FROM WEAK TO STRONG, DEAD OR ALIVE? AN ANALYSIS OF SOME MUTATION TESTING ISSUES.

M.R.Woodward and K.Halewood

Department of Computer Science, University of Liverpool,
P.O. Box 147, Liverpool, L69 3BX, U.K.

Abstract.

Despite the intrinsic appeal of the mutation approach to testing, its disadvantage in being computationally expensive has hampered its widespread acceptance. When weak mutation was introduced as a less expensive and less stringent form of mutation testing, the original technique was renamed strong mutation. This paper argues that strong mutation testing and weak mutation testing are in fact extreme ends of a spectrum of mutation approaches. The term **firm mutation** is introduced here to represent the middle ground in this spectrum†. This paper also argues, by means of a number of small examples, that there is a potential problem concerning the criterion for deciding whether a mutant is 'dead' or 'live'. A variety of solutions are suggested. Finally, practical considerations for a firm mutation testing system, with greater user control over the nature of result comparison, are discussed. Such a system is currently under development as part of an interpretive development environment.

1. Introduction.

Mutation testing is akin to the technique of fault simulation [8] used in the design and evaluation of circuits. This technique is well established in the hardware community, but use of the software analogue has been hindered by its perceived expense and also, no doubt, by a shortage of commercially available tools. The underlying philosophy of mutation testing is the assumption that experienced programmers often write almost correct programs. Hence by concentrating on small perturbations of the original program it is hoped that any surviving errors will be discovered.

The idea of **strong mutation testing** [1,2,3,4] is to make many small changes, one at a time, to a given program. Then an attempt is made to provide test data which distinguishes each of these so-called 'mutants' from the original program. If a mutant gives a different outcome from the original with some test data, it is said to be **dead**. Since it has been distinguished from the original it need no longer be considered. If on the other hand, a mutant gives the same outcome as the original with the test data, then it is said to be **live**. Further investigation should be able to reveal whether the test data could be enhanced to 'kill' this live mutant or whether no test data could ever be constructed to kill the mutant. In the former case the original test data can be considered to have been inadequate and hence has been improved by the mutation test. In the latter case the mutant program is 'equivalent' to the original program. A major disadvantage of strong mutation testing is the potentially large number of mutants that could be constructed. Since each of the many mutants needs to be executed to completion with test data in order for a comparison to be made between the outcome of the mutant and the outcome of the original, the method is computationally expensive. Partly as an answer to this problem, the idea of weak mutation testing was proposed.

The fundamental concept of **weak mutation testing** [7] is to consider elementary program components and simple errors in them. For example a simple error in a variable reference component might be to reference the wrong variable. One then seeks test data such that the component would give a different outcome on *at least one execution* of the component. For the case of a variable reference, the value of the variable needs to be different from the values of all other variables at that point in time, for the wrong variable mutant to give a different outcome. Weak mutation is not so expensive for two reasons. First, the necessary conditions for test data to 'kill' mutant components can be determined in advance, so in fact mutants need never be constructed explicitly. Secondly, since individual executions of components are being considered rather than the complete execution of the entire program, many mutant components can be considered in the same program test run. A disadvantage of weak mutation is that test data may be such that it satisfies the weak mutation testing criteria for a certain component yet under strong mutation of the same component the test data is inadequate.

† The term firm mutation was in part inspired by the sequence of syntactic positions in Algol 68, namely: strong, firm, meek and weak.

TH0225-3/88/0000/0152$01.00 © 1988 IEEE

In the next section the notion of **firm mutation testing** is introduced as an intermediate form of mutation testing between the existing techniques of weak mutation and strong mutation.

2. Firm mutation testing.

The basic idea of mutation testing in all forms is to make a small change in a program and compare outcomes of original and changed versions. Yet, within this framework there are a number of factors that may vary. These can affect the way the technique is applied and the conclusions that can be drawn. Consider statements such as those inside loops that may be executed more than once in a test run. If a change in such a statement is allowed to operate on at least one execution, but possibly not every execution, the following parameters may be altered and may give rise to different mutation testing results:-

(i). t_{change} , the stage in the program execution at which the change is made,

(ii). t_{undo} , the stage in the program execution at which the change is reversed and outcomes are compared,

(iii). the precise nature of what is being compared.

It is important to note that, in what follows, it is considered that at stage t_{undo} , not only is the mutant change reversed, but also the effect of executing with the change is reversed. That is, execution continues from t_{change} with the original unchanged version. The mechanism by which this might be achieved is discussed later. The reason for requiring it is the desirability of a framework which enables more than one such change with a short lifetime to be made independently in the same test run.

Weak mutation corresponds to the situation where t_{change} and t_{undo} are immediately before and after each single execution of a component. Strong mutation corresponds to the situation where t_{change} and t_{undo} are before and after execution of the entire program. Hence there are a range of possible alternatives where the duration of a change as specified by t_{change} and t_{undo} , consists of some proper slice of the program execution but at least as long as the execution of a single statement. There would be little point in selecting an execution slice which did not include execution of the statement under change. Firm mutation is thus the situation where a simple error is introduced into a program and which persists for one or more executions, but not for the entire program execution.

To specify t_{change} and t_{undo} it is necessary to relate them to positions in the program text and indicate details of which execution is being referred to, if more than one is possible. In order to resolve problems of result comparison, as explained later, it is envisaged that in firm mutation the positions of t_{change} and t_{undo} will bracket a reasonably self-contained region of program text. The specification of the time span for the change could be stated in a variety of ways, either in terms of component execution counts or in terms of values of data objects in the program. Although it would be possible to compare outcomes at some point other than t_{undo} , there would seem to be little merit in the extra complication that ensues. However, a viable alternative to having some fixed point t_{undo} , would be to perform continuous monitoring of the program state after t_{change} and then undoing the change at what would be the first deviation from the execution of the original. There is also the possibility that t_{undo} is never reached if the mutation introduces an endless loop or infinite recursion for example. In such a case first deviation from the original execution or an alternative such as expiry of some appropriate time slice may be essential to distinguish outcomes.

The next section is devoted to a discussion concerning the entities that may be compared in mutation testing. Then section 4 gives an extended example for illustrative purposes and section 5 deals with some practical considerations for implementing firm mutation.

3. Dead or alive?
Dependency on the entities being compared.

When attempting to provide automated assistance for the strong mutation testing approach, a problem arises in how to decide whether a mutant is dead or live. In [4] DeMillo et al say a mutant is live if it "*gives the same results*" as the original. Similar phrases have been used elsewhere and also indeed earlier in this paper. What entities should be compared to decide if the result is the same?

In many ways the simplest option is to capture in a file the sequence of characters written to output devices and then do a comparison between the output file of the original and the output file of each mutant. Lipton et al [9] recognised that a character by character comparison might in some circumstances be too stringent. They proposed also the alternative of comparing only non-blank characters.

This concentration on the output to compare outcomes soon reveals inadequacies in the approach. Consider, for example, the extreme, and admittedly unlikely, case of a program which produces no output. All of its mutants would remain live. At the other extreme, with sufficient appropriate print statements inserted, every mutant output can be made to appear different from the original and hence dead. All of this means that mutants that get to the correct final result by a different route from the original or perhaps with a redundant computation can be classified as either dead or alive, depending on the fineness of detail in any intermediate printing.

Example 1:

Consider for example the Pascal version given in Figure 1 of the Fortran subroutine used by DeMillo et al in [4], which determines the index of the first occurrence of a maximum element in a one-dimensional integer array.

```
PROGRAM max_index ( input, output );
CONST n = 10;
TYPE  index = 1..n;
VAR   j, k, imax : index;
      a : ARRAY[index] OF INTEGER;
BEGIN
   FOR k := 1 TO n DO read ( a[k] );
   imax := 1;
   FOR j := 2 TO n
   DO BEGIN
      IF  a[j]  >  a[imax]
      THEN imax := j;
      writeln ( 'First', j, ' elements scanned - ',
         ' index of first maximum is', imax )
   END;
   writeln ( 'All elements scanned - ',
      ' index of first maximum is', imax )
END.
```

Figure 1. Pascal program 'max_index'.

The mutant with the loop index starting at 1 instead of 2 merely performs one extra redundant comparison of a[1] with itself. Since the final outcome, as viewed by the value of imax is unchanged, one would expect to classify this mutant as live and in fact equivalent to the original. However, the effect of performing the extra time around the loop with the write statement inside the loop would be to change the overall output and any automatic output comparison would classify the mutant as dead.

In weak mutation notional changes are made to components and test data is required which would give a different outcome to the component when the change is present from the outcome of the same component without the change. The emphasis is entirely on components, which may be but a small part of a program statement or construct. Hence a mutated component may indeed give a different outcome to the component on one execution, but the net result of the statement as a whole on the same execution may not be changed.

Example 2:

x := a**2 + b**2

Suppose on one execution of this statement a has the value -1 and b has the value +1 and both of these values are different from the values of all other variables at the current instant. Then the weak mutation criterion is satisfied for the references to both variable a and variable b. However, in fact the two mutants obtained by replacing a by b and vice versa will both store the value of 2 in the variable x, which is the same as the value stored in the non-mutated version.

Example 3:

IF a < b THEN ...

Suppose on one execution of this statement a has the value +1 and b has the value +2 and both of these values are different from the values of all other variables at the current instant. Then once again the weak mutation variable reference criterion is satisfied for both variable references, yet there may be many mutants obtained by replacing a or b by other variables for which the predicate outcome is true and so the THEN clause will be executed, exactly as in the non-mutated version.

Example 4:

c := array [i] + array [j]

In this expression the variable array indices i and j may have values different from each other and indeed all other variables on one execution. However, the values of array[i] and array[j] may in fact be identical. This means that the mutants obtained by replacing i by j and j by i would make no difference to the value of c from that of the non-mutated version.

These small examples have shown that the looseness of the definition about what entities are to be compared in strong mutation may lead to ambiguity in the decision about what is live and what is dead. On the other hand, although weak mutation prescribes exactly what shall be compared, namely the outcome of the component being notionally mutated, there are many common circumstances when a different component outcome would not affect the outcome of the statement in which the component is embedded, let alone any global behaviour of the module or program.

4. Firm mutation example.

Consider the Pascal selection sort program given in Figure 2, when run with the test data set having n = 4 and array a = (1, 0, 1, 2). Note that the program sorts integer array a into descending order. For each of i = 1, 2, ..., n-1, it finds the index of the first maximum in the current array slice a[i..n] and then swaps the maximum with element a[i]. The state of the given test data array as the program proceeds is given in Table 1.

```
PROGRAM sort (input, output);
CONST n = 4;
TYPE  index = 1..n;
VAR   i, j, k, imax : index;
      temp : INTEGER;
      a : ARRAY[index] OF INTEGER;
BEGIN
   FOR k := 1 TO n DO read ( a[k] );
   FOR i := 1 TO n-1
   DO BEGIN
      imax := i;
      FOR j := i+1 TO n
      DO BEGIN
         IF  a[j]  >  a[imax]
         THEN imax := j
      END;
      temp := a[i];
      a[i] := a[imax];
      a[imax] := temp
   END;
   writeln ( 'Sorted array is:' );
   FOR k := 1 TO n DO write ( a[k] )
END.
```

Figure 2. Pascal program 'sort'.

Table 1. Test data array as 'sort' proceeds.

	a[1]	a[2]	a[3]	a[4]
Initially	1	0	1	2
After i=1	2	0	1	1
After i=2	2	1	0	1
After i=3	2	1	1	0

In this section four different ways of applying the mutation operator which replaces the only occurrence of a[j] by j in the predicate of the conditional, will be considered. The four ways correspond to selecting the positions of t_{change} and t_{undo} to encompass four different components, all containing the change, but of gradually increasing scope. Just for simplicity in this example, the times at which t_{change} and t_{undo} are considered to operate, are each entry into the chosen component, and each subsequent exit from it respectively. In other words, individual executions of the chosen component are considered. In each case the outcome of the original is compared with what the outcome would be for the mutated component.

Component 1:

The variable reference a[j] is treated as a component entity on its own with its outcome being none other than the value of the item referenced.

Original	Mutant
a[j]	j

There are six executions of the component and on each of these the value of the component with the change would be different from the value of the component without the change as detailed below.

Execution		Component outcome		Live/Dead
		Original a[j]	Mutant j	
i=1	j=2	0	2	dead
i=1	j=3	1	3	dead
i=1	j=4	2	4	dead
i=2	j=3	1	3	dead
i=2	j=4	1	4	dead
i=3	j=4	1	4	dead

Component 2:

The component selected is the entire conditional clause so the original and the mutated components are as follows:

Original	Mutant
IF **a[j]** > a[imax] THEN imax := j	IF **j** > a[imax] THEN imax := j

The outcome of this component is considered to be the value of imax. As has been seen the value of j would be different from a[j] on each of the six executions. However, the value of imax under the mutation would be the same as the original on three executions as shown below.

Execution		Component outcome		Live/Dead
		Original imax	Mutant imax	
i=1	j=2	1	2	dead
i=1	j=3	1	3	dead
i=1	j=4	4	4	live
i=2	j=3	3	3	live
i=2	j=4	3	4	dead
i=3	j=4	4	4	live

Component 3:

The component selected is the entire loop with index j, so the original and mutated components are as follows:

Original	Mutant
FOR j := i+1 TO n	FOR j := i+1 TO n
DO BEGIN	DO BEGIN
IF **a[j]** > a[imax]	IF **j** > a[imax]
THEN imax := j	THEN imax := j
END	END

Once again the component outcome is the value of imax, but this time after execution of the entire loop. There are three executions of this component corresponding to i = 1, 2, 3. The first and last of these would give the same final value of imax when mutated.

Execution	Component outcome		Live/Dead
	Original imax	Mutant imax	
i=1	4	4	live
i=2	3	4	dead
i=3	4	4	live

Component 4:

The component selected is the entire main loop with index i.

Original	Mutant
FOR i := 1 TO n-1	FOR i := 1 TO n-1
DO BEGIN	DO BEGIN
imax := i;	imax := i;
FOR j := i+1 TO n	FOR j := i+1 TO n
DO BEGIN	DO BEGIN
IF **a[j]** > a[imax]	IF **j** > a[imax]
THEN imax := j	THEN imax := j
END;	END;
temp := a[i];	temp := a[i];
a[i] := a[imax];	a[i] := a[imax];
a[imax] := temp	a[imax] := temp
END	END

The component outcome is the final contents of array a which would be different as indicated below if the mutant were executed.

Execution	Component outcome		Live/Dead
	Original a	Mutant a	
i=1,2,3	2 1 1 0	2 1 0 1	dead

The mutation results for the four components which have just been considered, are summarised in Table 2. Firm mutation of the first component is rather similar to, though not quite the same as, weak mutation. The distinction is that for a variable reference to satisfy the weak mutation testing criteria there should exist at least one execution of the component for which **all** variable replacements would give a different outcome. Situations 2 and 3 are clear examples of firm mutation and situation 4 is basically strong mutation of the entire program.

It can be seen from this example that firm mutation does indeed bridge the gap between weak and strong mutation and that the same change may remain live under firm mutation yet under both weak and strong mutation, the change is dead with the same test data. The converse is also true, though not demonstrated by this example, namely that a change may be dead under firm mutation testing but live under weak and strong mutation with the same test data.

5. Firm mutation practical considerations.

There seem to be several approaches which a firm mutation testing aid might employ in its implementation. One technique might be to create a physical copy of the selected region with the change and execute this in parallel with the original. A rendezvous would be required on exit from the region so that comparison of outcomes could take place and the result be recorded. However, in this way, an original test execution could proceed almost unimpeded. An alternative technique would be to execute the component with the change, save the outcome, perform reverse execution to the start of the component, execute the original without the change, compare outcomes and continue normal execution. The reverse execution could be performed by restoring the state of the program to that saved at point t_{change}. Such a scheme could be handled most naturally in an interpretive environment where t_{change} and t_{undo} would in many ways resemble traditional debugging breakpoints. This contrasts with the 'batch' mode of operation to be found in the mutant execution phase of several mutation tools which assist either with strong mutation testing [3] or weak mutation testing [5].

In an interpretive environment, firm mutation would offer the user a greater degree of control in a number of ways including:

(i) the code selection process,

(ii) result comparison,

(iii) mutant operator selection.

Each of these is now considered in turn.

Table 2.
Summary of firm mutation outcomes.

Execution		Component			
		a[j]	IF a[j] > a[imax] THEN imax := j	FOR j := i+1 TO n DO ...	FOR i := 1 TO n-1 DO ...
i=1	j=2	dead	dead		
	j=3	dead	dead		
	j=4	dead	live	live	
i=2	j=3	dead	live		dead
	j=4	dead	dead	dead	
i=3	j=4	dead	live	live	

5.1 The code selection process

Traditionally strong mutation has been applied to a whole program or to a module (SUBROUTINE or procedure) within a program. The mutation of modules provides a clue as to how to tackle smaller structural areas. By considering a selected portion of a program as a **headerless procedure** it is possible to limit mutations to the selected area. Selectable regions can be any program structure or sequence of structures which, when surrounded by an imaginary **begin** - **end** pair, can be thought of as forming an inline procedure. Static data flow analysis can be used to construct a genuine procedure header detailing the interface between it and the surrounding area of program. Values for input parameters could be provided as test data, if firm mutation of the component is being performed in isolation. Values of output parameters can be used for result comparison.

It should also be possible to create subareas within the selected area that are **mutation immune.** These correspond to sections which have been tested thoroughly and which are masked out for the current mutation activities. The main benefit of selectable program areas for the mutation process is that mutation testing can proceed on partially completed programs or modules. That is to say mutation testing becomes part of the development process.

5.2 Result comparison.

Once a region of program text has been selected and the interface has been determined, those objects that are either *defined* or *referenced and defined,* can be used in the result comparison. For example, when the 'FOR j' loop is selected in the sort example program used in section 4, the input and output parameters are as follows:

input: i, imax, n, a

output: imax

The variable imax being referenced and then defined appears as both an input and an output parameter. Being the sole output parameter, it alone is used for outcome comparison to decide whether an execution of the component is live or dead.

It may be that the user may wish to select just a subset of output parameters for result comparison. For example, when the 'FOR i' loop is selected in the sort program, both of the variables temp and imax are defined objects, but both can be considered as local to the loop and hence not needed in the result comparison. Alternative candidates for result comparison could include:

- data definition trace,
- data reference and definition trace,
- control flow trace,
- actual physical output,

or perhaps a combination of these. The trace output comparison may be useful for giving a clearer understanding of exactly how the execution of the mutant differs from the original. However, it gives more scope for distinguishing between mutant and original and so, in that sense, may be less interesting. A very similar point has been made by Hamlet [6] in describing a testing system to determine expression substitutions which are 'simpler' than the original. He stated that few live substitute expressions are found which surprise the programmer, when exact equivalence of detailed runtime history is demanded.

5.3 Mutant operator selection.

It would be possible to guide the user in what mutant operators to select based upon frequency of applicability in the selected region or the range of influence of a change. Further restrictions which the user might enable are those which depend upon scope. In variable replacement, for

example, the user might wish to state whether a substitution candidate must be chosen from the selected region, an enclosing range or the global range.

6. Conclusions.

When the technique of weak mutation testing was proposed, its shortcomings were explicitly pointed out. Basically, what is worrisome with weak mutation is that different components (data access, data store, relational expression etc.) of the same program statement or construct can give different outcomes from the original on different executions when considered under the notional mutant change, yet somehow events can combine to give the overall correct outcome to the statement concerned or indeed to the entire program execution. Hence to have achieved complete weak mutation testing can give a false sense of security. Although strong mutation does not suffer this problem, it does insist on complete program executions and hence is expensive to perform.

The advantages of firm mutation over weak and strong mutation can be summarised as follows:

- Firm mutation is more transparent than weak mutation and can insist that mutants cause a distinguishable difference at the statement level. Since actual changes are made to program statements it is easier to identify what mutants have been attempted and what difference in outcome resulted.
- Firm mutation is less restricted than weak mutation in terms of mutation operators and components to which they may be applied. As yet, weak mutation has been concerned solely with certain fundamental low level components.
- Firm mutation is potentially less expensive than strong mutation in that partial executions of mutated components are performed rather than complete executions.
- Firm mutation provides more control than both weak and strong mutation over result comparison. The user can in effect specify what shall be deemed sufficient to kill a mutant.

Of course there are disadvantages with firm mutation. These include:

- It is not easy to relate what has been achieved in deriving test data which kills mutants using firm mutation testing, to either weak or strong mutation test data adequacy.
- There is no obvious systematic basis on which to select the areas of program code for firm mutation testing.

Nevertheless, firm mutation, as proposed in this paper, does combine the best aspects of weak and strong mutation. It can be viewed either as strong mutation, but using any partial execution of program components so that many mutants can be considered in the same test run, or as weak mutation, but using components with more extensive scope. It is difficult to imagine firm mutation being applicable in anything other than an interpretive system. The authors are currently building facilities of the sort described into a comprehensive programming environment. It is hoped that, once experience has been gained, that the importance of the advantages and disadvantages can be assessed.

Acknowledgement.

K.Halewood wishes to acknowledge the United Kingdom SERC for the award of a Research Studentship.

References.

[1] T.A.Budd, R.A.DeMillo, R.J.Lipton and F.G.Sayward, "Theoretical and empirical studies on using program mutation to test the functional correctness of programs", *7th ACM Symposium on the Principles of Programming Languages,* Las Vegas, Jan. 1980.

[2] T.A.Budd, R.J.Lipton, R.A.DeMillo and F.G.Sayward, "Mutation Analysis", Technical Report No. 155, Department of Computer Science, Yale University, April 1979.

[3] T.A.Budd, R.J.Lipton, F.G.Sayward and R.A.DeMillo, "The design of a prototype mutation system for program testing", *National Computer Conference,* AFIPS Proceedings, Vol. 47, pp. 623-627, 1978.

[4] R.A.DeMillo, R.J.Lipton and F.G.Sayward, "Hints on test data selection: help for the practicing programmer", *IEEE Computer,* Vol. 11, No. 4, pp. 34-41, April 1978.

[5] M.R.Girgis and M.R.Woodward, "An integrated system for program testing using weak mutation and data flow analysis", *Proceedings of the 8th International Conference on Software Engineering,* pp. 313-319, IEEE Computer Society Press, London, August 1985.

[6] R.G.Hamlet, "Testing programs with the aid of a compiler", *IEEE Trans. Software Engineering,* Vol. SE-3, No. 4, pp. 279-290, July 1977.

[7] W.E.Howden, "Weak mutation testing and completeness of test sets", *IEEE Trans. Software Engineering,* Vol. SE-8, No. 4, pp. 371-379, July 1982.

[8] Y.Levendel and P.R.Menon, "Fault simulation", Chapter 3, pp. 184-264 in *"Fault-tolerant computing: theory and techniques"*, Vol. 1, Editor D.K.Pradham, Prentice-Hall, 1986.

[9] R.J.Lipton and F.G.Sayward, "The status of research on program mutation", *Digest for the Workshop on Software Testing and Test Documentation,* Fort Lauderdale, Florida, pp. 355-372, Dec. 1978.