help us to guide the search to select a more closer value until the goal is reached. If a false outcome of the branch to be taken, then any value of x less than 50 will reach the goal. However, if a value of 49 is taken, after the loop is executed, the value of x will be 59. If we do cost function calculation after the loop is executed, we wont be given a solution until the value of x less than 40 is chosen. However, by keeping the minimum and maximum values, the framework avoids this problem.

Table 4. Cost-Functions

Element	Value#1
$a = b$	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 $(b-a) \geq 0$ then 0 else $(b-a)$+K
$a \vee b$	$min(cost(a), cost(b))$
$a \wedge b$	$cost(a) + cost(b)$

The framework can be used with different search based optimization techniques. Currently the framework is implemented with Simulated Annealing and integrated with genetic algorithm through an open source API JGAP[23]. However in this work we have restricted our experimentation with SA only.

4. Experimentation

We performed experiments with small programs which are taken mostly from current software testing literature. Triangle is a benchmark program and has been used many times in many works related to software testing. We used a variant of this program that contains more complex branching structures. CalDate is taken from May's PhD thesis [24]. This program converts the specified date (given as three arguments: day, month,year) into a Julian date. The Quadratic program determines the root of a quadratic equation. Complex is a custom written program with difficult branching structures. Expint, Exponential Integral Program, is taken from [25]. The program is translated in Java to be used with our tool.

We used the following parameters for Simulated Annealing. The parameter were optimized through a small set of experiments.

Move strategy: fixed

Geometric Temperature decrease rate: 0.9

Number of iteration in inner loop: 100

Maximum number of iteratoin in outer loop: 500

Stopping criteria: Either a solution is found or maximum number of iterations are reached.

Table 5 summarises the results for Multiple Condition Coverage whereas Table 6 summarises the result for MCDC. The results are averaged over thirty runs of each program. Column three gives the average number of execution taken to search input data for all branches. Success rate calculated by dividing the number of times the data was found to the total number of runs. Coverage is calculated by dividing the number of combinations covered to the total number of feasible combinations.

The search tool successfully obtained hundred percent coverage in all programs but one. In case of Expint, it achieved coverage for both criteria in 21 branches. A further identified branch for which data could not be found was actually unachievable.

5. Conclusion and Future Work

In this paper we demonstrated how search based test data techniques can be used to generate test data at the lowest level of branch predicate. To the best of our knowledge, this is the first work of its kind. The generation of test data at the lowest level enabled us to target more practical coverage criteria such as MCDC. We believe that this work is important for the wide acceptance of SBTDG techniques, especially at industrial level.

We developed a flexible prototype tool. The tool can be further extended to include other search based algorithms. This is particularly useful for comparative studies of these techniques.

References

[1] IEEE, *Guide to software engineering body of knowledge*, 2004, pp. 5–1.

[2] G. Myers, *The Art of Software Testing*. John Wiley & Sons, 1979.

[3] S. Rapps and E. Weyuker, "Selecting software test data using data flow information," *IEEE Transactions of Software Engineering*, vol. 11, no. 4, pp. 367– 375, April 1985.

[4] "Software considerations in airborne systems and equipment certification," RTCA/DO-178B, pp. 2455–2464, December 1992, rTCA Inc.

Porgram	No Of Branches	Average No of Execution	Success Rate (%)	Coverage (%)	Average Time of Execution in Sec
Triangle	8	85306	99.99	100	353.0663
CalDate	6	25196	99.91	100	38.03
Quadratic	6	6256	99.94	100	16.46
Complex	6	91587	100	100	128.3
Expint	22	92253	99.5	96	630

Table 5. Test Data Generation for Multiple Condition Coverage

Porgram	No Of Branches	Average No of Execution	Success Rate (%)	Coverage(%)	Average Time of Execution in Sec
Triangle	8	28198	99.98	100	37.97
CalDate	6	29856	99.91	100	39.1
Quadratic	6	10370	99.94	100	16.937
Complex	6	49813	99.97	100	54.1
Expint	22	45925	100	96	130.7

Table 6. Test Data Generation for MC/DC

[5] C. C. Michael, G. McGraw, M. Schatz, and C. C. Walton, "Genetic algorithms for dynamic test data generation," in *Automated Software Engineering*, 1997, pp. 307–308. [Online]. Available: citeseer.ist.psu.edu/michael97genetic.html

[6] E. Diaz, J. Tuya, and R. Blanco, "Automated software testing using a metaheuristic technique based on tabu search," in *Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, Oct. 2003, pp. 310 – 313. [Online]. Available: citeseer.ist.psu.edu/diaz03automated.html

[7] N. Tracey, J. Clark, and K. Mander, "Automated program flaw finding using simulated annealing'," in *proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, Clearwater Beach, Florida, United States, 1998, pp. 73 – 81.

[8] N. J. Tracey, "A search-based automated test-data generation framework for safety-critical softwares," DPhil, University of York, 2000.

[9] B. Korel, "Automated software test data generation." *IEEE Trans. Software Eng.*, vol. 16, no. 8, pp. 870–879, 1990.

[10] R. Ferguson and B. Korel, "The chaining approach for software test data generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 1, pp. 63–86, 1996.

[11] K. Chang, J. Cross, W. Carlisle, and S.-S. Liao, "A performance evaluation of heuristics-based test case generation methods for software branchcoverage," *International Journal of Software Engineering and Knowledge Engineering.*, vol. 6, no. 4, pp. 585–608, 1996.

[12] M. J. Gallagher and V. L. Narasimhan, "Adtest: A test data generation suite for ada software systems," *IEEE Trans. Softw. Eng.*, vol. 23, no. 8, pp. 473–484, 1997.

[13] B. F. Jones, H. H. Sthamer, and D. E. Eyres, "Automatic structural testing using genetic algorithms," *Software Engineering Journal*, vol. 11, no. 5, pp. 299–306, Sep 1996.

[14] N. Tracey, J. Clark, K. Mander, and J. A. McDermid, "An automated framework for structural test-data generation," in *Automated Software Engineering*, 1998, pp. 285–288. [Online]. Available: citeseer.ist.psu.edu/tracey98automated.html

[15] P. Mcminn, "Iguana: Input generation using automated novel algorithms. a plug and play research tool," Department of Computer Science, University of Sheffield, Tech. Rep., 2007.

[16] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *Journal of Chem. Phys*, vol. 21, pp. 1087–1091, 1953.

[17] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.

[18] J. J. Chilensky and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9 (5), pp. 193–200, 1994.

[19] "Antlr parser generator, http://www.antlr.org."

[20] J. Mitchell, T. Parr, J. Lilley, S. Stanchfield, M. Mohnen, P. Williams, A. Jacobs, S. Messick, and J. Pybus, "Java parser and tree parser (grammar version 1.22 -april 14, 2004)," http://www.antlr2.org/grammar/java, 2004.

[21] A. Tripp, "http://jazillian.com/antlr/emitter.html."

[22] A. P. Mathur, *Foundation of Software Testing draft V2.11.*, 2006.

[23] "Jgap java genetetic algorithms package, http://jgap.sourceforge.net." [Online]. Available: jgap.sourceforge.net

[24] P. S. May, "Test Data Generation: Two Evolutionary Approaches to Mutation Testing," Ph.D. dissertation, Kent University, UK, May 2007.

[25] W. T. Vetterling and B. P. Flannery, *Numerical Recipes in C*, W. H. Press and S. A. Teukolsky, Eds. Cambridge University Press, 1992.