A Practical Method for Software Quality Control via Program Mutation

D.Wu  
M.A.Hennell  
D.Hedley and  
I.J.Riddell

Department of  
Department of  
Liverpool Data  
Computer Science,  
Statistics &  
Research Associates,  
East China University  
Computational  
Liverpool, England  
of Chemical  
Mathematics,  
Technology,  
University of  
Shanghai, China  
Liverpool, Liverpool,  

Abstract

Program mutation is a suitable technique for investigating software reliability and quality control since it is able to detect many potential errors. However it is necessary to improve the technique for industrial practice. A new method of program mutation is presented here which increases the feasibility, effectiveness and efficiency of searching for those errors which have escaped from the activities ofAfterers and competent programmers. It is based on syntax direction and it is aided by the language semantics. This means that the scope of a program mutation (i.e. the mutation rules of the method), and its corresponding mutants, are rigorously directed by a syntax and related semantics as defined by the tester. A paradigm for the mutation syntax and semantics when limited to boolean expressions and the corresponding test coverage metrics in terms of this method are given in the paper.

Introduction

With the rapid progress in techniques for large scale integrated circuits and information processing, software systems have become more sophisticated and expensive. Most of them are the result of collaboration between many analysers, designers and programmers for a number of years. Nobody can master the whole of the software system very clearly. Consequently, a proper criterion for quality can be reached only by using effective methods of management and control.

There are three categories of quality control in software production derived from three different views as follows:

1) requirement or function based.
2) structure based.
3) fault or error based.

A requirements specification to control the quality of software production, including reliability and performance is basically the same as in any kind of production. However software is a mental product, it acts on the basis of the knowledge and techniques of human beings. In general the requirements and functions are rather sophisticated, in some sense they manifest a kind of mechanical intelligence. Therefore, with powerful software it is difficult to produce a precise and formal requirements specification. If the software is tested against intuitive requirements, it is observed that only 50–80% of the program code is executed. Therefore quality control based on requirements or functions is primarily utilised in the phases of systems analysis and design in the software life cycle. Control can be exercised either in a manual manner, as in desk checks and walkthroughs, or sometimes in an automatic way, e.g. specifying the specification in PSL or SREM [2].

In contrast quality control based on software structure has been developed systematically. In order to achieve reliability a variety of test methods are available in the coding and test stages of the software life cycle, e.g. statement testing, branch testing, interior testing and path testing. By systematically monitoring execution and collecting test data, important indicators of reliability, e.g. test coverage metrics, can be obtained to control software quality. Whilst it is conceptually advantageous, path testing is infinite and
therefore infeasible in general. The LCSAJ test method is a reasonable compromise to avoid the infeasible path problem [3]. Even if path testing has been utilised in favourable situations, some software faults still may not be discovered.

The third method of software quality control is based on revealing various covert faults or errors. Revealing such errors is often recommended, using for example, boundary testing or stress testing, but it is difficult in practice to provide a corresponding systematic method for quality control. In fact such tests are randomly executed by personal experience and intuition. The idea of this method is especially appealing for some life critical applications. It is also of great benefit for application in the acceptance and trial run stage of the software life cycle for validating the reliability in some extreme or odd cases. The purpose of this paper is to present a basis for a new systematic method of this type.

PROGRAM MUTATION

Program mutation was presented by R.A.Demillo, R.J.Lipton and F.G.Sayward [4]. In principle it is capable of revealing any potential simple errors in programs. Mutation systems have been used as powerful experimental tools for research on software testing, but they have not been widely applied in industrial validation due to their high cost.

If a program has already been tested by another method, then program mutation may not need any additional sensitive test data to check the program's reliability, i.e. the existing data may be adequate. After executing all the modified programs (i.e. mutated programs) with the same set of test data, if different (wrong) results appear, then we have more confidence in the program, since the test data is sensitive enough to detect these seeded errors. Otherwise, if the results are the same as the original for any mutants, then a suspicious point has been found. It may be that an equivalent program exists or a covert fault in the program has been found or the test data is inadequate. However, the possibilities (i.e. mutants) for realistic programs are effectively infinite in extent. If and only if all the possible mutants of a program have been detected, can one be sure that the program contains no errors of commission. Obviously, it is infeasible to generate all possible errors. To reduce this infinity, R.A.Demillo etc. present an "experimental principle": the error coupling effect: "test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it all also distinguishes more complex errors." [4] Hence only simple errors need to be generated and detected.

The extent of the correctness of this hypothesis is not proved and will not be discussed here. But we can still make use of the concept of a simple error, and no doubt during the process of revealing all the simple errors we may get some inspiration for the complex errors. The number of possible simple errors in a program, (i.e. the number of mutants) is still prohibitively large. It increases roughly with the square of the program length. The scope of the mutation and the set of mutation rules (or mutant operators, mutation transformations in some papers) are critical to the method's capability for error detection, efficiency and feasibility. Initially there were no constraints on using mutation techniques. A tester could modify, delete and insert almost any components of a program from a single character, e.g. a typographic error, to a whole sentence, e.g. a RETURN statement. Afterwards some mutation systems [10][11][12] with a set of specified mutation rules including variable mutation were presented, and the mutation rules were defined with some consideration of syntax.

The weak mutation schema [5] as presented by W.E.Howden focuses on the resulting value delivered by specified computational components. The eventual results of the whole program are not considered in weak mutation. The method improves the efficiency of mutation dramatically, because "it is often possible to test all of the application of a mutation transformation to all applicable program components with a single test" [4]. A series of important results were inferred showing promise for its industrial application, but as indicated by Howden, the problem of mutation efficiency is still blocking its application to variable mutation. For instance, the N^2 mutants of variables in N lines of source code is "prohibitively inefficient to monitor whether or not the data over which a component has been executed distinguishes the component from all of its possible mutations" [5], for most test cases. So a limited mutation schema with a particular subset of lexemes has been proposed in [6][7], e.g. the set of six relational operators. Combining the above ideas into a general strategy gives a method known as Syntax Directed and Semantics
Aided Mutation, abbreviated to SDSAM in the context of this paper. By means of SDSAM an individual or group can define their own mutation grammar using a set of productions of a context free syntax or exploit a paradigm of SDSAM presented in this paper.

Syntax Directed and Semantics Aided Mutation can be a practical test approach or a quality control method for software production. It consists of the following four points: mutation syntax, additional mutation rules, the partition principle of limited mutation, and a set of criteria for an effective SDSAM system. A paradigm for the mutation syntax and semantics, limited to boolean expressions, and the corresponding test coverage metrics are given in this article and analysed in [8] to demonstrate their great benefit to the feasibility and efficiency of program mutation.

**MUTATION SYNTAX**

In the SDSAM approach the specification of a mutation is described by the mutation syntax. According to weak mutation, suppose that P is a program, that C is a component of P, and there is a mutation applied to C to produce C', then we have a mutated program P'. With the SDSAM any component to be mutated must be derivable from the specified mutation syntax and conform to the mutation rules defined by the mutation syntax and certain semantics. For instance, reference to a variable might be a mutation component, but in an SDSAM system we can apply the restriction that only those references to variables located in boolean expressions will be mutated. Moreover it can be specified that only those variables appearing in control flow elements (i.e. predicates) would be the candidate mutants.

The mutation syntax consists of a limited set of mutation productions compatible with the syntax and semantics of the programming language in which the program being tested is written. The productions given will restrict the program mutation to a strictly specified region. Consider a grammar G of a programming language L:

\[ G = (S, N, T, P) \]

- **T** -- set of terminators, e.g. a, b, c,... with characters in lower case,
- **P** -- set of productions in the following form.

\[
A ::= BC \mid aB \\
B ::= a \mid b \mid c \mid...
\]

The mutation grammar might then be defined as:

\[ G_m = (S_m, N_m, T_m, P_m) \]

- **G** -- mutation grammar defined by tester,
- **S_m** -- start symbol for mutation, e.g. <expression>,
- **N_m** -- set of nonterminals of mutation, e.g. A_m, B_m, C_m, ...
- **T_m** -- set of terminators of mutation, e.g. a, b, c,...
- **P_m** -- set of mutation productions in the following form.

\[
A_m ::= B_m C_m \mid aB_m \\
(1) \\
B_m ::= a \mid b \mid c \mid ...
(2)
\]

In productions above the symbols A_m, B_m and C_m are nonterminals, and symbols a, b and c are terminators.

The form (1) of the mutation syntax will define the scope of the mutation, i.e. which of the tokens contained in a program would be mutated. The form (2) is the major part of the definition of the mutants (mutation rules). In order to restrict the mutation and prevent syntax errors, we must have: \( N_m \in N \), \( T_m \in T \), \( P_m \in P \), \( S_m \in S \cup N \). Therefore \( G_m \in G \), that is the mutation grammar \( G_m \) is a subset of the grammar \( G \). If a program is syntactically correct in the grammar \( G \), then each of the mutant programs derived from \( G_m \) must be valid in \( G \). For example:
<relational operator> ::= = | ≠ | > | < | >= | <= (3)
<relational operator> m ::= = | ≠ (4)

(3) is the production for the relational operators of the grammar for L, and (4) defines the corresponding mutation production and a mutation rule: "= ↔ ≠". It means that all relational operators '=' in a program would be mutated to '≠', and vice versa. The mutation production (4) is a subset of the original productions (3).

MUTATION RULES

Many of the mutation rules can be derived from the mutation syntax, mainly those of the form (2): B m := a | b | c... Suppose that the B m is a nonterminator of the mutation, and a, b, c, etc., represent tokens and related mutants, e.g. the mutation rule "θ ri ↔ θ r j" is based on the following productions:
<relational operator> ::= = | ≠ | > | < | >= | <=
θ ri θ r j -- any relational operator of ( '=' , '≠' , '>' , '<' , '<=' , '>=' ).

Sometimes a large set of mutants is possible, so an additional restriction is needed, e.g. we would like a simple variable to be mutated only to certain others, perhaps those identifiers differing by only one character. Ordinarily, the set of terminators of G m is defined as all valid tokens of the language. In practice the set of tokens is a small set of strings which have been declared and are meaningful in the programs (T m ∈ T). In terms of the restriction above the additional mutation rules based on the semantics will confine the T m to a much more limited scope T m ′, where T m ′ ∈ T m.

According to the definition of the mutation syntax mentioned above G m must be a limited and compatible subset of a programming language's grammar. Furthermore, the semantics of the mutants can be used to restrict the generation of those mutants. The mutation rules originating from the mutation syntax and the mutant semantics are the essentials of SDSAM. The advantages of SDSAM are: a very flexible choice for executing any kinds of mutation and more effective mutation through not generating unnecessary, invalid and absurd mutants. The strategy of SDSAM will force the mutation into a limited and more promising region.

PARTITION PRINCIPLE OF SDSAM

As a general guideline for setting up a mutation strategy for SDSAM, the following principle is advocated:

If an infinite set or a great many mutants are generated by a given mutation rule then partition that rule into several rules in terms of various factors. A factor is an additional criterion which selects a subset of the class of mutants. Let an offset be the 'difference' between two mutants: such a difference can be applied either to the physical characteristics of the mutants, e.g. n characters differ, or to the abstract properties such as variable type. Such an offset can be used to specify a factor. If we select a set of mutants, each of which differs from the component being mutated by only one syntactic or semantic factor and the offset between them is always the minimum (i.e. only including the nearest neighbours within the rule), then that is termed the "minimum set" of mutants for the rule. Sometimes the minimum set is an effective set for testing, sometimes it is not (see discussion below). The partition principle can be applied in all situations. For example, variable mutation is one of the largest sets of mutations for a complicated program. In this case, the following partitions might be considered:

a. simple variable ↔ simple variable (of the same type)
b. subscripted variable ↔ subscripted variable
   (different arrays with same dimension, same index value)
c. subscripted variable ↔ subscripted variable
   (same array with index value differing by +1)
d. subscripted variable ↔ subscripted variable
   (same array with any different index value)
e. loop variable ↔ loop variable (of the same type)

f. field of record ↔ field of record
   (of same type in same record)

g. control variable ↔ control variable
   (of the same type and appearing in any boolean expression)

For example, let there be four arrays (vectors) A, B, C, and D in a program each with seven elements (subscripted from 1 to 7). If the original component being mutated in the program is D[3] then one set of elements worth mutating consists of { A[3], B[3], C[3], all the elements of D except D[3] }. In this case the mutation rule b) and d) above is utilised. The first factor is the subscriptor (or index), the second is the identifier of the arrays. In this example, the number of mutation rules is 2, the number of elements is 7, and the number of arrays is 4, giving the total number of mutants in the set 7+4- 2=9 [15]. This is reduced by a factor of three from the original 28-1=27 (no of arrays * no of elements - 1). By using semantic rules b) and c) together, we can obtain the minimum set which consists of only five mutants: { A[3], B[3], C[3], D[4], D[2] }, decreased by a further half. Various choices can be selected with the aid of specific semantics.

In summary, the simple error and simple mutants defined in SDSAM, with the use of factors, is more efficient in normal validation circumstances and more feasible in practice than conventional mutation methods. It may however miss some errors detectable only by using multiple factors, since it is impossible to uncover a complex error which is not coupled to a simple one.

MUTATION OF BOOLEAN EXPRESSIONS

As program errors are not evenly distributed within the code so mutation testing ought not strive to search for all possible simple errors. From the point of view of testing there are three interesting categories of error: predicate error (path selection error), missing path error and computation error. The path selection error appears in the control flow part of a program, the computation error occurs in the data processing part, and the missing path error can occur anywhere. It has been shown that simple faults in arithmetic expressions are relatively easy to detect [9] and if a program has passed some systematic testing process, e.g. branch testing, then it is unlikely that a significant number of computation errors still exist.

It seems more probable that path selection errors and some missing path errors might be retained after traditional testing. A boolean expression only has two values (TRUE and FALSE), therefore the probability of coincidental correctness is more likely than in the data processing part, especially when using only a few test data values. Some relational operators overlap each other semantically, e.g. "X<5" with "X=5". These are considered to be the most interesting for mutation. If test data for "X>5" is found to be absent in the set of data for the predicate X<5 then an error possibility (missing path error) is discovered. If only the path "X<5" has been executed then there may be a path selection error ('>' should possibly be '<').

Sometimes, when a control flow variable is erroneously processed in a data computation, an attempt to discover errors by the mutation of that same variable in a boolean expression might be a heuristic to discovering the data computation error. Hence, the coverage of the mutants of the boolean expression may have a rather broad scope. A full investigation of mutation analysis for boolean expressions has been presented in [8]. The research results are rather encouraging due to the minimum set of mutants for the whole boolean expression (including boolean operators) being small, and hence feasible for industrial practice. The number of variables involved in control flow is usually much less than in data computation, so the problem of combinatoric explosion in variable mutation is greatly reduced.

The mutation analysis of SDSAM can be applied to the data processing part and any other components of a program. However, the mutation of boolean expressions would be a basic part of an effective SDSAM system. The complete mutation syntax and mutation rules for boolean expressions are described as follows:

a. mutation syntax.

<boolean expression> ::=<boolean term>
   OR<boolean expression>
   | <boolean term>

<boolean term> ::=<boolean element>AND
   <boolean term>
   | <boolean element>
TEST COVERAGE METRICS FOR THE MUTATION OF BOOLEAN EXPRESSIONS

Consider the relational expressions of a program in the form "X \Theta_c c" or "X \Theta_a kY \Theta_r c". Here X and Y represent arithmetic variables, c and k are arithmetic constants. The symbol \( \Theta_a \) represents an additive operator, i.e. '+' or '-', and the symbol \( \Theta_r \) in this form represents an arbitrary relational operator. More complex predicates can still be treated as the form "X \Theta_r c" but the X will represent an expression. The conventional test coverage metric for mutation (abbreviated to TCM) is as stated below:

\[
TCM = \frac{\text{number of dead mutants}}{\text{total number of mutants}} \times 100\%.
\]

This metric may also be applied to specific mutation subsets. For example, the TCM for a particular relational operator is:

\[
TCM_r = \frac{\text{no of dead mutants of that relational operator}}{5} \times 100\%.
\]

since there are only five possible mutants.

The TCM for a given variable is:

\[
TCM_v = \frac{\text{number of dead variable}}{\text{total no of variable mutants in the program}} \times 100\%.
\]

usually the choice would be variables of the same type.

The TCM for a given additive operator:

\[
TCM_a = \frac{\text{no of dead additive operator mutants}}{\text{total no of additive operator mutants}} \times 100\%.
\]

and the TCM for a boolean operator:

\[
TCM_b = \frac{\text{no of dead boolean operator}}{\text{total number of boolean operators}} \times 100\%.
\]

b. mutation rules.

\[\begin{align*}
\text{TRUE} & \leftrightarrow \text{FALSE} \\
B_i & \leftrightarrow B_j \text{ if } j \neq i \\
B_i & \leftrightarrow \text{NOT } B_i \\
\text{OR} & \leftrightarrow \text{AND} \\
\Theta_{ri} & \leftrightarrow \Theta_{rj} \quad (\Theta_{ri} \text{ and } \Theta_{rj} \text{ represent any relational operators}, \ i \neq j) \\
A_i & \leftrightarrow A_j \quad \text{if } j \neq i \\
c & \leftrightarrow c + k \\
+ & \leftrightarrow - \\
k_i & \leftrightarrow k_j \\
k_{1i}, k_{1j}, k_{1i}, k_{1j} & > 0
\end{align*}\]
The conventional test coverage metric TCM is not suitable when there are an effectively infinite number of mutants as can occur for constants and coefficients in boolean expressions. Some new test coverage metrics based on minimum test data sets for the mutation of boolean expressions [8] are suggested below. The definition of these new test coverage metrics is based on the concept of a minimum set of specified test data for killing all (of an infinite number of) mutants of a component. If the total (finite) number of test data points for this set is \( n \) then, for each point of the specified test data, let the maximum value of the test coverage metric be \( \frac{1}{n} \times 100\% \). In this paper a set of input values for a variety of variables in a test unit (program or module) which must be initialised for executing a test is referred to as one test data point. If a specified test data point cannot be constructed exactly then the approximation of the test data to the requested exact value should be calculated. For instance, according to [8] only two test data points are necessary to kill all the mutants of a constant in a relational expression: \( \exp \equiv c \). Each of the two test data points must be very close to the input data space border implied by the relational expression and on opposite sides, i.e. \( \exp = c \pm \varepsilon \). The symbol "\( \varepsilon \)" represents the smallest number which can be distinguished from unity of a particular type of number in a given computer. In practical validation it might be difficult to design a test data point to exactly satisfy these requirements. However the TCM of a constant for these kinds of relational expressions can be defined as the approximation of the test data to the requested exact value.

The TCM for a real constant \( c \) in a relational expression: \( x \Theta_a ky \Theta_r c \) can then be expressed as:

\[
TCM_c = \left( \frac{1}{1 + (x \Theta_a ky - c)_{\min}} \right)_{\text{when } x \Theta_a ky > c} + \left( \frac{1}{1 + (c - (x \Theta_a ky))_{\min}} \right)_{\text{when } x \Theta_a ky < c} \]

*50%* 

and the minimum is defined over all test data points on a particular side of the border.

TCM for a constant \( c \) of integer type in a relational expression:

\[
TCM_c = \left( \frac{1}{1 + (x \Theta_a ky - c)_{\min}} \right)_{\text{when } x \Theta_a ky > c} + \left( \frac{1}{1 + (c - (x \Theta_a ky))_{\min}} \right)_{\text{when } x \Theta_a ky < c} \]

*50%* 

The TCM for a constant is thus the sum of the TCMs on either side of the boundary defined by the predicate "\( x \Theta_a ky \Theta_r c \). The term \( (x \Theta_a ky - c)_{\min} \) of the first part of TCM represents the minimum value of "\( x \Theta_a ky \Theta_r c \) among all the executed test data points characterized by \( x \Theta_a ky > c \), i.e. above the border. The second part of TCM represents the test coverage metric for all the test data points beneath the border. The '\( x \)' and '\( y \)' are the actual numerical values for each test data point. The value "\( x \Theta_a ky - c \) is defined therefore as the 'relative distance' between a test data point \((x, y)\) on one side of the border and the inclusive border of the predicate "\( x \Theta_a ky \Theta_r c \). For example, consider there is a relational expression \( X + ky < c \) as a predicate in a program and a corresponding test data point \((x, y)\). Consider further, a straight line (dotted line in figure 1a) through the point \((x, y)\) parallel to the predicate border. The relative separation of these two lines will be defined as the distance between the two points defined by the intersected of these two lines with the X axis. TCM will be 1 when we have a test data point on the border and another one closely approaching the border from the exclusive side and it tends to 0 when the 'relative distance' of the test data points on both sides is very large. The operator '+' to conjoin the two independent metrics is chosen to be preferable because it does not produce as pessimistic a result as would a '*' operator.
points in the minimum test data set is defined as the mutation test coverage metric for that component.

The test coverage metric of a coefficient is an example of this type. Two test data points closely approaching the border are needed for killing all mutants of a coefficient in a relational expression [8].

As shown in figure 1b, the TCM for a coefficient of real type in a relational expression is defined in terms of the slopes of the border and line connecting the test data point \((x,y)\) to \((c,0)\) as:

\[
TCM_k = \left( \frac{|c-x|}{|ky|} \right)_{\min} + \left( \frac{|ky|}{|c-x|} \right)_{\min} \times 50% \\
(\text{when } |ky| > |c-x|) \quad (\text{when } |c-x| > |ky|)
\]

and the minimum is defined over all test data points on a particular side of the border.

It is shown in [8] also that two test data points are necessary to kill all the mutants of the coefficient \(k\) of a relational expression. These two test data points must be either on the border or approaching the border from a specified side. But these conditions are still not sufficient for killing all the mutants of a coefficient as discussed below.

It is proved in [8] that if the signs and values of \((c-x)\) and \(ky\) of a test data point are chosen as \((x_1,y_1)\) and \((x_2,y_2)\) in figure 1b then these test data points are completely ineffective for killing any mutants of a coefficient. In these cases the TCM of a coefficient will be equal to zero. Only those test data points within the shadow areas in figure 1 are effective in killing the mutants of a coefficient (the details are described in [8]). The TCM of a coefficient will be the sum of the TCM of two sides of the border of the predicate \(x \Theta_a ky \Theta_r c\). The
The TCM$_k$ for each side should be always less than 0.5, so it is easy to decide which expression in the above formula has to be used. The ratio of "k" and "kx" represents the $\gamma$ approximation of the border to the straight line connecting the test data point and intersection of the X axis with the border. This ratio represents the relative slope of these two lines.

If a comprehensive indicator of the TCM for the mutation of all components is needed, the arithmetic average value of all the individual TCMs listed above excluding the TCM of a variable can be utilized. A comprehensive TCM is suggested below which costs much less in terms of overheads during testing. The concept of a comprehensive TCM for a relational expression is similar to that of a constant:

$$\text{TCM}_R = \frac{\text{effectiveness no of executed test data point set}}{\text{total no of test data points in the minimum set}}$$

When more test data points have been executed, will the TCM$_R$ of a relational expression increase proportionally? It may and it may not. In order to kill all the mutants of any component in a relational expression each test data point in a minimum test data set must be of a different kind and specifically defined. They must be independent and orthogonal to each other. If two executed test data points are of the same kind and of the same or smaller calculated TCM value then the second is an ineffective test data point. With the mutation of several computational components the total number of mutants is often infinite, but there is always a finite number of test data points in a minimum test data set for the mutation of a boolean expression according to [8]. The total number of test data points in the minimum mutation set of all components of a relational expression is more suitable as a comprehensive measure of the test coverage for mutation. The TCM$_R$ for mutation will increase from 0 to 1 if the tests are always executed with specified test data derived from a minimum set. If more test data points are added from outside this set then the TCM will not increase. In terms of the ratio of the effectiveness number of the executed test data points and the total number of test data points in a minimum set, the test coverage metric precisely presents what percentage of the minimum set has been executed. This definition of the TCM$_R$ exactly indicates to what extent the mutants have been killed by the executed test data, no matter whether the number of mutants is finite or infinite.

For example according to [8] the minimum number of test data points is three for the predicate form "X $\Theta$ kY $\Theta_r$ c". One test data point is on the inclusive border, the second should be very close to the exclusive border and the third can be anywhere on the inclusive side of the border. Thus, a suitable set of test data $(x_1, y_1)$, $(x_2, y_2)$ and $(x_3, y_3)$ will guarantee the relational expression to be free from any simple error in the predicate "X $\Theta$ kY $\Theta_r$ c". The three test data points satisfying "$x_1 + ky_1 = c$", "$x_2 + ky_2 = c - \varepsilon$" and "$x_3 + ky_3 > c$" will kill the mutants of all components of the predicate "X + kY < c" under some simple assumptions [8]. Again the symbol "$\varepsilon$" represents a sufficiently small positive number. The detailed proof of the minimum set of test data for the mutation of all the components of a relational expression may be found in [8].

The comprehensive TCM$_R$ for a relational expression of a real type is defined as follow:

1) For the form "X $\Theta$ kY $\Theta_r$ c",

$$\text{TCM}_R = \left( \frac{1}{1+\frac{x}{a}ky-c} \right)_{\text{min}} + \gamma$$

Where $\gamma$ is for the test data on the special side of the border and is either 1 or 0 i.e. 1 if the test data is on the special side, 0 otherwise.
2) For the form "X a k Y r c",

\[
T_{CM_R} = \frac{1}{1+(x a_k y - c)_{\min}} + \frac{1}{1+(c - (x a_k y))_{\min}} + 2 \gamma
\]

\[
+ \frac{\left(\frac{\left|c - x\right|}{\left|ky\right|}\right)_{\min} + \left(\frac{\left|ky\right|}{\left|c - x\right|}\right)_{\min}}{12} \times 100% \]

\[
+ \frac{\text{number of dead additive operator mutants}}{\text{number of additive operator mutants}} \times 100\%
\]

The contribution \( \gamma \) is again for the test data on the special side of the border.

If a more complex relational expression appears in a program it will be dealt with in the same way as form 1).

The TCM of a boolean operator and a variable have to be calculated individually as \( T_{CM_b} \) and \( T_{CM_v} \). The test coverage metrics serve as a relative measure of reliability and a test criterion as well.

**COMPARISON WITH OTHER ERROR-BASED METHODS**

With each test approach there is a corresponding method for software quality control. Program mutation is an error-based test approach. Although error-based testing methods are usually designed to uncover specific classes of errors, e.g. boundary testing, program mutation is characterised by stimulating various elementary components of a program which may have any number of alternatives. The simplest and most reliable way to implement the method is by strong mutation but this will be at the greatest cost. It is still useful to use strong mutation as a powerful experimental tool when the theoretical analysis of mutation is undecidable and impossible, e.g. mutation of labels and repeated mutation within a loop statement [11].

The strategy of SDSAM based on weak mutation is much more efficient than strong mutation, and hence more adequate for industrial practice, especially with general forms of expression. It is characterised by being a formal and systematic way to limit the scope of mutation. It provides a flexible and feasible approach. Although it might be more limited due to the restriction of mutation syntax and semantics imposed by the tester, in practice it is probably more effective and efficient. If necessary, the users can make their own compromise between cost and reliability. A set of test coverage metrics is established based on SDSAM to evaluate the software testedness. Rigorous analysis directed by a mutation syntax has yielded a series of important theoretical results as reported in [8]. They provide a more accurate and complete view of mutation for boolean expressions reported previously in [5]. We will briefly describe two aspects of these results below.

1) The minimum set of test data points for the mutation of all components in a relational expression, e.g. X + k Y < C, is re-identified.

In [5] for the weak mutation of a relational expression it is suggested: "Two simple kinds of mutations will be considered for arithmetic relations: wrong relational operator and off-by-an-additive-constant" and "The data that are required to distinguish off-by-a-constant and wrong-relation mutations in a relational expression exp1 r exp2 can be combined by requiring the execution of the relation over data for which exp1 - exp2 = -E, 0, and +E".

According to our investigation [8] the set of test data suggested in [5] is complete for killing the mutants of an additive constant and the relational operators, but it might fail to guarantee the relational expression being free from other simple component errors, e.g. both additive arithmetic operator and coefficient. In order to kill only the mutation of an additive constant or a relational operator respectively two test data points each are enough [7], and in order to kill both of them three test data points are required, one of the test data points "+E" which is very close to the inclusive border can be changed to another one "<c" or ">c" [8].
It is much easier to generate a test data point within a broad area like "<c" or "+c" than very close to the inclusive border.

2) The exhaustive test set for a boolean expression is not necessarily that which is implied in [5]. The mutation of a complex boolean expression is a rather sophisticated problem because it is the top nonterminator of the computational components. All possibilities have to be considered. This means that the problem of all the combinations of the mutations of the six elementary components (arithmetic variable, relational operator, additive constant, additive operator, coefficient and boolean operator) should be solved. The detailed analysis of the mutation of a boolean expression in [8] has proved that the minimum number of test data points for killing all the mutants in a boolean expression is equal to \(2^{(p+q)}\) (see below for definition), much less than the exponentially increasing combination of the subexpressions. This is because only a few combinations of the TRUE and FALSE values of the subexpressions are critical to mutation testing.

Suppose the boolean expression consists of several ANDed and ORed segments. In each segment there is only a single kind of boolean operator (AND or OR). The relational expression with six kinds of relational operators can be divided into two classes: with a relational operator '<=', '=>', or 'x' whose mutants can only be killed by executing the TRUE path, and with a relational operators '<', '>', or '=' whose mutants can only be killed by executing the FALSE path.

Then \(p = p' + p''/2\),

\(q = \max(q'/2 + q')\), (among all ANDed and ORed segments). Where \(p'\) is the number of the ANDed or ORed segments containing the relational operators '<=', '=>', and 'x', and \(p''\) represents the number of the ANDed or ORed segments which only contain relational operators '<', '>', or '='. In addition \(q\) represents the maximum number of the sum of the number of relational operators '<', '>', '=' (q') and the half of the number of '<=', '=>', or 'x' (q'/2) contained in an ANDed or ORed segment among all the ANDed or ORed segments.

This demonstrates that the testing strategy for killing all the mutants of the six basic components of a boolean expressions is feasible including complex boolean expressions. The preceding results are only a brief summary regarding this problem the details may be referred to in [8].

The domain testing strategy provides a formal approach for satisfying the often suggested guideline of revealing path boundary errors. Due to the error prone characteristic of the path border it is appealing to establish a test system using the domain strategy. Some interesting theoretical discussions of the selection of test data and useful propositions for domain testing test coverage metrics were presented in [13][14]. However, there are still some problems to be solved before applying it in an industrial validation process, e.g. how to select the \(2^{*1}\) or \(2^{*2}\) test data points on and near the ends of the closed path border, or how to put test data at the centroid of the hyperplane segment of the border and how to calculate the domain coverage metrics DEM (domain error magnitude) and BSE (border shift error). The SDSAM method and domain testing aim at a similar test target but in two distinct ways. The domain test strategy is attempting to form a theoretical model for the phenomenon of path border shift. The SDSAM is striving to find the practical technique to detect all component errors in a predicate. The mutation test strategy described here is limited, however, to revealing simple errors of a predicate. The domain testing strategy in general may possibly uncover some multiple errors in a path border. However in practice SDSAM possesses the following advantages:

1) better usability. All requirements for killing the mutants of various computational components are easier to implement.

2) better test coverage metrics. The test coverage metrics of SDSAM can indicate a comprehensive evaluation of all the individual mutated components. A global coverage metric can be given for whole programs. In contrast it is difficult for the domain test strategy to accumulate the total effects of the DEM or BSE of all predicates.
3) it is capable of detecting boolean operator errors, variable reference errors and variable assignment errors.

4) practicality. It is unlikely that domain testing can be successfully carried out in practice for most non-trivial programs whereas SDSAM can be implemented.

CONCLUSION

The SDSAM attempts to detect most simple errors in software. It sets up a sophisticated and flexible static and dynamic test system to accomplish this task. Comparison with other error-based test approaches shows that the SDSAM strategy possesses better applicability to software quality control and industrial validation.

REFERENCES


