Instrumenting Programs with Flag Variables for Test Data Search by Genetic Algorithm

Leonardo Bottaci
Computer Science Dept.
Hull University
Hull, HU6 7RX, U.K.

Abstract

Evolutionary search is potentially a powerful way of searching for software test data to satisfy various structural testing criteria. Specific test cases are evaluated by a fitness function constructed by instrumenting the program under test. The more discriminating the fitness function, the more efficient the search. When a program uses flag variables to store the results of predicate expressions, it is difficult to instrument the program effectively. The problem is examined and a solution is given for a special case. An approach for tackling the general cases is described.

1 INTRODUCTION

The testing of software is a time consuming and expensive activity and consequently the idea that it might be automated is an attractive prospect. The vast majority of testing tools in current use focus on automating the execution of test cases and on collecting coverage and output data to compare with known results. What these tools do not do, however, is generate the test data to satisfy a given criterion. In some cases, the data may be generated randomly but this is unlikely to be adequate to satisfy the criterion. After application of an initial set of tests, testers often have the problem of constructing additional tests to completely satisfy the given criterion. Even with a good knowledge of the subject program under test this can be challenging. The problem is worse when testers work, as they often do, on code written by others.

The automation of test data generation is a problem that has been tackled by a number of researchers. Ince (Ince 1987) gives an account of relatively early work in this area. The test data generation problem, for a nontrivial criterion, is undecidable in general. This, coupled with the increased awareness of the potential of heuristic search techniques has prompted researchers to use these techniques to find test data for various structural testing criteria. Korel (Korel 1990) and Ferguson (Ferguson and Korel 1996) have used function minimisation to find tests to satisfy path criteria. Jones *et al.* (Jones, Sthamer, and Eyres 1996) and Wegener *et al.* (Wegener, Sthamer, Jones, and Eyres 1997) have applied genetic algorithms (Goldberg 1989) to find test data to satisfy branch coverage and minimum and maximum execution times. Tracey *et al.* (Tracey, Clark, and Mander 1998) have used simulated annealing to search for failure conditions. A genetic algorithm is used in the GADGET (McGraw, Michael, and Schatz 1998), test data generation system. Pargas *et al.* (Pargas, Harrold, and Peck 1999) describe a test tool in which a genetic algorithm is used to search for test data that reaches a given node in the program control flow graph. The conformance of the test execution path to the control dependency conditions (Ferrante, Ottenstein, and Warren 1987) for the given node is used as the fitness function. The tool developed by Wegener *et al.* (Wegener, Baresel, and Sthamer 2001) uses evolutionary algorithms and, by combining node and path conditions, may be used to generate test data for most structural test criteria.

A common technique in the work mentioned above is the instrumentation of the subject program to produce a heuristic evaluation function or fitness function. Ultimately, the fitness function must evaluate the extent to which a given test case satisfies specific predicate expressions in the subject program. When the value of a predicate expression is stored in a flag variable there is the risk of losing the information gathered by the instrumentation code. This paper considers this problem and shows how to instrument a special case. An approach for tackling the general case is also described. The techniques described in this paper have

been implemented in a prototype test data generation tool.

2 INSTRUMENTING A PREDICATE EXPRESSION TO SEARCH FOR TEST DATA

Consider the control flow graph of a subject program. The nodes are the basic blocks of the subject and the edges are the possible transitions between basic blocks. The conditional transitions are associated with a branch predicate. There is a distinguished start node and a distinguished exit node. Many test criteria require a test case to execute a given statement. If the particular path is unimportant then the control dependency predicate path (or sometimes paths) of the goal statement specifies the required value for each critical branch predicate expression. A predicate expression may not be critical because it is not part of any path to the goal statement or because both outcomes may lead to the goal statement.

There are a number of approaches, grouped under the general heading of static methods, that attempt to calculate the appropriate condition on the input data by analysis of the program. One of these is symbolic evaluation (Clarke 1976) (Howden 1977). As the name suggests, the program itself is not executed but instead a description is constructed of how execution along a particular path would affect a set of variables. This description cannot give the precise values of variables but instead provides constraints on their possible values, expressed, ultimately on the values of the input variables. In general, however, the relationship between the input data and the values of internal variables at the point where they are used in a predicate expression may not be readily analysable because of the presence of loops and computed storage locations, e.g arrays and pointers.

Dynamic analysis is an alternative to static methods. In this approach, the subject program is instrumented in order that it may reveal, during execution, the information that can be used to guide the search towards the required test case. In the most basic instrumentation, a record is kept of the values of all branch predicate expressions executed. A cost for the given input is computed by counting the number of predicate conditions in the control dependency path of the goal statement that have not been satisfied by the execution of the program. This is the method described by Pargas *et al.* (1999). A zero cost indicates that a solution has been found whereas a nonzero cost indicates that an undesired branch was taken at some predicate.

Although a count of undesired branch decisions provides some guidance to the search; in some situations, the first few conditions of the control dependency path will be easily satisfied but the next condition may be quite difficult to satisfy. Our experience is that during genetic search in this situation, the entire population of test cases all quickly evolve to the same fitness which is that obtained by satisfying only the easy predicates. At this point, the search space has become flat and the search becomes random.

As an example, consider the program fragment below.

```
...
if (a <= b)
   ...          // EXECUTION REQUIRED
                // TO ENTER THIS BRANCH
```

Suppose we are seeking a test case that will cause execution of the true branch of the conditional shown above. If the required branch is difficult to enter, many test cases will cause `a <= b` to be false. To discriminate between these tests, the program in instrumented to calculate a cost measure that penalises those tests that may be considered to be "far from" satisfying `a <= b`. For this expression a suitable cost measure would be $a - b$. A test that has a zero cost ($a - b = 0$) "just" satisfies the condition. A positive cost indicates that the predicate expression is false.

The subject program is instrumented at the point a particular condition is required to hold, in our example at `a <= b`. Through instrumentation, the subject program has in effect been converted into another program that computes a function that we seek to minimise to zero. This method has been used by Korel (Korel 1990), Tracey *et al.* (Tracey, Clark, Mander, and McDermid 1998), Wegener *et al.* (Wegener, Baresel, and Sthamer 2001) and others.

Below are shown the typical relational predicate cost formulae. a, b are numbers and ϵ is the smallest positive constant in the domain (i.e. 1 in the case of integer domains and the smallest number greater than zero in the particular real number representation).

Predicate expression	Cost of not satisfying predicate expression
$a \leq b$	$a - b$
$a < b$	$a - b + \epsilon$
$a \geq b$	$b - a$
$a > b$	$b - a + \epsilon$
$a = b$	$abs(a - b)$
$a \neq b$	$\epsilon - abs(a - b)$

The cost formulae must be extended beyond the re-

lational predicates to the logical predicates to provide a cost for branch predicate expressions such as `a <= b and not(x > 0)`. As a simple case, consider the logical negation operator. Presented with the argument *true* with cost c it must return $false$ with cost $-c+\epsilon$. This cost formula for negation follows from consideration of the costs of $a \leq b$ and $a > b$. Possible cost tables for *or* and *and* are given in Table 1 where c_a is the cost representation of a boolean value a. In the table below, c_a and c_b are always positive so that the four rows correspond to the four rows of the classical truth table for two boolean values.

Table 1: Logical Operator Cost Table

a	b	a *or* b	a *and* b
c_a	c_b	$min(c_a, c_b)$	$max(c_a, c_b)$
c_a	$-c_b$	$-c_b$	c_a
$-c_a$	c_b	$-c_a$	c_b
$-c_a$	$-c_b$	$min(-c_a, -c_b)$	$max(-c_a, -c_b)$

The intuition behind the cost formula for *or* is that if either of the c_a or c_b costs are to be incurred we need incur only the least cost hence the min function. When both costs must be incurred, we are obliged to accept at least the maximum cost[1].

Tracey *et al.* (Tracey, Clark, Mander, and McDermid 1998) use essentially the same cost functions although their's are restricted to nonnegative values. A notable difference, however, is the use of + rather than max for *and*. One can argue that when both costs must be incurred, we are obliged to accept them both. There are also situations in which + is a better operator than min for *or* and the above truth table is presented only as a heuristic; the precise cost formulae used is not relevant to the subject of this paper.

3 FLAG VARIABLE PROBLEM

Given an effective cost function, problems can nonetheless arise in trying to use it. A particular problem is that sometimes the point in the program where a predicate expression is evaluated, this is the point at which a cost may be calculated dynamically, may not be the point at which the predicate value is used, which is where the cost value is required. In the software testing literature, this is often referred to as the boolean flag problem.

[1]Note that the above formulas are consistent with De Morgan's laws. For example, the cost of the negation of a *or* b is $-min(c_a, c_b) + \epsilon$ which is equal to $max(-c_a + \epsilon, -c_b + \epsilon)$ which is the cost of *not a and not b*. This cost equality is a stronger condition than is necessary, we require only that truth values be preserved.

3.1 SPECIAL CASE

The following code fragment illustrates a special case of this problem.

```
flag := a <= b;  // COST a - b CALCULATED
...
if (flag)        // COST a - b REQUIRED
   ...           // AS EXECUTION REQUIRED
                 // TO ENTER THIS BRANCH
```

In the predicate expression of the condition, we require `flag = true` but this is not a useful expression to instrument because a boolean variable can provide only one of two values leading to an ineffective cost function with a flat surface.

Since the problem arises because the information that is used to compute the boolean value of the flag is discarded once the flag is set, the solution described in this paper is to retain this information so that it may be used later in the execution when, for example, the flag is evaluated as part of a conditional. In this way, when the flag variable is evaluated in the predicate of the conditional, it may be associated with the cost of the expression `a <= b` computed at the time that the flag was set earlier, in the execution.

A prototype test data generation tool has been constructed in which the above scheme has been implemented. All predicate expressions in the subject program are instrumented to compute the costs described in the previous section. In addition, whenever any variable is assigned the value of a predicate expression, the cost of that expression is associated with the variable. In the example above, if `flag` is set then the cost $a - b$ is associated with `flag`. When the variable `flag` is used in the if-statement, the saved cost is retrieved and associated with the if-statement.

There may be a number of statements where a flag variable may be set and used during the execution of a program as for example

```
if (...)
   flag := a <= b;   // COST a - b IF EXEC
else
   flag := a >= c;   // COST c - a IF EXEC
...
flag := not(flag) and (x > 0);  // COST OF
                                // flag USED
                                // AND SAVED
```

Whenever a variable is set to the value of a predicate expression, the variable is also associated with the cost

of that expression and so the cost of any expression involving flag variables may be calculated.

Note that the above scheme allows for the cost of a predicate expression to be associated with the assignment of a value to any variable, even a computed variable such as an array element or pointer reference. Pointer variables have not yet been implemented in the prototype but they will not present a problem for this scheme.

3.2 GENERAL CASE

The above technique fails, however, when the boolean expression that in effect determines the flag value is not directly assigned to the flag but is used to control the assignment of a "summary" value, as is shown in the following fragment.

```
flag := false;
...
if (a <= b) {       // COST KNOWN HERE
   ...
   flag := true;
}
...
if (flag)          // COST POSSIBLY USEFUL HERE
   ...             // DESIRED BRANCH
```

When the flag is false and it is desired to set it true there is no cost value associated with the flag that can be use to guide the search towards satisfying `a <= b`. The cost of this expression is computed, however, but it is associated only with the first conditional statement.

It is not at all clear how the computed cost can be propagated usefully in this situation. To establish that this cost is even relevant to the problem it is necessary to recognise that the second assignment to `flag` is relevant to the selection of the required branch. Data dependence analysis (Aho, Sethi, and Ullman 1986) could be used to do this. We might then identify the conditional statement closest to the unexecuted assignment to the flag, i.e. that conditional which controls entry to the basic block that contains the assignment. If in the code `flag` is set true then the cost of the conditional predicate expression should be associated with the use of the flag in the second conditional statement and conversely if in the code `flag` is set false (and a false value is required) then the flag should be associated with the negation of the cost of the conditional predicate expression.

In the above fragment, we have the benefit of knowing that if the flag is set, it is set true, in general, unless the statement is executed, the value of the flag is unknown. With an unknown flag value there is the danger that the cost of the conditional predicate expression is not useful since it may guide the search towards the execution of a statement that does not change the value of the flag to the required value. A more difficult case is shown in the fragment below.

```
...
if (a <= b) {       // COST KNOWN
   ...
   if (a > c) {     // COST KNOWN IF a <= b
      ...
      flag := true;
```

The variable `flag` is set true only when both predicate expressions `a <= b` and `a > c` are true. This suggests that the cost to be associated with the value of `flag` at the assignment is the cost of `a <= b` *and* `a > c` but if `a <= b` is false then there is no cost for `a > c`.

In general, flag values can be known only when they are set and cost information can be collected only when predicate expressions are executed. The need to execute code in order to analyse it is a fundamental limitation of dynamic program analysis in general.

Ferguson*et al.* (Ferguson and Korel 1996) also tackle the problem of generating test data in a program with flag variables. They do not associate costs with flag values. The approach they take when a search fails to find a test case that will execute a required branch is to identify the statements which could affect the value of the flag. Data dependence analysis (Aho, Sethi, and Ullman 1986) is used to do this. Once these statements are identified, their execution become subgoals of the test generator. In this way, the search for statements to execute is goal directed and depth first in that a subgoal is pursued before a sibling goal.

Clearly, program execution must be directed to currently unexecuted parts of the program. It may not be necessary, however, to use a focussed technique to identify the specific statements that may affect the predicate expression under consideration. In any case, it is not possible to tell if the execution of the statement will solve the problem unless the statement is executed. A much simpler approach, for example, would be to attempt to execute all acyclic paths that reach the problem node.

A genetic algorithm is well suited to exploring many areas of the search space in parallel (breadth first search on a serial machine). The initial population of random test cases could be separated into separate subpopulations or islands. In each island there would

be a search for a specific acyclic path. Given the use of a genetic algorithm as a search tool for this work it seems sensible to investigate if the ability of the genetic algorithm to search different parts of the search space breadth first can be exploited to solve this problem. It may turn out that a search for test cases that will execute all acyclic paths to the required branch is reasonably efficient in practice in which case it is possible to avoid the implementation complexities of a more goal directed search.

4 IMPLEMENTATION

A prototype test data generation system has been written (using CMU Common Lisp) to apply the ideas described in this paper to example programs. The prototype has two main modules. One module is concerned with the instrumentation of the subject program and the other, smaller, module is responsible for searching for test data using a genetic algorithm.

The subject program is parsed into an abstract syntax tree[2]. From this tree is generated the instrumented subject program. The subject and mutant program is instrumented as follows. Each conditional statement has a fixed length FIFO queue[3] in which predicate expression costs are saved as they are calculated. In addition, the conditional retains the lowest positive cost produced (recall that the cost is positive when the predicate is false) and the highest non-positive cost (recall that the cost is zero or negative when the predicate is true). In this way it is possible to determine if both branches have been taken. Each variable also has a similar fixed length FIFO queue in which is saved the cost of any predicate expression value assigned.

In addition to the subject program, the only additional information that the user may supply is a constraint and probability distribution on the subject program input domain. This is done by defining subsets of the input domain and assigning a probability to each subset. The input domain definition and probability distribution is used to create the initial random population of test cases. For each designated subset, the user may also specify the parameters of the genetic mutation operator (uniform or Gaussian distribution and variance).

Test inputs are coded not as binary strings but as strings of atomic values of the common programming language data types, i.e. integer, float, etc. The genetic algorithm is of the steady-state variety and similar to Genitor (Whitley 1989). Reproduction takes place between two individuals who produce one or two offspring (depending on the choice of reproduction operator). These offspring are then immediately inserted into the population expelling the one or two least fit. The population is kept sorted according to cost and the probability of selection for reproduction is based on rank in this ordering.

5 CONCLUSIONS

The instrumentation of predicates in a program under test is a common technique for guiding the search for test data. The presence of flag variables, however, has long been recognised as an impediment to such instrumentation. The approach described in this paper is to propagate the cost information from the predicate expression instrumentation, i.e. the statement in the program where it is calculated, to the conditional where it is required. This technique cannot solve the general problem with flag variables but here it is proposed to use the genetic algorithm to search among the relevant acyclic paths.

[2]Currently the subject program must be hand translated into a common input language. The common input language is an expedient that for the purpose of research avoids the need to construct a parser for the language of the subject program. It has not yet proved to be a handicap since it is not difficult to find quite small subject programs that provide the test generator with a difficult challenge. The common input language is procedural and block structured although not all the features found in this type of language are as yet available. In particular, pointers are absent and the array is the only aggregate data type.

[3]A queue of costs is required to tackle the problem of instrumenting code within loops, a problem not relevant to this paper.

References

Aho, A. V., R. Sethi, and J. D. Ullman (1986). *Compilers: Principles, Techniques and Tools.* Addison - Wesley.

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

Ferguson, R. and B. Korel (1996, January). The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology 5*(1), 63–86.

Ferrante, J., K. J. Ottenstein, and J. D. Warren (1987, July). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems 9*(3), 319–349.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning.*

Addison Wesley.

Howden, W. (1977). Symbolic testing and the dissect symbolic evaluation system. *IEEE Transactions on Software Engineering SE-4*(4), 266–278.

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

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

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

McGraw, G., C. Michael, and M. Schatz (1998). Generating software test data by evolution. Technical Report RSTR-018-97-01, RST Corporation, Suite 250, 21515 Ridgetop Circle, Sterling VA 20166.

Pargas, R. P., M. J. Harrold, and R. P. Peck (1999). Test-data generation using genetic algorithms. *Software Testing, Verification and Reliabillity 9*, 263–282.

Tracey, N., J. Clark, and K. Mander (1998, March). Automated program flaw finding using simulated annealing. *Software Engineering Notes 23*(2), 73–81.

Tracey, N., J. Clark, K. Mander, and J. McDermid (1998). An automated framework for structural test data generation. *Procceedings of the 13th IEEE Conference on Automated Software Engineering*.

Untch, R. H., A. J. Offutt, and M. J. Harrold (1993). Mutation analysis using mutant schemata. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis ISSTA 1993*, New York, NY, USA, pp. 139–147. ACM.

Wegener, J., A. Baresel, and H. Sthamer (2001). Evolutionary test environment for automatic structural testing. *Information and Software Technology 43*, 841–854.

Wegener, J., H. Sthamer, B. F. Jones, and D. Eyres (1997). Testing real-time systems using genetic algorithms. *Software Quality Journal 6*, 127–135.

Whitley, D. (1989). The genitor algorithm and selective pressure: why rank based allocation of reproductive trials is best. *Proceedings of the Third International Conference GAs.*, 116–121.