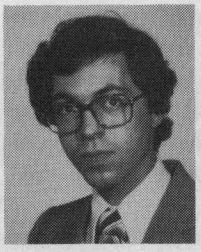
[110] G. F. Walters and J. A. McCall, "The development of metrics for software R & M," in *Proc. Annu. Rel. and Maintainability Symp.*, 1978, pp. 78–85.
[111] —, "Software quality metrics for life-cycle cost-reduction," *IEEE Trans. Rel.*, vol. R-28, pp. 212–220, Aug. 1979.
[112] E. J. Weyuker and T. J. Ostrand, "Theories of program testing and the application of revealing subdomains," in *Dig. Workshop on Software Testing and Test Documentation*, FL, Dec. 1978, pp. 1–18.
[113] B. B. White, "Program standards help software maintainability," in *Proc. Annu. Rel. and Maintainability Symp.*, 1978, pp. 94–98.
[114] M. V. Zelkowitz, "Perspectives on software engineering," *Comput. Surveys*, vol. 10, pp. 197–216, June 1978.

C. V. Ramamoorthy (M'57–SM'76–F'78) received the undergraduate degrees in physics and technology from the University of Madras, Madras, India, the M.S. degree and the professional degree of Mechanical Engineer, both from the University of California, Berkeley, and the M.A. and Ph.D. degrees in applied mathematics and computer theory from Harvard University, Cambridge, MA.

He was associated with Honeywell's Electronic Data Processing Division from 1956 to 1971, last as Senior Staff Scientist. He was a Professor in the Department of Electrical Engineering and Computer Sciences at the University of Texas, Austin. Currently, he is a Professor in the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley.

Dr. Ramamoorthy was Chairman of the Education Committee of the IEEE Computer Society and Chairman of the Committee to develop E.C.P.D. Accreditation Guidelines for Computer Science and Engineering Degree Programs. He also was the Chairman of the AFIPS Education Committee, a member of the Science and Technology Advisory Group of the U.S. Air Force, and a member of the Technology Advisory Panel of Ballistic Missile Defense (U.S. Army). Currently, he is Vice President of the IEEE Computer Society for Educational Activities.



Farokh B. Bastani (M'82) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Bombay, India, in 1977, and the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California, Berkeley, in 1978 and 1980, respectively.

He joined the University of Houston, Houston, TX, in 1980, where he is currently Assistant Professor of Computer Science. His research interests include developing a design methodology and quality assessment techniques for large-scale computer systems.

Weak Mutation Testing and Completeness of Test Sets

WILLIAM E. HOWDEN

Abstract—**Different approaches to the generation of test data are described. *Error-based* approaches depend on the definition of classes of commonly occurring program errors. They generate tests which are specifically designed to determine if particular classes of errors occur in a program. An error-based method called *weak mutation testing* is described. In this method, tests are constructed which are guaranteed to force program statements which contain certain classes of errors to act incorrectly during the execution of the program over those tests. The method is systematic, and a tool can be built to help the user apply the method. It is extensible in the sense that it can be extended to cover additional classes of errors. Its relationship to other software testing methods is discussed. Examples are included.**

Different approaches to testing involve different concepts of the adequacy or *completeness* of a set of tests. A formalism for characterizing the completeness of test sets that are generated by error-based methods such as weak mutation testing as well as the test sets generated by other testing methods is introduced. Error-based, functional, and structural testing emphasize different approaches to the test data generation problem. The formalism which is introduced in the paper can be used to describe their common basis and their differences.

Index Terms—**Complete, effective, mutations, testing.**

Manuscript received May 1, 1980; revised June 19, 1981. This paper is a revised version of the paper "Completeness Criteria for Testing Elementary Program Functions," presented at the 5th International Conference on Software Engineering, 1981.

The author is with the Department of Electrical Engineering and Computer Science, University of California at San Diego, La Jolla, CA 92037.

INTRODUCTION

In *functional testing*, the programmer identifies the functions which are supposed to be implemented by his program, and then tests the code against the specifications for those functions [1]–[3]. The functions to be tested may be described in either requirements or design specifications [4]. Design functions may correspond to parts of a program. Guidelines have been developed both for the identification of and for the construction of data for testing functions.

0098-5589/82/0700-0371$00.75

Structural testing can be more precisely defined than functional testing. It requires that tests be constructed that result in the execution of components of a program [5]. Branch structural testing, for example, requires that tests be constructed which result in the execution of every program branch at least once [6]-[8]. Other methods require the execution or use of other types of program components.

Error-based testing involves the construction of tests which are designed to uncover specific errors or classes of errors [9]. Error-based testing guidelines may consist only of informal rules such as "test for off-by-one indexing errors" or "test for extremal values of input variables." Recent work has attempted to make error-based testing more systematic. One idea is to design systematic procedures for identifying error-prone constructs and associated "revealing" test data. Goodenough and Gerhart proposed the use of decision tables for systematically identifying error-prone situations in programs [10]. Weyuker and Ostrand have proposed the use of path-based testing combined with the identification of "error-revealing subdomains" of a program's input domain [9].

Other error-based testing methods have attempted to define procedures for generating test data for revealing problem independent classes of errors. The results reported in [11] characterize the tests that are required for "algebraic errors" in certain classes of programs. In [12], Cohen and White characterize the tests that are required for discovering "domain errors."

Mutation testing is one of the most recent error-based testing methods [13]-[18]. It requires the definition of a set of mutation transformations which, when applied to elementary components of a program, introduce "errors" of certain types into the program. Typical transformations change variable names in expressions, alter labels in go-to's, and add one to loop bounds in loop constructs. The goal in mutation testing is to construct a set of tests T which will distinguish between a given program P and any nonequivalent program P' which can be generated from P by the application of mutation transformations to components of P.

In the approach to mutation testing described in [14]-[17], test data are constructed which will distinguish between a program P and a mutation P' which can be generated from P by the application of a single mutation transformation. A test t distinguishes between P and a mutation P' of P if P and P' give different outputs for t. The term "mutation testing" was introduced by the authors of [13]-[16]. Their method will be referred to as *strong mutation testing*.

The authors of strong mutation testing speculated that the test data which will distinguish between a program P and programs which can be generated from P by a single mutation transformation will also distinguish between P and programs generated by repeated applications of mutation transformations. They called this the "coupling effect." The coupling effect will not be assumed, and the term "strong mutation testing" in this paper will refer to the construction of tests which are designed to distinguish between a program and programs which can be derived by the application of a single mutation transformation.

This paper describes an error-based testing method called *weak mutation testing*. Suppose that P is a program, that C is a simple component of P, and that there is a mutation transformation that can be applied to C to produce C'. Let P' be the mutated version of P containing C'. In weak mutation testing, it is required that a test t be constructed which has the property that C is executed during the execution of P over t, and that on at least one such execution of C, C computes a different "value" from C'. The implementation of weak mutation testing depends on the specification of a set of components and a set of associated component mutation transformations. Components will normally correspond to elementary computational structures in a program. References to variables, arithmetic expressions and relations, and Boolean expressions are all examples of components. One component may appear as part of another more complex component.

It is important to note that even though a component C of a program P may compute a different value from a mutated version C' of C during an execution of P over a test t, is is still possible for P to compute the same program output for t as the mutation P' containing C'. This is the disadvantage of weak mutation testing. Its use, unlike strong mutation testing, does not guarantee the exposure of all errors in the class of errors associated with the mutation transformations.

B. Component Mutations

1) Variable Reference: The input to a variable reference component consists of the variable state space for the program in which the reference occurs. The output consists of a value of one of the variables. The *wrong variable* mutation transformation, when applied to a variable reference component, causes the component to reference a different variable. Less general instances of the wrong variable mutation include the *wrong array element* mutation and the *wrong input variable* mutation. The first of these two causes a component which references an array element to reference some other element of the array. The second causes a component which references a call by value formal parameter to refer instead to some other call by value formal parameter.

Suppose that v is the variable associated with a variable reference component C in a program. In order to cause C to compute a value which is different from a possible mutation C' of C, it is necessary to execute C over a variable state space in which v has a value which is different from the values of all other variables in the space.

It is very expensive to check for completely general wrong variable mutations. It is basically an n^2 operation to check, for every variable reference component, whether or not the variable involved has a value which is different from the values of all other variables. The process of carrying out such a check is equivalent to (strong) mutation testing at the statement level, and it is necessary to restrict wrong variable mutation testing to certain special cases such as wrong array element and wrong input variable testing. Error studies indicate [19] that wrong array element references occur frequently due to indexing errors, and it is important to force array element reference components to be executed over data which cause them to be distinguished from mutated components which reference different elements of the array. This can be accomplished by forcing the elements of arrays, if possible, to be

distinct. The studies also indicate that many wrong variable errors are associated with the referencing of the initial input values of variables. It is often sufficient to require that the values of the input variables to a module be distinct. Other restricted classes of wrong variable errors can also be formulated.

Fortunately, wrong variable mutation transformations are the only general class of mutations which are considered in this paper for which it is prohibitively inefficient to monitor whether or not the data over which a component has been executed distinguish the component from all of its possible mutations.

2) Variable Assignment: The input to a variable assignment consists of the variable state space for the program and a value. The output consists of a (possibly) new state space. The wrong variable mutation transformation, when applied to a variable assignment component, causes the component to assign the value to a different variable.

Suppose that C is a variable assignment component and that C' is a wrong variable mutation of C. Let v be the variable to which C assigns a value. In order to cause C to return output which is different from that which is computed by any wrong variable mutation C' of C, it is sufficient to execute C over data in which the value stored by C into v is different from the value currently stored in v. This will guarantee that the state spaces returned by C and C' will be different.

The advantages of weak mutation testing are twofold. The first is efficiency. A large number of mutants P' can be generated by the application of a relatively modest set of mutation transformations. It is estimated that there are on the order of n^2 mutants for an n-line program. This implies that if a proposed test set contains k elements, it will be necessary to carry out between n^2 and n^2k program executions to determine if the test set satisfies the requirements for strong mutation testing. Although there is the same number of mutations in weak mutation testing as in strong mutation testing, it is not necessary to carry out a separate program execution for each mutation. It is often possible to test all of the applications of a mutation transformation to all applicable program components with a single test.

The second advantage of weak mutation testing is the possibility of describing *a priori* for a mutation the types of data over which a *component* C must be executed in order for C to compute a different value from the mutated version C' of C. This does *not* mean that it is possible to describe *a priori* the *program* tests t that will result in the execution of particular components over the required data. It is still an important advantage, since it provides the user of the method with specific guidance as to the types of tests for which he must look.

In [18], Brooks introduced a method in which certain classes of program traces rather than output values are used for distinguishing between a program P and its mutants P'. The advantage of using traces is that it eliminates the need for unproved assumptions such as the coupling effect. Brooks' method results in the generation of test data which are guaranteed not only to distinguish between a program and the mutants which can be generated with a single application of a mutation transformation, but between the program and mutants generated by repeated applications of mutation transformations. The validity of the method has only been explored for programs written in a version of pure Lisp and for particular classes of mutation transformations. Both strong and weak mutation testing can be applied to general classes of programs in any language and are, in this sense, more general than trace mutation testing.

Weak Mutation Testing

A. Program Components

Five basic types of program components will be considered. The five components are variable reference, variable assignment, arithmetic expression, relational expression, and Boolean expression. The first two are the most basic and appear as parts of other types of components. Arithmetic expression components may appear as parts of relational expressions, and relational expressions may appear as parts of Boolean expressions. For each type of component, one or two mutations that correspond to commonly occurring errors in components of that type will be considered.

Variable reference and variable assignment are the most basic program components. Variable reference in a program involves the access and retrieval of a variable's value. Variable assignment involves the assignment of a value to a variable. Arithmetic expressions are built from variables and arithmetic operators. Relational expressions are built from arithmetic expressions and the operators $<$, $\leqslant$, $=$, $>$, $\geqslant$, and $\neq$. Boolean expressions are built from variables, relational expressions, and the logical operators AND, OR, and NOT.

3) Arithmetic Expression: The input to an arithmetic expression consists of a vector of variable values. The output consists of a computed value. Three classes of errors (mutation transformations) will be considered for arithmetic expressions: off by an additive constant, off by a multiplicative constant, and wrong coefficient. The first two cases are special cases of the third, but are considered separately because they require substantially fewer tests than the general case.

Suppose that E is an arithmetic expression and that E' is an additive constant mutation of E. It is sufficient to execute E over any vector of variable values in order to distinguish it from E'. If E' is a multiplicative constant mutation of E, then it is sufficient to execute E over any vector of values for which E gives a nonzero value in order to distinguish E from E'.

If E' is a mutation of E in which one or more of the coefficients of E have been altered, then more extensive data may be required to distinguish E from E'. Suppose that k is an upper bound on the exponents in E (and hence E'). Then in [20], it is proved that any cascade set of degree $k + 1$ will distinguish E from E'. The cascade set results in [19] can be extended to cover rational forms over unique factorization domains, rational forms for which the singularities are known, and even rational forms containing radicals. Unfortunately, the size of the reliable test sets for general classes of forms of this type becomes prohibitively large very quickly. Suppose, for example, that E is a rational form in three variables, that the maximal exponent is 5, and that the coefficients are integral.

$exp_1 < exp_2$	$exp_1 = exp_2$	$exp_1 > exp_2$	$exp_1 \neq exp_2$	$exp_1 \leqslant exp_2$	$exp_1 \geqslant exp_2$
T	F	F	T	T	F
F	T	F	F	T	T
F	F	T	T	F	T

Fig. 1. Outcomes for different possible relations.

Then the cascade set necessary to distinguish E from coefficient mutations E' of E will contain 11^3 elements.

DeMillo and Lipton prove in [21] that it is not necessary to consider large sets of tests for arithmetic expression mutations if probabilistic rather than deterministic results are acceptable. Suppose that E is an arithmetic expression of degree k and that E' is a coefficient mutation of E. Let X be any randomly chosen vector of values for the variables in E (and E'). If $E = E'$ at X, then X is a root of $E - E'$. The roots of multinomial of degree k from a subspace of "measure" zero of the space of possible variable values, and the probability of choosing a root of $E - E'$ at random is arbitrarily small. This indicates that random selection of a test point is "probably" sufficient to distinguish E from any coefficient mutation of E.

4) Arithmetic Relation: The input to an arithmetic relation expression consists of a vector of variable values. The output consists of the logical value T or F. Two simple kinds of mutations will be considered for arithmetic relations: wrong relational operator and off-by-an-additive-constant.

Let R be an arithmetic relation, and suppose that R' is a wrong relation mutation of R. Suppose R is of the form $exp_1\ r\ exp_2$ and R' is $exp_1\ r'\ exp_2$. If R is executed over data for which $exp_1 < exp_2$, $exp_1 = exp_2$, and $exp_1 > exp_2$, then in at least one case it will give a different output from R'. This can be seen by examining Fig. 1. For each possible kind of relation, the outcome of R and R' will differ in at least one case.

For some programs it may not be possible to construct tests that will cause some arithmetic relation R to be executed over data for which $exp_1 < exp_2$, $exp_1 = exp_2$, or $exp_1 > exp_2$. It is sufficient to cause R to be executed over data for as many of these three cases as possible. The argument for the case where it is possible to construct data for $exp_1 = exp_2$ and $exp_1 > exp_2$ but not $exp_1 < exp_2$ will be considered. The other cases can be argued similarly. If there are no input data to a program which results in the execution of an arithmetic relation component $exp_1\ r\ exp_2$ for which $exp_1 < exp_2$, then examination of Fig. 1 indicates that it is not possible to distinguish between $exp_1 > exp_2$ and $exp_1 \neq exp_2$ and between $exp_1 \leqslant exp_2$ and $exp_1 = exp_2$. It is assumed that it is not necessary to distinguish $exp_1 < exp_2$ from the other expressions since this case cannot arise. Now if $exp_1 < exp_2$ can never occur, then $exp_1 > exp_2$ is equivalent to $exp_1 \neq exp_2$. Similarly, if $exp_1 < exp_2$ can never occur, $exp_1 \leqslant exp_2$ is equivalent to $exp_1 = exp_2$. This implies that it is not necessary to distinguish between $exp_1 > exp_2$ and $exp_1 \neq exp_2$ and between $exp_1 \leqslant exp_2$ and $exp_1 = exp_2$ because these are equivalent expressions (i.e., equivalent given that $exp_1 < exp_2$ can never occur).

To simplify the discussion of off-by-an-additive-constant mutations, it is assumed that arithmetic relations are of the form $exp\ r\ 0$. Suppose that R is a relation, and that R' is a mutation of R of the form $(exp + k)\ r\ 0$ where k is any nonzero constant. In order to described data for revealing constant mutations, it is necessary to consider quantities such as "the largest number less than zero which can be taken on by *exp*." Thus quantity is meaningful due to the finite representation of numbers. If *exp* is an integer expression, the quantity is at most -1. If *exp* is real, it is at most $-\epsilon$ where ϵ is the smallest nonzero quantity which can be represented on the machine in use. In situations where it is impractical to require the generation of nonzero quantities ϵ which are as close to zero as possible, it is still possible to construct data for distinguishing additive mutations in relations if there is a quantity ϵ which measures the smallest required degree of discrimination between numbers. Two numbers which differ by ϵ are considered to be effectively equal.

a = maximum value of *exp* which is less than zero
b = maximum value of *exp* which is less than or equal zero
c = minimum value of *exp* which is greater than zero
d = minimum value of *exp* which is greater than or equal zero
e = zero

Fig. 2. Constants used for distinguishing data.

r	*exp*
$<$	a, d
$>$	c, b
$\leqslant$	c, b
$\geqslant$	a, d
$=$	e
$\neq$	e

Fig. 3. Distinguishing data.

The definitions in Fig. 2 describe data that can be used to distinguish between an arithmetic relation component $exp\ r\ 0$ and a mutation of that component which is off by an additive constant. Fig. 3 defines the required data on the basis of relation r in the component. It indicates, for example, that if r is the relation $<$, then tests should be selected which cause *exp* to evaluate to a and also to evaluate to d.

The ability of the data in Fig. 3 to distinguish between a component $(exp\ r\ 0)$ and a mutation $((exp + k)\ r\ 0)$ can be argued as follows. Suppose r is the relation $<$. If $(exp + k) < 0$ when $exp = a$, $exp + k$ is equivalent to *exp* for all values for which *exp* evaluates to a negative number. Otherwise, the datum for which $exp = a$ discriminates between $(exp\ r\ 0)$ and $((exp + k)\ r\ 0)$. If $(exp + k) \geqslant 0$ when $exp = d$, then $exp + k$ is equivalent to *exp* for all values for which *exp* evaluates to a num-

ber greater than or equal to zero. If $(exp + k) < 0$ when $exp = d$, then the test for which $exp = d$ discriminates between exp and $exp + k$. Hence, either exp and $exp + k$ are equivalent or tests for which $exp = a$ and $exp = d$ distinguish between exp and $exp + k$.

The arguments for the reliability of the other relations in Fig. 3 are similar to that for the relation $<$. If no values for the variables in exp can be selected so that $exp = 0$, then it will not be possible to construct distinguishing test data for expressions having relations = and $\neq$. The impossibility of selecting these last values in itself points out the error.

The data that are required to distinguish off-by-a-constant and wrong-relation mutations in a relational expression $exp_1\ r\ exp_2$ can be combined by requiring the execution of the relation over data for which $exp_1 - exp_2 = -\epsilon, 0$, and $+\epsilon$.

The domain testing strategy described by White and Cohen in [12] is closely related to weak mutation testing of arithmetic relations, and ideas from domain testing can be applied to the problem of distinguishing between a relational expression R and a mutated expression R' which represents a "domain shift" in R. The domain procedure requires the use of three test points for relations containing other than equality operators and four test points for equality operators. Although the domain testing strategy requires more test points than the procedure described above, it has the advantage that it can be applied to a wider class of errors. Its applicability is less formally defined than the above procedure, and the errors for which it is useful are defined geometrically in terms of their effect on the surface $exp = 0$ involved in the relation $exp\ r\ 0$. The arguments for its validity are also geometric, and although easy to follow in two or three dimensions, they are difficult for higher dimensions. They could be made more rigorous, and the ideas in domain testing be used to extend weak mutation testing of arithmetic relations to additional classes of errors.

The wrong relation and off-by-a-constant results described above are derived from work described by Foster in [22]. In his paper, Foster assumes that all variables are integers, and that it is possible to construct tests which result in all possible relationships between the expressions exp_1 and exp_2 in an arithmetic relation $exp_1\ r\ exp_2$. His paper contains an informal discussion of the test cases needed to detect wrong-relation errors and integer off-by-a-constant errors. The results are easily extensible to real-valued relations and to generalized off-by-a-constant errors. Foster also discusses a number of other ad hoc error-based testing techniques for which there is no obvious model for discussing their effectiveness.

5) Boolean Expression: Boolean expressions are functions of the form $B = L(E_1, E_2, \cdots, E_n)$ where E_i, $1 \leqslant i \leqslant n$ is an expression or variable that evaluates to T or F. L is assumed to be a syntactically correct logical expression in the logical operators OR, AND, and NOT. The Boolean expression can be though of as a function in n logical variables. A test set can be constructed which will distinguish B from all other Boolean expressions B' in E_i, $1 \leqslant i \leqslant n$ by requiring that values for the variables in the E_i be selected in such a way that all *possible* combinations of T and F values for the E_i are generated. Note that this test set is still effective if, due to dependence between the E_i, it is not possible to generate some combinations of T and F values.

An exhaustive test set for a Boolean expression B grows exponentially in the number of subexpressions E_i, $1 \leqslant i \leqslant n$ in the Boolean expression. The number of tests required to distinguish an incorrect mutated Boolean B' from a Boolean B can be decreased if the types of errors which are to be considered are restricted. Substantial work on the testing of logic circuits has been carried out for stuck-at and bridging or short-circuit faults. The analogies of these hardware faults can be defined for software (e.g., a variable fixed at a constant value for stuck-at faults) and the types of tests used to distinguish expressions containing the errors defined. Akers surveys testing procedures for logic circuits in [23].

C. Implementation

The effective use of weak mutation testing depends on the availability of a tool which can be used to monitor statement executions during program testing. The tool should record, for selected classes of program components and associated component mutations, if the components have been executed over data which distinguish them from mutated versions of the components. The programmer is expected to examine reports from the tool, and upon discovering that a particular component C has never been executed over data which distinguish C from some mutated version C' of C, attempt to construct a program test that will cause C to be executed over data which will distinguish C from C'. Such a tool would be similar to, but more powerful than, the tools used to measure branch coverage. A weak mutation tool is now under construction.

D. Examples

The following two examples describe program errors which are detectable by weak mutation testing.

Example 1–Error in an Input Variable Reference Component: The program in this example is part of a computerized dating system. The (Cobol) system maintains a database of information about potential dates. Each date record contains the date's identification number and information about the date's personal characteristics and interests. One of the programs in the system is the compatibility date finder. The date finder takes as input the identification number of a dater. The dater is assumed to be one of the dates from the database. The date finder retrieves and then matches the dater's record in the database against all other dates in order to find the most compatible date. Compatibility is evaluated by comparing personal characteristics and interests of the dater to those of the candidate date. The record containing the most compatible date is stored in a data item called DATE.

The program contains a report construction module that takes as input the data record for the most compatible date and the data record for the dater. The program uses these two records to print out information about the selected date it has found for the dater. Each date record contains the date's preferred kind of first date (e.g., dinner, movie, etc.). This information is contained in a field called FIRST DATE. The program prints out, along with the name, etc., of the date, his/her preferred kind of first date. The program incorrectly

references FIRST DATE OF DATER when it is supposed to reference FIRST DATE OF DATE. This will result in suggesting the wrong kind of first date, except if both the date and the dater prefer the same kind of first date.

The error occurs in the variable reference component in the statement that is supposed to retrieve the FIRST DATE information that is later included as part of the output. The error is detectable if the value of FIRST DATE OF DATER (the referenced variable) is different from that of all other input variables to the module, including that of FIRST DATE OF DATE. The error is not detectable by branch or even complete path testing.

Example 2–Error in an Arithmetic Relation Component: The program in this example is used for printing out tables of monthly loan payments. Its input variables are PAYMENT, PERCENT, and PRINCIPAL. For each payment period, it calculates the amount of the PAYMENT that goes towards reducing the BALANCE of the PRINCIPAL and the amount paid out in interest.

The program is based on a WHILE loop that repeats the loan payment computation while BALANCE is larger than zero. The loan payment computation checks to see if BALANCE is less than or equal to PAYMENT. If it is, a special last payment message is computed, and BALANCE is reduced to zero. Otherwise, BALANCE is reduced by PAYMENT.

The error occurs in the relation used to determine if the looping process should continue. The relation checks to see if BALANCE is greater than zero. It should check to see if BALANCE is greater than 0.004. This is because the payments which are printed out are only printed to the nearest penny. Hence, in any example where BALANCE is reduced to less than or equal to 0.004, but is still larger than zero, the program will print out a last payment message along with a request for zero dollars and zero cents.

If the finest discrimination to which values are kept is 0.005, then weak mutation testing will require the testing of the relation over a value for which $0 <$ BALANCE $\leqslant 0.004$. This will result in the zero loan payment amount request, and the presence of the error will be revealed.

The following example describes an error for which weak mutation testing is not powerful enough to guarantee the discovery of the error. The error will be discovered by strong mutation testing.

Example 3–Weak Mutation Testing Fails, Strong Mutation Testing Succeeds (Error in Arithmetic Relation Component): The program is used to compute safe resource allocations using the Banker's algorithm. The inner loop of the algorithm checks to see if a process' remaining unallocated resource requirements (its CLAIM) is less than the remaining unused resources. If it is, it assumes that the process could complete, and it simulates the return of its allocated resources to the unused resource pool in order to determine if othe processes could also complete, given that this process could complete, and hence free all of its resources. The output from the algorithm consists of either the message "potential deadlock" or "system in safe state."

The error occurs in the comparison of CLAIM with UNUSED RESOURCES. It should check to see if CLAIM is less than or equal to UNUSED RESOURCES, not less than. Weak mutation testing requires that a test be constructed for which CLAIM = UNUSED RESOURCES. When this happens, the algorithm will incorrectly fail to note that the process being checked (say process A) can complete, and it will not simulate the return of its resources to UNUSED RESOURCES. After checking A, the routine will consider all other processes. It will then go back and do all the processes in order again since a process whose claim was previously too large may now have its CLAIM less than UNUSED RESOURCES because of the simulated return of resources returned by processes considered after A. This may result in the determination that A can complete because the CLAIM for A is now smaller than UNUSED RESOURCES. In this case, the algorithm will terminate with the correct output, even though it made incorrect decisions during its computations. Weak mutation testing is not reliable for this error.

Strong mutation testing is effective for the error in the sense that the necessity of distinguishing between the given program and the mutation in which "less" is replaced by "less than or equal to" will force the discovery of the presence of the error.

Effectiveness and Completeness

A. Effectiveness of Test Sets

The concept of effectiveness in testing has been defined in several ways. Goodenough and Gerhart defined it in terms of two related concepts which they called reliability and validity. In [24], a test is defined to be reliable for an error if its use is guaranteed to result in the discovery the presence of that error. Weyuker and Ostrand developed the concept of effectiveness used by Goodenough and Gerhart, and invented the related concept of a revealing subdomain [9].

The following definition is closely related to the definition of reliability used in [11] and to the basic ideas in mutation testing.

Definition: Suppose that S_f is a set of functions which have the same domain and that S_f contains the function f. A test set T is *effective* for f relative to S_f if T is nonempty and if for all functions f' in S_f, $f' = f$ over T implies that $f' = f$ over the entire domain of the functions in S_f.

The set S_f in the definition can be thought of an modeling the errors for which a test set is effective. Suppose that a program P computes a function f and that P has an error. Suppose that P is "supposed to compute" some other function f'. The difference between f and f' represents an error in P. The differences between f and the functions $f' \neq f$ in S_f represents a set of possible errors in P. An effective test set for S_f will reveal any of these errors.

The set S_f can also be used to define special classes of programs for which it is possible to build effective test sets. Reference [11] describes how to build effective tests sets for a class of array manipulation programs.

B. Completeness of Test Sets

Systematic testing methods involve different concepts of the completeness of a set of tests. Their use is justified by intuition, by experience with programming errors, and by the extent to which they can be systematically applied.

Suppose that P is a program which computes a function f, and let U_f be the set of all functions which have the same domain as f. It is theoretically impossible to construct a general testing procedure for all P which has the property that the complete test sets generated by the procedure are effective for f relative to U_f. There are two ways of dealing with this problem that have been used in testing. The first involves the consideration of subsets of U_f. Tests are selected for P which correspond to the evaluation of f over data which are effective for f relative to a subset S_f of U_f. The second way is to consider "subfunctions" of f. f is associated with the computation carried out by P as a whole. Subfunctions are associated with parts of the total computation carried out by P. Suppose F is a set of subfunctions. Tests are selected for P which result in the evaluation of each function in F over data which are effective for that function. In the first way of dealing with the problem, subsets S_f are selected which correspond to classes of possible errors in P. In the second, subfunctions are selected which correspond to parts of programs in which an error can occur.

The following definition defines the completeness of a test set T in terms of the functions which are effectively tested by T.

Definition: Suppose that P is a program and that F is a set of functions associated with P. Assume that there is a mapping M such that for each subset T of the domain of P and each function f in F, M defines a subset of the domain of f. Assume that for each function f in F, there is an associated set of functions S_f. Let T be a set of tests for P. Then T is a *complete* set of tests for P, relative to F and to $\{S_f : f \in F\}$, if $M(T,f)$ is effective for all $f \in F$ relative to S_f.

The definition can be used to characterize the notion of completeness used in weak mutation testing and to compare it to other testing methods. In weak mutation testing, F is a set of functions which are computed by components of P. M maps subsets T of the domain of P onto the data sets over which the functions are evaluated when the components corresponding to the function are executed during the execution of P over T. If some component is not executed when P is executed over T, then M maps T onto the empty subset. For each f in F, S_f is the set of functions which are computed by mutations of the component which corresponds to f.

A complete test set for a testing method is usually effective for a number of different sets of functions F relative to different associated sets of functions $\{S_f : f \in F\}$. A complete set of weak mutation tests, for example, is effective for the function f computed by a program P as a whole relative to the following set S_f. Let P' be a mutation of P that is produced by mutating a component C of P and let C' be the mutated component. Then the function computed by P' is in S_f if and only if when P is executed over data which distinguish C from C', P' gives different output from P. In the following discussions of different testing methods, only those sets F and associated sets S_f will be discussed which correspond to the basic concept or motivating idea used in the definition of a testing method.

In strong mutation testing [14], the set F consists of the single function f which is computed by P. M maps (T,f) onto T, and F_f consists of the functions which are computed by mutations of P. Complete test sets in strong mutation testing are intended to be effective for the function computed by the program as a whole. Complete test sets in weak mutation testing are only intended to be effective for the component functions.

Two sets F will be described for branch testing [6]. One way of looking at branch testing is to define F to be the single function f which is computed by a given program P. M maps (T, f) onto T, and S_f contains the functions computed by programs P' which have the property that for some branch in P, when P is executed over data that cause that branch to be followed, P and P' give different output.

Another way of characterizing branch testing is to consider the functions which are associated with the Boolean expressions in conditional branching instructions. Let F be those functions, and for each f in F, let S_f contain f, f^C (the logical complement of f) and the functions f_{TRUE} and f_{FALSE} which always evaluate to *TRUE* and *FALSE*. For subsets T of the domain of P and each branch function $f, M(T,f)$ are the data over which the expression corresponding to f is evaluated when P is evaluated over T. If T is a complete set of branch, tests, then $M(T,f)$ will be effective for each f in F relative to S_f.

The second way of describing branch testing allows it to be more easily compared to weak mutation testing and illustrates why weak mutation testing can be thought of as a refinement of branch testing. It has long been recognized that the detection of many errors requires not only that a branch be executed during some test, but that it be executed over data that are related to specific kinds of errors that can occur in statements and branches. The sets S_f in branch testing correspond to a limited number of very coarse errors, errors that are revealed by almost any data which cause the statement containing the error to be executed. The sets S_f in weak mutation testing are more refined. They correspond to a wider class of errors, including subtle errors which require the evaluation of branch functions over very specific kinds of values. In addition, the sets F are larger in weak mutation testing and are not limited to functions corresponding to Boolean expressions.

The set F in domain testing [11] consists of functions which are computed by path conditions. For each path through a program P, there is a path condition function f. For each element x of the input domain of P, $f(x)$ is *TRUE* or *FALSE*, depending on whether or not x causes the path associated with f to be followed when P is executed over x. M maps (T, f) onto T, and S_f contains all functions f which can be derived from the path conditions associated with f by a "domain shift" error. White and Cohen have defined test sets T which are effective for path condition functions f relative to sets of "domain shift" functions S_f. They have also defined the conditions under which the test sets T are effective for the function f computed by the program as a whole.

A major stumbling block in the practical use of domain testing is the size of the function set F. It may contain an infinite number of functions. In [24], Zeil and White describe a technique for limiting the size of this set.

Weak mutation, branch, and domain testing are effective for functions which correspond to parts of a program. They are effective relative to sets S_f which correspond to errors in those parts of the program. Weak mutation testing is more general since the sets F can correspond to any type of program component, whereas domain testing is restricted to path condition components. It is the restriction in domain testing that makes it possible to isolate a meaningful set of conditions under which domain testing is effective for the function f computed by a program as a whole.

The set F in trace mutation testing [18] consists of a single function f whose range contains special kinds of value traces that are generated when P is executed. S_f contains f and all trace generation functions that are computed by programs which can be derived from P by the application of one or more shape-preserving mutation transformations. M maps (T, f) onto T. Let S_f' be the subset of S_f containing functions which correspond to programs that can be derived from P by the application of a single mutation. Brooks proves in [18] that if T is complete relative to F and $\{S_f' : f \in F\}$, then it is also complete relative to F and $\{S_f : f \in F\}$.

The set F in functional testing [1], [4] consists of functions which are described in the specifications for P. For functions which correspond to parts of P, M maps T onto the data over which those parts of the program are executed when P is executed over T. For each f, S_f contains functions which can be distinguished from f when f is evaluated over a set of functionally important test cases. The definition of a functionally important test case is imprecise, and this is one of the problems in functional testing. Duncan describes a data grammar method in [26] which may make it possible to systematically define the characteristics of a complete set of functional test cases.

The completeness definition can be used to discuss other testing methods including Woodward, Hennell, and Hedley's LCSAJ testing [27], Miller's levels of test coverage completeness [7], and Pimont and Rault's branch-pair testing [28].

There are three parts to the definition of test set completeness: the set of functions F which is tested, the sets of functions S_f which model the errors for which a complete test set is effective, and the complete test set T. In each of the three approaches to testing (functional, structural, and error-based), one of these three parts is defined in terms of the other two. In functional testing, the emphasis is on the direct *a priori* identification of the functions F to be tested and the test sets T which are to be complete. The functions S_f for which T is effective are defined indirectly in terms of T. In structural testing, the functions F and the complete test sets T are again defined first, and then the function sets S_f are defined afterwards. Both kinds of methods begin with a presumed notion of what a complete test set is, rather than with a definition of the errors for which they will test. Error-based testing begins with F and the function sets S_f. Complete test sets T are then defined in terms of the S_f. Weak mutation testing is different from structural and functional testing methods in this respect. It is different from other error-based methods such as strong and trace mutation testing in the set of functions F for which it is intended to be effective. Both strong and trace mutation testing are intended to be effective for the functions computed by the program as a whole, whereas weak mutation testing is intended to be effective for functions computed by elementary computational components in programs.

Summary

Two ideas have been developed in the paper. The first is that of *weak mutation testing*. Weak mutation testing requires the identification of classes of elementary program components and of simple errors that can occur in the components. Arithmetic relations, arithmetic expressions, and Boolean expressions are all examples of simple program components. References to and assignments of values to variables are examples of simple components that occur as part of other components. Component errors include wrong operator, off-by-a-constant expression, and wrong variable. Weak mutation testing requires that when programs are tested, test data be selected that result in the execution of components over data which distinguish the components from components which contain simple component errors. Data distinguish between two components C and C' if C and C' return different results when executed over the data. The datum $x = -2$ for example, distinguishes between $x + 2 < 0$ and $x + 2 \leqslant 0$, but not the datum $x = 2$.

Weak mutation testing has features in common with algebraic testing, domain testing, and error-based testing, as well as strong and trace mutation testing. The advantage of weak over strong mutation testing is that weak mutation testing does not require a separate program execution for each mutation of a program, and that it is possible to specify *a priori* the data over which components must be executed in order to carry out a complete set of weak mutation tests. Its advantage over trace mutation testing is that it does not require a knowledge of the correct internal states of a program during its execution over a test. In addition, weak mutation testing is applicable to a wider range of programs. The trace mutation testing system described by Brooks in [18] is restricted to detecting "shape-preserving" errors in a limited class of Lisp programs. The application of trace-testing ideas to a wider class of programs is described in [29].

The disadvantage of weak mutation testing is that a program containing a mutation error may act correctly over a set of complete weak mutation test data. Complete weak mutation test sets are not effective for the function computed by a program. They are effective only for component functions. Both complete strong and trace mutation test sets are guaranteed to reveal specified classes of mutation errrors, and are effective for the function computed by a program as a whole.

Weak mutation testing can be thought of as a refinement of branch testing. Branch testing requires that each branch in a program be executed at least once during some test. Weak mutation testing forces the testing of branches, as well as other simple program components, over error-related data.

Weak mutation testing was derived from work by Foster [22] and is related to ideas in circuit testing [23]. In circuit testing, tests are built for revealing the presence of very specific types of errors occurring in single circuit components (e.g., gates). The most widely studied component errors are "stuck-at" line faults.

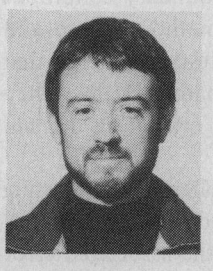
The second idea described in the paper is a formalism for discussing complete and effective test sets. A test set is defined to be effective for a function implemented by a piece of code P if it will distinguish that function from functions implemented by other nonequivalent pieces of code. A test distinguishes between two functions f and f' if they evaluate to different values over that test. Test sets are normally only effective for certain classes of errors, so that test sets are said to be effective relative to a set of functions which correspond to those errors. A function corresponds to an error if it is computed by a program which differs from the correct program by that error. The formalism defines the completeness of a test set for a testing method in terms of the classes of functions for which the method generates effective test sets. Different testing strategies emphasize different aspects of the formal definition of test set completeness, and result in the construction of test sets that are effective for different sets of functions. The formalism is sufficiently general to allow the discussion and comparison of specific testing methods, as well as different classes of methods such as functional, structural, and error-based testing.

Acknowledgment

The author would like to thank K. Foster, J. Goodenough, and the referees of the earlier version of the paper for their many useful criticisms and suggestions. The term "weak mutation testing" was suggested by one of the referees.

References

[1] W. E. Howden, "Functional program testing," *IEEE Trans. Software Eng.*, vol. SE-6, 1980.
[2] M. A. Hennell, "Approaches to testing," Dep. Comput. Statist. Sci., Univ. Liverpool, Liverpool, England, 1980.
[3] J. R. Brown and E. C. Wilson, "Functional programming," TRW Defense and Space Syst. Group for Rome Air Develop. Cen., Contract Tech. Rep. F 30 60 2-76-C-0315, 1977.
[4] W. E. Howden, "Functional testing and design abstractions," *J. Syst. Software*, vol. 1, 1980.
[5] C. V. Ramamoorthy and S. F. Ho, "Testing large software with automated software evaluation systems," *IEEE Trans. Software Eng.*, vol. SE-1, 1975.
[6] L. Stucki, "New directions in automated tools in improving software quality," in *Current Trends in Programming Methodology*, vol. 2, R. T. Yeh, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1977.
[7] E. Miller, M. P. Paige, J. P. Benson, and W. R. Wisehart, "Structural techniques of program validation," in *Proc. IEEE COMPCON*, 1974.
[8] J. C. Huang, "An approach to program testing," *ACM Comput. Surveys*, 1975.
[9] E. J. Weyuker and T. J. Ostrand, "Theories of program testing and the application of revealing subdomains," *IEEE Trans. Software Eng.*, vol. SE-6, 1980.
[10] J. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Software Eng.*, vol. SE-3, 1977.
[11] W. E. Howden, "Algebraic program testing," *Acta Inform.*, vol. 10, 1978.
[12] L. J. White and E. A. Cohen, "A domain strategy for program testing," *IEEE Trans. Software Eng.*, vol. SE-6, 1980.
[13] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Trans. Software Eng.*, vol. SE-3, 1977.
[14] R. J. Lipton and F. G. Sayward, "The status of research on program mutation testing," in *Dig. Papers, Workshop on Software Testing and Test Documentation*, Ft. Lauderdale, FL, Dec. 1978.
[15] T. A. Budd, R. J. Lipton, R. A. DeMillo, and F. G. Sayward, "Mutation analysis," Dep. Comput. Sci., Yale Univ., New Haven, CT, Res. Rep. 155, Apr. 1979.
[16] 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," in *Proc. ACM Symp. Principles of Programming Languages*, Las Vegas, NV, 1980.
[17] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, Apr. 1978.
[18] M. Brooks, "Determining correctness by testing," Stanford Univ., Stanford, CA, Comput. Sci. Tech. Rep. STAN-CS-80-804, 1980.
[19] W. E. Howden, "Applicability of software validation techniques to scientific programs," *ACM Trans. Programming Languages and Syst.*, vol. 2, 1980.
[20] —, "Algebraic equivalence of elementary computational structures," Univ. California, San Diego, 1976 (revised 1980).
[21] R. A. DeMillo and R. J. Lipton, "A probabilistic remark on algebraic program testing," *Inform. Processing Lett.*, vol. 7, June 1978.
[22] K. A. Foster, "Error sensitive test cases analysis (ESTCA)," *IEEE Trans. Software Eng.*, vol. SE-6, 1980.
[23] S. B. Akers, "Test generation techniques," *Computer*, vol. 13, 1980.
[24] W. E. Howden, "Reliability of the path analysis testing strategy," *IEEE Trans. Software Eng.*, vol. SE-2, 1976.
[25] S. J. Zeil and L. J. White, "Sufficient test sets for path analysis testing strategies," in *Proc. 5th IEEE Int. Conf. Software Eng.*, 1981.
[26] A. G. Duncan and J. S. Hutchison, "Using attributed grammars to test designs and implementations," in *Proc. 5th IEEE Int. Conf. Software Eng.*, 1981.
[27] M. R. Woodward, M. A. Hennell, and D. Hedley, "Experience with path analysis and testing of programs," *IEEE Trans. Software Eng.*, vol. SE-6, 1980.
[28] S. Pimont and J. C. Rault, "A software reliability assessment based on a structural and behavioral analysis of programs," in *Proc. 2nd IEEE Int. Conf. Software Eng.*, Long Beach, CA, 1976.
[29] W. E. Howden and P. Eichhorst, "Proving program properties from traces," in *Software Testing and Verification Techniques* 1st ed., E. Miller and W. E. Howden, Ed. Long Beach, CA: IEEE, 1978.



William E. Howden received the Ph.D. degree in computer science from the University of California, Irvine, in 1973.

He was employed at the University of Victoria, Victoria, B.C., Canada, and is presently an Associate Professor of Computer Science at the University of California, San Diego. He has carried out extensive research projects in both the practice and theory of program testing. He is coauthor of the IEEE tutorial book, *Software Testing and Validation Techniques*, and he has conducted professional seminars on software verification both in the U.S. and abroad.

Dr. Howden is a Distinguished Visitor of the IEEE Computer Society and a member of the Executive Board of the Society's Technical Committee on Software Engineering.