An improved strategy for the automatic generation of test data

S.T. Holmes\textsuperscript{a}, B.F. Jones\textsuperscript{b}, D.E. Eyres\textsuperscript{a}

\textsuperscript{a}Department of Computer Studies, University of Glamorgan, Pontypridd, Mid Glamorgan, CF37 1DL, UK
\textsuperscript{b}Department of Electronics and Information Technology, University of Glamorgan, UK

ABSTRACT

Mutation analysis is used to check the efficacy of automatically generated test data. A program is mutated by making a single, syntactically correct change such as deleting a statement or changing the predicate in a selection. The mutation adequacy is defined to be the percentage of all mutants killed by the test data. Our strategy for automatically generating test data uses direct assignment and alternating variable heuristics to kill 79.2\% of the mutants. Minor changes to predicates in if-statements are best revealed by generating test data which fall close to the domain boundary for that predicate using linear extrapolation of previous tests. Having found the domain boundary, a heuristic which follows the boundary is used to search for the isolated domains relating to equalities. For example, there are very few instances when the predicate 'if \((B^2-4\times A\times C)=0\)' is satisfied for integer input variables A, B and C. The introduction of the domain boundary follower heuristic increases the mutation adequacy of the automatically generated test data to 97.9\%.

INTRODUCTION

Testing is a key activity in the assurance of software quality. Nevertheless, repeated execution of a program can never engender complete confidence that a computer program is free from errors, even when every execution produces the expected results. This is because input domains for a program are often so
large that all the test data sets could be equivalent in the sense that they check
the same function and exercise the same path through the software. This is just
one of many reasons why testing is such an important and time-consuming part
of the software life-cycle, and why errors continue to be discovered long after
the software has become an industry standard.

Those software engineers engaged in testing need a measure of how well their
test sets have exercised their software. In response, several test effectiveness
ratios have been developed (Woodward, Hedley and Hennell, 1980 [9]); they
measure such things as the cumulative percentage of the total number of
statements which have been executed by the test data sets. This and other test
effectiveness ratios are appropriate for white box testing in which the software
gineer uses a knowledge of the control flow of the software to guide the
choice of the test sets. Such metrics are severely limited by the problem of the
equivalence of test sets; sets of test data may be devised to give test
effectiveness ratios approaching a value one for incorrect software.

A more rigorous means of assessing the thoroughness of tests is to use
mutation analysis (Budd, 1981 [1]; DeMillo, Sayward and Lipton, 1979 [2];
Offut, 1992 [6]). Test data are input both to the original program and to a
mutated version in which a single, syntactically correct change has been made.
If the presence of the mutation is revealed, the test data are said to have killed
that mutant, otherwise the mutant remains alive and the test data are deemed
inadequate. In the latter case, a new set of test data must be substituted.

The application of the above approach to the testing of a large software system
could only be made in Utopia. The need for a tool which will generate test sets
automatically is desperate. The minimum test set which will adequately
exercise the software is required for regression testing which can be applied to
new versions. Usually, updated software must still perform the original
functions correctly, and failure to do so would result in a damaging loss of the
users' confidence.

The aim of this project is to generate test sets automatically and adaptively, and
to assess the efficacy of those test sets using mutation analysis as a metric.
ADAPTIVE TEST DATA GENERATION.

Adaptive generation uses previously tried test data to produce new and more effective test data. The fact that the new test data are expected to be more effective implies that the software engineer has a metric to measure the success in satisfying a specific goal, such as causing the flow of control to pass along a particular path. In this case, a control flow tree like the one shown in figure 1 must be produced for the software under test. Such a control flow tree is useful when the software engineer is using the test effectiveness ratio based on the cumulative percentage of all paths traversed. Our intention is to adaptively generate the test data automatically, so removing an immense burden from the software engineer. In order to do this, several heuristics have been developed; the heuristics use historical test information to predict new test data in the pursuit of forcing the flow of control to pass along a hitherto untraversed path in the tree.

DESCRIPTION OF THE HARNESS AND THE PROCEDURE UNDER TEST.

The aim of the project is to generate test data which cause every path of an ADA procedure to be executed (see figure 1); this is a more effective measure of software reliability than either ensuring that every statement is executed or that every branch is taken (Veevers, 1991 [7]). Coverage of paths rather than branches is possible because as a starting point, the chosen procedure contains only sequences and selections. This avoids the problem of an infinite control flow tree arising from the presence of iterations.

The selection condition is simplified to be of the form condition_variable:relation_operator:zero, which is similar to the approach used by Korel (1990) [5] who describes a series of predicate transformations. This enables us to measure easily the progress towards satisfying certain conditions. Conditions with logical connectives will be included in later versions of the system.

A quadratic equation solver is chosen as the procedure under test; the code is shown in figure 2, and its control flow tree in figure 1. The nodes of the
control flow tree represent the sequence of statements between each selection. The arcs represent the predicate which causes control to branch from the parent node to the child.

The quadratic equation solver has three input variables A, B, and C; it proves to be an interesting subject for investigation because as reference to figure 2 shows, 1) one of the input variables, A, is involved directly in a condition, 2) D is a non-linear function of the input variables, namely \((B^2 - 4A*C)\), and is involved in two conditions controlling a total of three paths, 3) D is involved in an equality \((D=0)\), 4) There are no iterations, 5) the results may be checked automatically against post-conditions (namely, the sum of the roots equals \(-B/A\), and the product of the roots equals \(C/A\)).

The input variables are integers rather than floating point variables; otherwise the equality condition, \(D=0\), would not be valid because of rounding errors inherent in the calculation of D. At first sight, this appears to be a simplification but in fact it makes the problem less tractable and gives rise to instabilities in the prediction of test data sets under certain circumstances. The main thrust of this work is to solve this problem which arises frequently for conditions involving ordinal data types.

MUTATION ANALYSIS

Test data which are to be applied to the original program and its mutant are chosen either from a database of test cases or selected by hand. If the output from the original program is incorrect, the error must be found and corrected. Under these circumstances, the whole process of mutation analysis is repeated from the beginning. If on the other hand, the output from the original program is correct, the same test data are submitted to the mutant. The output from the original program are compared with that from the mutant. If there is no difference, further test data are devised in the hope of detecting the mutation. When the outputs from the original and mutated programs become different, the mutant is said to be killed; the test data are then stored in a data base of test cases for the purposes of killing further mutants and regression testing. This process is repeated for all possible mutants.
The rules that define mutations are termed mutation operators. Budd (1981) [1] has classified them to correspond to other established criteria for the adequacy of test data. A significant problem with using mutation analysis is that some mutants will be functionally equivalent to the original program; in this case, no test data will reveal a difference. The criterion known as mutation-adequacy is the percentage of non-equivalent mutants which have been killed. Offut (1992) [6] suggests that a mutation-adequacy of 95% is an acceptable indication that the test program has been thoroughly tested.

In our experiments, we have used mutation analysis in a slightly different way in that we use the post-conditions to check the validity of the results rather than by comparing two sets of output data. The procedure under test is relatively simple and is correct to the best of our knowledge. The mutation operators used are from two classes of operators or levels of analysis (Budd, 1981 [1]). The first level is statement analysis in which either a statement is deleted or a condition is replaced (ie 'if A=0' becomes 'if TRUE'). This ensures that every branch is taken and that every statement is necessary. The second level is predicate analysis in which a predicate is altered by a small amount (eg 'if D=0' becomes 'if D=1'), the abs operator is inserted (eg 'if D>0' becomes 'if abs(D)>0'), or the relational operator is altered (eg 'if A=0' becomes 'if A/=0'). Using these mutation operators, the 48 non-equivalent mutants indicated in Table 1 were produced.

In our experiments, the quadratic equation procedure is mutated and re-compiled manually. A single set of input test data is generated randomly and the procedure executed. The adaptive test generator now takes over and attempts to produce test data which cause every path to be traversed. If the post-condition is violated, the system stops because the mutant has been killed. Otherwise, the system continues until either all paths have been traversed or terminates because a pre-defined arbitrary limit on the number of attempts has been reached.

THE DIRECT ASSIGNMENT HEURISTIC.

When the variable involved in a condition is also an input parameter, the heuristic is straightforward, and is similar to one of the rules proposed by
Howden (1987) [4]. The direct assignment heuristic assigns a value directly to the input parameter to cause the condition to be set as required. For example, one of the conditions in our problem is A=0 (see figure 2); it is a simple matter to recognise that A is an input variable and to set it equal to zero.

THE ALTERNATING VARIABLE HEURISTIC.

This heuristic comprises two stages. There are three input parameters to the quadratic equation solver. In stage 1, one variable is increased and decreased by a small amount, whilst the others are held constant. After each change, the value of the condition variable relevant to the node under investigation is checked to see if it has moved closer to the required value to bring control flow to that node. Stage 2 of the strategy comes into play when this is the case; the input variable is then modified further in the successful direction with ever-increasing steps in the hope that the condition will become true or false as required. Stage 1 is thus an exploratory search, and stage 2 is a pattern search similar to those suggested by Glass and Cooper (1965) [3]. If the situation arises where there is no further progress towards the goal, another input variable is selected and stages 1 and 2 are repeated.

THE EFFECTIVENESS OF TEST DATA GENERATED BY THE DIRECT ASSIGNMENT AND THE ALTERNATING VARIABLE HEURISTIC.

The direct assignment and alternating variable heuristics are straightforward to implement and efficient in their operation. They are used in the first version of our system to generate sets of test data automatically. The effectiveness of the resulting tests is measured using the criterion of mutation adequacy, and the results are given in Table 1. Over 80% of the mutants belonging to the statement-analysis class of mutation operators are killed. Similarly, over 80% of the mutants arising from the predicate-analysis class of mutation operator are killed, with the exception of the alteration of the predicate by a small amount when the mutation adequacy falls to 33%.

Although it is not obvious from Table 1, there are three causes for the reduction of the mutation adequacy below 100%. The first reason is that the automatic test system fails to generate data which causes the flow of control to
follow the path 1-3-5-6 (figure 1) ie it fails to predict a combination A, B and C to give a value of zero for D \((= B^2 - 4*A*C)\). This is not surprising since no information about this relationship is given to the system in order to preserve its generality. In certain areas of the input domain, there are very few combinations of A, B and C which give a zero value for D; in fact if B is odd, there are no zeros. Under these circumstances, the direct assignment heuristic is not applicable, and the alternating variable heuristic is too coarse an instrument to hit one of the isolated zeros within a reasonable number of attempts. This affects the mutation adequacy for both classes of mutation operator.

The second reason affects only the second class of mutation operator in which the predicate is changed in a minor way; for example 'if D>0' becomes 'if D>1'. This mutation is only revealed when the inputs combine to give a value of 1 for D. This demonstrates clearly that errors are more likely to be detected when test data are chosen close to the domain boundaries. These results prompted the development of the boundary follower heuristic.

The third reason is that the direct assignment heuristic cannot handle the mutation of 'if A=0' to 'if A<=0' which would require a value of -1 for A.

**LINEAR PREDICTOR HEURISTIC.**

When the alternating variable heuristic approaches and oversteps the value required to satisfy the condition, the system returns to the closest previous test point. A better value for the input parameters may be predicted by a linear extrapolation on each of the input variables in turn. The heuristic works well of course if there is a linear relationship between the input parameters and the condition_variable; this is the most usual case (White and Cohen, 1980 [8]). For example, in our problem, D varies linearly with either A or C and an accurate prediction is made provided that the other two input variables are held constant. However, misleading results would be obtained when attempts are made to predict a value of B to generate a certain D because the relationship is quadratic. This problem is particularly acute when trying to satisfy the condition D=0. To ameliorate this problem, the heuristic detects the non-linearity, returns to the original point and alters the input variable by a
small amount in the appropriate direction. Under certain circumstances, these three heuristics fail because of the sparseness of zeros in the domain of a predicate involving an equality such as ‘if D=0’. A domain boundary follower has been developed to search for sparse or disconnected domains such as that for D=0.

**DOMAIN BOUNDARY FOLLOWER HEURISTIC.**

All of the previous heuristics may have difficulty in locating a domain which comprises a sparse collection of disconnected points. The domain for D=0 is such an example. These points may be located by tracing the boundary of the domains for D>0 and D<0.

Before the domain boundary follower heuristic can come into play, the boundary itself must be found using the principle underpinning the linear predictor heuristic. Once found, the boundary is followed by visiting sequentially all the points which are adjacent to it until either a point in the required domain is encountered or the heuristic decides that further searching is fruitless. There are two possible moves in following the boundary (see figure 3); they are (a) a follow move, in which modifying a particular input variable moves to a new point which may either stay in the same domain or cross the boundary into a different domain, and (b) a cross move, in which modifying a different input variable moves to a new point which must be on the other side of the domain boundary. The follow and cross moves are therefore associated with two specified input variables. In our example, the boundary follower ‘stitches’ the boundary between the domains for D>0 and D<0 until a point where D=0 is found.

**IMPROVED RESULTS**

When the direct assignment, alternating variable and boundary follower heuristics are applied to the mutants, a total of 47 out of the 48 mutants (97.9%) are killed automatically. The only mutant to survive is the mutation of the condition involving an input variable, viz ‘if A=0’ changed to ‘if A<=0’. The introduction of the domain boundary follower heuristic has increased the mutation adequacy from 79.2% to 97.9%. A drawback is the large amount of
computer processing time which may be required to kill the mutant. An arbitrary limit of 500 attempts is placed on the domain boundary follower; it terminates if a solution is not found. Three mutants in particular caused problems in this respect: (a) the deletion of a statement following the 'if D=0' statement; (b) the alteration of 'if D=0' to 'if D=1'; (c) the alteration of 'if D=0' to 'if D=-1'. The overall reason for these problems is that the inputs must combine to give particular values for D. In (a), D must be exactly 0. In (b), D must be 0 since a value of 1 would cause control to peel off correctly at a higher level in the control flow tree. In (c), values of 0 or -1 reveal the mutation. Such values are not found since the random number generator chose large initial values (between 500 and 1000) for A, B and C so that the search concentrates in an area of the domain where values of 0, 1 and -1 for D are few and far between. In this case, the domain boundary follower tends to push the search area in the most unpromising direction. This problem is overcome by limiting the initial values for A, B and C to be between 0 and 10. In this case the mutants are killed quickly. The issues of improving the heuristics for selections involving the input variables, and of devising more efficient means of following the domain boundaries are being tackled.

CONCLUSIONS

The domain boundary follower heuristic has proved successful in devising test data which exercise thoroughly a piece of software. In the present work, 47 out of 48 mutants (97.9%) are killed automatically using our library of heuristics. The large amount of computation time required may be a problem in some circumstances. This is solved by more careful choice of the starting point for the tests so that the search concentrates on an area of the domain where there is a high concentration of goals. Work continues on improving the efficiency and intelligence of the heuristics.
REFERENCES


### Table 1: Mutation experiment results using the direct assignment and alternating variable heuristics.

<table>
<thead>
<tr>
<th>Mutation operator</th>
<th>M</th>
<th>KM</th>
<th>K%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement Analysis</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statement deletion</td>
<td>22</td>
<td>18</td>
<td>82</td>
</tr>
<tr>
<td>Condition replacement</td>
<td>6</td>
<td>5</td>
<td>83</td>
</tr>
<tr>
<td>Predicate Analysis</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Predicate alteration by small value</td>
<td>6</td>
<td>2</td>
<td>33</td>
</tr>
<tr>
<td>Abs operator insertion</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>Relational op. alteration (line 3, figure 2)</td>
<td>5</td>
<td>4</td>
<td>80</td>
</tr>
<tr>
<td>Relational op. alteration (line 7, figure 2)</td>
<td>5</td>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>Relational op. alteration (line 14, figure 2)</td>
<td>3</td>
<td>3</td>
<td>100</td>
</tr>
<tr>
<td>Overall</td>
<td>48</td>
<td>38</td>
<td>79</td>
</tr>
</tbody>
</table>

- M - Number of mutants produced
- KM - Number of mutants killed
- K% - Percentage of mutants killed (mutation adequacy)

### Figure 1: The control flow tree of the quadratic equation solver
procedure QUADRATIC ( A, B, C : in INTEGER; XI, X2 : out COMPLEX; QUAD_KIND : out QUAD_TYPE ) is

D : INTEGER; REAL_PART : FLOAT; IMAG_PART : FLOAT;

begin
1 SET ( X1, 0.0, 0.0 );
2 SET ( X2, 0.0, 0.0 );
3 if ( A = 0 ) then
4 QUAD_KIND := NOT_A_QUADRATIC;
else
5 D := (B*B)-(4*A*C);
6 if ( D > 0 ) then
7 QUAD_KIND := ROOTS_ARE_REAL_AND_UNEQUAL;
8 REAL_PART := ( FLOAT (-B) + SQRT(FLOAT (D))) / FLOAT (2*A);
9 SET ( X1, REAL_PART, 0.0 );
10 SET ( X2, REAL_PART, 0.0 );
else
11 QUAD_KIND := ROOTS_ARE_REAL_AND_EQUAL;
12 REAL_PART := FLOAT (-B) / FLOAT (2*A);
13 SET ( X1, REAL_PART, 0.0 );
14 SET ( X2, REAL_PART, 0.0 );
else
15 QUAD_KIND := ROOTS_ARE_COMPLEX;
16 REAL_PART := FLOAT (-B) / FLOAT (2*A);
17 IMAG_PART := SQRT( FLOAT (((4*A*C)-(B*B))) / FLOAT (2*A);
18 SET ( X1, REAL_PART, IMAG_PART );
19 SET ( X2, REAL_PART, IMAG_PART );
20 end if;
21 end if;
22 end if;
23 end QUADRATIC;

Figure 2: The ADA code of the quadratic equation solver.
Figure 3: The follow and cross moves of the domain boundary follower heuristic.