A Genetic Algorithm Fitness Function for Mutation Testing

Leonardo Bottaci*
*Department of Computer Science, University of Hull, Hull HU6 7RX, U.K.

April 2001

1 Introduction

Many problems in software engineering have no algorithmic solution, one of these is the problem of finding good test data for a given program. Heuristic search techniques, however, have been used to find test data for various structural testing criteria. Korel (1990) used function minimisation to find tests to satisfy path criteria. Jones *et al* (1996) (1997) has applied genetic algorithms (Goldberg 1989) to find test data to satisfy branch coverage and minimum and maximum execution times. Tracey *et al* (1998) have used simulated annealing to search for failure conditions.

The mutation testing criteria, originally proposed by DeMillo *et al.* (1978) and Hamlet (1977), requires that test data demonstrate the absence of a pre-specified set of faulty programs. A fault is assumed to be manifest as a small modification to the correct program code. A copy of the subject program that contains such a modification (deliberately introduced) is called a mutant and a mutant is said to be killed by a test case if for that test the output of the mutant differs from that of the subject.

To demonstrate the absence of the fault in the mutant (to kill the mutant) it is necessary that the test case cause execution to reach the mutated statement (reachability condition), the value of the mutated expression must be such that it is possible to have a different data state in the mutant compared to the subject (necessity condition) and this state difference must be propagated to the output (sufficiency condition). By incorporating these three conditions into the fitness function of a genetic algorithm a search may be made for test data to kill a given mutant. This paper describes such a fitness function. The function described has been implemented and preliminary results should be available in time for the workshop.

2 Genetic algorithm fitness function

The genetic algorithm searches the input domain of the subject program for suitable test cases to kill a given mutant. Guidance is provided by the fitness function which assigns a non-negative cost to each candidate input value. An input with a zero cost is a test case that kills the mutant.

2.1 Reachability condition

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.

To kill a mutant, the execution path of a test must reach the mutated statement, the particular path is unimportant; however, for the purpose of explanation consider first the simple (if unlikely) case in which there is only a single path to the mutated statement, call it the goal path. The reachability cost of an input is determined by comparing its execution path with the goal path. A cost for path - goal path difference may be defined as the length of the goal path minus the number of nodes in the longest common prefix of the path and goal path. Any zero cost path is the goal path.

In the case where two candidate inputs have paths that have the same nonzero cost, they both fail to satisfy the same required branch predicate, it may be possible to discriminate them on the basis that the preferable candidate path is that which comes closer to satisfying the common failed branch predicate. To do this, a predicate may be associated with a numerical value that indicates the extent to which it is not satisfied. The following scheme (or something similar) is typical for guiding the search for values that satisfy numerical predicates.

If a, b are numbers then for a small positive constant P (say 1 in the case of integer arguments and something very small in the case of floating point arguments) and a large positive number L then the cost of not satisfying a predicate is given below.

Expression	Cost
$a = b$	0 when $a = b$ else $min(abs(a - b), L)$
$a < b$	0 when $a < b$ else $min(a - b + P, L)$
$a \leq b$	0 when $a \leq b$ else $min(a - b, L)$
$a > b$	0 when $a > b$ else $min(b - a + P, L)$
$a \geq b$	0 when $a \geq b$ else $min(b - a, L)$

The use of L to provide an upper bound for the cost is a practical necessity to avoid overflow when summing costs. Note that if L is not sufficiently large then discriminatory information may be lost. For expressions other than those above:

Expression	Cost
e	0 when $e = true$ else P
$e_1 = e_2$	0 when $e_1 = e_2$ else P
$e_1 \neq e_2$	0 when $e_1 \neq e_2$ else P
$\neg e$	remove $\neg$ by rewriting relational operators in e
$e_1 \wedge e_2$	$min(cost(e_1) + cost(e_2), L)$
$e_1 \vee e_2$	$min(cost(e_1), cost(e_2))$

In the formula for the disjunctive case, the higher cost is ignored, an acceptable tactic when only one condition need be satisfied.

The reachability cost thus consists of two components, the path difference cost and the the cost of failing a branch predicate. The path difference is the most significant in that the branch cost is used only when the path costs are equal.

Since, in general, a mutated statement may be reached by many paths, each such path is a goal path giving a disjunction of goal path conditions to satisfy. The cost of any given path is thus the minimum cost obtained when it is costed against all the goal paths.

2.2 Necessity condition

The necessity condition is the condition on the mutated object, or some expression containing the object, that must be satisfied for the mutant to be killed. For example, if in a program the integer variable `x` is mutated to `abs(x)` (the function that returns the absolute value) then a necessary condition for the mutant to be killed is that `x` be negative. If the necessity condition is not satisfied then the data state immediately after the mutated statement must be equal to the data state immediately after the corresponding statement in the subject program and consequently the mutant will behave identically to the subject program. In some cases, for example the mutation obtained by the insertion of the unary logical operator `not`, the necessity condition is always true since irrespective of the input, the logical expression `p` and `not(p)` will always have different values. More generally, the necessity condition costs may be calculated using the same scheme as that used for branch condition costs with the proviso that if the mutated statement is not reached then the necessity condition cost is the maximum, i.e. L.

A necessity condition may be satisfied but yet not cause a data state difference after execution of the mutated statement, i.e, the necessity condition may be weaker than the weak mutation condition which requires the data state after execution of the mutated statement to be different to the corresponding state in the subject. The necessity condition may thus be useful to evaluate the fitness of tests that do not satisfy the weak mutation condition.

Sometimes the necessity condition depends only on the mutation and can therefore be evaluated once only at the time the mutant is specified. In general, however, the necessity condition depends on the value of the mutable object or the surrounding expression and thus requires the execution of the mutant. Where the mutated statement is executed more than once, we require only a single instance of the necessity condition to hold. In this case the necessity condition cost is the minimum of the costs of the instances.

2.3 Sufficiency condition

A mutant survives when, even though there is a data state difference between the subject and mutant program at the mutation statement, that difference is not propagated to the output. The execution traces of a test case, a record of successive data states, in both the subject and mutant program may, however, be used to evaluate the cost of that test case in terms of failing to propagate a state difference from the mutated statement to the output, i.e. the data trace equality cost.

Since the mutant differs from the subject in only a single statement, up to that statement the two programs have identical behaviours. Beyond the mutated statement, data states may differ and paths may diverge although path divergence does not guarantee state divergence. Furthermore, the output may depend on a proper subset of the final data state and so to kill the mutant the difference between subject and mutant programs must be lie within the output part of the final data state.

Consider how, for a given test case, we might compare the subject and mutant execution traces to obtain the data trace equality cost. Assume, for example, that for a given test, the subject and mutant programs execute the same path. In this case, the respective execution traces in the subject and mutant can be “synchronised” for state by state comparison. If the mutant is not dead then the final data states, restricted to the output variables, are equal. Some number of previous consecutive states may also be equal. A cost function might penalise a test case in proportion to the number of equal state pairs it produces.

When the subject and mutant execute different paths a difficulty arises in identifying which state in the subject trace should be compared with which state in the mutant trace. Consider the subject and mutant traces shown below. Assume that the subject and mutant paths share no common nodes except the mutated statement and the exit.

```
subject S1   S2   S3   S4   S4

mutant  S5   S6   S2   S3   S4
```

If state comparisons are synchronised from the output (rightmost end of each trace shown) then only one pair of states (the output states) S4 and S4 are equal. This gives a trace cost of 1. Notice, however, that by deleting one of the repeated S4 states in the subject trace, because of the matching of S2 and S3, the cost increases to 3, perhaps a more accurate value. If we consider that for a typical program the output range is usually smaller than its input domain then the execution trace can be seen as a process in which different values in the input converge to the same value in the output. It could be argued that although, in general, data is processed differently along the different subject and mutant paths, for the input in the example, convergence takes place at state S2 and hence the cost should be 3. A trace equality cost might therefore be calculated by deleting, from each trace, repeated elements from a subsequence of consecutive equal elements and then returning the length of the longest common suffix.

This rule does not always give a reasonable result, however. In the example below,

```
subject S1   S2   S3   S3   S3   S3   S3

mutant  S5   S6   S2   S3   S3   S3   S3
```

the above rule gives a cost of 2 but if these two traces had been generated from the same path in the subject and mutant we would want to simply count equal states to give a cost of 4. A possible compromise, however, is to delete not all but only some repeated elements from a subsequence of consecutive equal elements so as to maximise the number of equal state pairs in the longest common suffix. Clearly, cost rule variations such as these need empirical investigation.

When two inputs have the same data trace equality cost it may be possible to discriminate them on the basis of the latest pair of non-equal states in the trace of each input. When comparing a subject and mutant data state, a state equality cost can be defined in terms of the number of state components that have equal value. Ideally, the cost function will be biased to those variables that have the most influence on the output. Variables that cannot influence the output would be ignored. Dependency analysis could be used to identify the relevant variables, for a specific mutant and test case. The relationship between the state equality cost and the trace equality cost is analogous to that between the branch condition cost and the path difference cost.

2.4 Combining reachability, necessity and sufficiency costs

The various cost components, path difference, branch condition, necessity, data trace equality and state equality may be maintained separately as ordered keys for sorting the population of the genetic algorithm. An individual's fitness is then determined by its rank.

The disadvantage of not calculating a single scalar fitness value is that credit cannot be assigned easily to the operators that produce the best individuals. Such credit, the difference in fitness between the new offspring and the previous best individual, can be used to bias the selection of genetic operators towards those that have produced the fittest offspring. To implement biased operator selection the various cost components must be combined into a single scalar value. To do this it is necessary to know the maximum value of each component. From the control flow graph and the length of the data state store of each statement, the maximum value of all components may be determined.

3 Path reduction

Loops in the subject program are likely to lead to execution traces that are too long to store and so they must be bounded in some way. Each statement is thus given a bounded length store in which to store successive data states. If the number of states produced during program execution exceeds the length of the store, states are lost. This policy can be thought of in terms of a function that given an arbitrary path from the program control flow graph, reduces the path by deleting repeated nodes until no node appears more than *k* times where *k* is the length of the statement data state store.

Note that to evaluate the reachability condition, retaining only the first node is adequate since multiple visits can be discounted when the aim is to simply reach the statement. To evaluate the sufficiency condition it is necessary to examine the last state produced at a given statement. This suggests that at least the first and last states should be saved for each statement.

4 Species

The basis of reproduction in a genetic algorithm is that two fit individuals (the so called parents) may be mated to produce an even fitter individual (the offspring) by combining in the offspring the best from each parent. This assumes that the good characteristics of one parent are compatible with those of the other. When searching for a path to reach a given statement, this is not necessarily so. In fact, taken as a whole, the branch predicates in different paths are contradictory. There is thus a danger that crossover, say, applied to two inputs that produce very different paths will generate an input with a path that is worse than that of either of the parents. The same problem arises when crossing two inputs that both reach the mutated statement but then take different paths from the mutated statement to the exit.

To avoid reproduction between incompatible individuals, the population of individuals is partitioned into subsets known as species. The species of an individual is determined by its execution path in the subject. Each path that does not reach the mutated statement defines a single species. The individuals with paths that reach the mutated statement do not require partitioning to achieve the reachability condition, only the sufficiency condition. They are thus partitioned according to the execution path suffix that begins at the mutated statement, i.e. each path from the mutated statement to the exit defines a single species.

A possible policy is to not allow reproduction between different species. A more liberal policy might allow reproduction between individuals of similar species. Again such questions require empirical investigation.

5 Conclusion and further work

A fitness function has been described that incorporates the three basic conditions required of a test case for it to kill a given mutant of some subject program. The use of a genetic algorithm to search for test cases that satisfy the reachability condition is not new. The attempt to incorporate necessity and sufficiency conditions is more innovative. Clearly, empirical investigation is required to assess the effectiveness of the fitness function described. The fitness function has been implemented, together with a genetic algorithm and mutation analysis tool. The results of a preliminary investigation should be available in time for the workshop.

References

DeMillo, R. A., R. J. Lipton, and F. G. Sayward (1978). Hints on test data selection: help for the practising programmer. *IEEE Computer 11*(4), 34–41.

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

Hamlet, R. G. (1977). Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering 3*(4), 279–290.

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.

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

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