Complexity of the EQUATE Testing Strategy

Steven J. Zeil

COINS Technical Report 86-22
May 1986

Software Development Laboratory
Dept. of Computer and Information Science
University of Massachusetts
Amherst, Ma. 01003

This work was supported by NSF grants DCR-8404217 and DCR-8318776, CDC grant 84M103, and RADC grant F30602-86-C-0006.

Abstract

Most testing strategies place strict limits on the data types that may appear in the software being tested. With the evolution of new design techniques and new languages that encourage the use of more abstract data types, such limits are increasingly unrealistic. The EQUATE testing strategy offers strong support for data and functional abstraction. EQUATE selects a number of test locations throughout the program and chooses a set of expressions derived from the abstract syntax tree of the module being tested. Test data is required that distinguishes these expressions from one another at every test location. The time complexity of EQUATE is at worst $O(L_p^2)$ and the space complexity $O(L_p^3)$ where L_p is the length of the program under test.

I. Introduction

Much of the past research into software testing has depended upon severe restrictions to the types of data and operations occurring in the modules being tested [4,5,7,9,15,16,17]. Perhaps the most common restriction for testing strategies has been to restrict their operation to numeric data and to the standard arithmetic operators. In some cases, even data structures as basic as arrays or vectors of numbers have been prohibited. Meanwhile, the trend in other areas of software engineering has been the emphasis upon widespread use of data abstraction. Thus it is becoming increasingly important that testing strategies begin to deal with a wider variety of data types, so that the process of testing can be brought up to date with the kinds of software that will actually be written.

The EQUATE testing strategy was designed with exactly this goal in mind. Its approach to testing software that uses user-defined and/or abstract data types is to form testing criteria based upon the values of the operations defined upon those types. The particular instances of operations and parameters to those operations that are used by EQUATE are chosen to reflect "reasonable" operations for the module being tested. Evidence of reasonability is obtained by examination of the ways in which the operations and parameters are used throughout the module. The actual testing strategy is described in Section II.

Section III is concerned with the computational complexity of EQUATE and with its efficient implementation. Section IV provides an example of its use and an illustration of some of the ideas advanced in the earlier Sections.

II. The EQUATE Testing Strategy

A given set of test data may do a thorough job of exercising some portions of a module while leaving other portions almost completely untested. One possible response to this observation is to require test data that causes execution to pass through all portions of the module, or through selected combinations of portions of the module [8,11,12,13]. One shortcoming of such *path selection* approaches is the lack of attention to the actual data used to force execution of the selected paths. It is entirely possible for a statement to be executed repeatedly without our gaining any real confidence in that statement's correctness. Such confidence seems tied to our perception of whether the statement has been tested over

a sufficient range of different program states, a question that is only partially related to the test paths selected through that statement.

Of course, what constitutes a "sufficient range of program states" is far from obvious and is probably not subject to any single answer. In this paper an approximation is proposed: *Each valued object (variables, constants, and expressions) in the module should take on values that, in at least one instance, can be distinguished from those of any other object and from those of any constant.* When this condition has been satisfied at module locations immediately before and after a given statement, that statement is considered to have been properly exercised by the test data. If this condition is not yet satisfied at some location, then a wide variety of possible faults may be present at that location without their having affected the test results. Examples include substitution of one object for another, missing assignment statements in which one object should be assigned the value of another, and a potentially infinite number of missing or over-simplified expressions that should have involved the two objects.

Given this goal for test data selection, we are still left with a key question: what does it mean to "distinguish" two objects from each other? Clearly a necessary condition is that the strings of bits representing the values of the two objects must differ. This is not, however, by itself a sufficient condition. A principal tenet of data abstraction is that an object's value is revealed through the operations provided for use with that object. A reasonable conclusion would seem to be that two objects can be distinguished only if the values of the operations applied to those objects are different. Thus, for example, a portion of a memory management system concerned with manipulating blocks of uninitialized storage might be less concerned with whether the contents of two blocks differed than with whether the sizes of those blocks differed. In testing a statement of that system it is entirely reasonable to claim that *THIS_BLOCK* and *THAT_BLOCK* have not been distinguished from each other until *SIZE(THIS_BLOCK)* is distinguished from *SIZE(THAT_BLOCK)*. Note that "distinguish" begins to take on a recursive nature. Each expression computed by a module, by virtue of its returning a value when evaluated, represents a valued object in its own right, which should then be distinguished from the other objects in the module.

Now if X and Y are variables and f is one of the operators[1] on X, X is distinguished from Y only if $f(X)$ is distinguished from $f(Y)$. If, for example, X is a floating point number and a module makes use of $ABS(X)$, it makes sense to say that X and Y have been distinguished only if they have taken on different absolute values. If g is an operator on the type of data returned by f, then $f(X)$ is distinguished from $f(Y)$ only if $g(f(X))$ is distinguished from $g(f(Y))$.

If this chain of reasoning were continued for all possible operations then it could continue in this fashion indefinitely. As a practical matter, we must choose only those operations that are not only legal but also reasonable for each object. If, for example, X is a floating point number, the operation $SIN(X)$ may be legal, but there is little point in checking the value of $SIN(X)$ if the module is not performing trigonometric calculations. On the other hand, if the module already contains a reference to $SIN(X)$, this constitutes prima facie evidence that SIN is a reasonable operation on X. Thus EQUATE operates by the rule that two objects X and Y are distinguished only if, for each operator f such that either $f(X)$ or $f(Y)$ appears in the module, $f(X)$ is distinguished from $f(Y)$. If there is no such operator f, then X and Y are distinguished if $X \neq Y$.

The EQUATE strategy provides a local measure of the effectiveness of a set of test data, a measure which is applied at a variety of *test locations* throughout the module in order to gauge the overall effectiveness of a set of test data. This measure consists of a set of designated expressions and constants, referred to as *terms*, each of which must be distinguished from the others during testing.

The test locations occur at the beginning of each basic block and immediately following each statement in the block except when that statement is a conditional or unconditional branch. As each test location is reached during the execution of a test, the terms are evaluated and the resulting values are examined for equivalences. For any terms that have so far always been equivalent at that location, subsequent test data should be chosen to give them non-equal values.

---

[1] We will refer to the function f as an *operator*, regardless of whether it is written in infix form or as a function applied to parenthesized operands. For the sake of simplicity, the examples of operations given in this paper will be restricted to functions. Procedures and other modules not implemented as functions are treated as functions returning the cross-product of their output parameter types, followed by an assignment of the appropriate components to the actual output parameter variables [18].

The key to EQUATE's power lies in the selection of the set of terms to be distinguished from one another. There are three major components to EQUATE's set of terms:

1. The first component is the set of all expressions and subexpressions from the abstract syntax tree of the module being tested. This set is called the *expression set* of the module. The procedure in figure 1, for example, has the expression set shown in figure 2. Testing is required to continue until, at each test location, every pair of expression set members have taken on distinct values at least once.

```
package Variable_Length_Strings is
    type VString is private;
    function Len (S: VString) return Natural;
    function Left (S: VString; Width: integer) return VString;
    function Right (S: VString; Width: integer) return VString;
    function Mid (S: VString; Start, Width: integer) return VString;
    function '&' (S, T: VString) return VString;
    function IsEmpty (S: VString) return Boolean;
private
    .
    .
    .
end Variable_Length_Strings;
    .
    .
    .
with Variable_Length_Strings; use Variable_Length_Strings;
prccedure Substitute (Target: in out VString;
                      Search, Replacement: in VString) is

    I, N: integer;

begin
    I := 1;
    loop;
        N := Len(Target) - Len(Search) + 1;
    exit when I > N loop
        if Mid(Target,I,Len(Search)) = Search then
            Target := Left(Target,I-1) & Replacement
                      & Right(Target,N-I);
        else
            I := I + 1;
        end if;
    end loop;
end Substitute;
```

Figure 1: String Manipulation module.

Integer: *I 1 Len(Target) Len(Search)*
Len(Target)−Len(Search) Len(Target)−Len(Search)+1
I−1 N N−I I+1

VString: *Target Search Replacement*
Mid(Target,I,Len(Search)) Left(Target,I−1)
Right(Target,N−I) Left(Target,I−1)&Replacement
Left(Target,I−1)&Replacement&Right(Target,N−I)

Boolean: *I>N Mid(Target,I,Len(Search))=Search*

Figure 2: Expression Set for String Manipulation module.

---

2. The second component of EQUATE's set of terms is the set of values taken on by the expression set terms during testing. Actually, only the first value taken on by each expression set term is really of interest, since the point of this component is to force each expression set term to take on at least two different values. Testing must therefore continue until each of these values has taken on a value distinct from that of the expression set term that generated it.

3. The final component is the set of expressions that can be formed by substituting any member of the expression set for any subexpression of another expression set member. Such terms will be called *operand substitution* terms. For the procedure in figure 1, some of the new terms would be *I+1>N*, *I−1>N*, *Mid(Target,Len(Search),Len(Search))*, and *Mid(Left(Target,I−1)&Replacement,I,Len(Search))=Search*. Testing is required to continue until each of these terms has taken on a value distinct from that of the expression set term that generated it. (Note that two substitution terms derived from different expression set terms need not be distinguished from each other.)

At any point in the testing process, the terms that have not yet been distinguished from one another can be portrayed by presenting the tester with a set of equivalence classes for each test location. Each equivalence class contains a set of terms that have had identical values each time that particular test location has been reached during previous test runs. Classes containing only a single term or containing no expression set terms can be eliminated from any further consideration, so the tester's goal becomes the choice of test data that will reduce both the size and the number of the classes remaining at the various test locations.

For modules where the data and operations are user-defined, EQUATE can be shown [18,19] more effective than weak mutation testing [2,3,10], perturbation testing [16,17], simpler expression coverage [7], and the DAISTS testing criterion [6]. Such results would, however, be of little practical interest if EQUATE were so computationally expensive as to

prevent its use. The next section therefore examines the computational complexity of EQUATE.

III. The Complexity of EQUATE

EQUATE may appear, on first inspection, to be inordinately expensive due to its rather heavy computational requirements. This appearance may, however, be deceptive for a number of reasons:

1. Testing appears to be an unavoidably expensive process if we are seeking high levels of confidence in the tested code. Estimates of the high percentage of project resources devoted to testing are well known [1]. Furthermore, the usual approaches to testing are highly labor-intensive while strategies like EQUATE would shift much of the emphasis into machine effort.

2. It is far from obvious just what constitutes an appropriate level of complexity for a testing strategy, since it is not clear how the "error-proneness" of a module is related to such factors as module size that are used as the basis for assessments of testing strategy complexity. Should the number of tests generated by a testing strategy increase linearly with module size, with some polynomial, or with an exponential function? It is probably a safe bet that a linear increase means that larger modules would be insufficiently tested since many factors influencing the complexity of a module (e.g. the number of paths, the number of possible definition-reference interactions) increase at polynomial or exponential rates. An exponential increase in the number of tests with increasing module size would certainly prove discouraging, but we cannot rule out the possibility that this is an appropriate and unavoidable penalty for the writing of unusually large modules.

3. There is no doubt that EQUATE places a heavy demand on the supporting software development environment in which the testing is conducted. It requires access to parsed expressions and access to type information for variables and operators. It requires the ability to frequently interrupt execution, to examine the values of the variables and to evaluate expressions including calls to user-defined functions during such interruptions. For best effect, it should be able to draw upon static data flow analysis and at least a limited form of automatic theorem proving to identify inherently equivalent terms. The cost of providing such a support environment should not, however, be confused with the cost of EQUATE itself. There is growing recognition that environments should provide such capabilities in order to support a wide variety of useful tools including debuggers, design and analysis tools, etc. [14].

4. Finally, it should be noted that the description of EQUATE in the previous section was presented in a manner intended to convey the clearest possible conceptual picture of EQUATE. The implementation may differ considerably from that picture without changing the set of test data that would be required

by the strategy.

With these cautions in mind, this section examines the complexity of EQUATE. The examination takes place in four steps. First, bounds on the number of terms are established. Second, the cost of evaluating the terms and of tracking the term equivalences is discussed. Third, the savings offered by various algorithmic shortcuts are examined. Finally, the cost of detecting inherently equivalent terms is discussed.

The Term Set Size

The term set consists of the expression set, the set of constants denoting the values taken on by the expression set members, and the operand substitution terms.

By definition, the expression set contains every expression and referenced variable name in the program. It follows that N_e, the number of terms in the expression set, cannot exceed the number of operator occurrences plus the number of referenced variable names in the module under test. N_e will therefore grow no faster than linearly with increasing program size, since we would not expect to be able to add new variables and operator calls to the program without a corresponding increase in size. In fact, duplicate expressions, repeated references to the same variable, declarations, or structural information added to the code would increase the program length without increasing N_e. If the program length (measured as the length of the character stream making up the source code) is denoted by L_p, we conclude that $N_e = O(L_p)$.

The number of constant terms is then easily determined. Since a constant will be equivalent to an expression set term only if it denotes the first and the only value taken on by that expression set term, there is at most one constant term corresponding to the initial value of each expression set term, for a total of N_e constant terms.

The number of operand substitution terms, N_s, can be bounded by noting that substitutions always involve replacing some expression set term, a subexpression of a second expression set term, by a third expression set term. The term being replaced must occur as a subexpression of one of the remaining N_e-1 others, but may occur in more than one subexpression of that term (e.g. X occurs twice in $X+(Y*X)$). Even if a limit is placed on the maximum number of parameters taken by any operator, the possibility of a given operator call having the same subexpression for every operand means that N_s could grow exponentially with N_e.

This assessment is, however, unnecessarily harsh. One reason is that in practical programs we would expect the incidence of repeated subexpressions to fall off rapidly with the size of the expressions, because of the natural tendency to replace bulky repetitions by references to "temporary" variables or by isolating the expression within a separate module. An even stronger reason is that the effect of repeated expressions on the program length has not been considered. Since each occurrence of an expression must entail the addition of at least a single operator call or variable name reference to the program, a high incidence of such repetitions would mean that N_e would increase much slower than linearly with increasing program size. In fact, since the number of operand invocations and variable references cannot grow faster than linearly with increasing program size, then the number of places at which operand substitution could take place can grow no faster than linearly with increasing program size. Since the number of substitutions at each such location is N_e-2, and N_e is at worst a linear function of the program size, we can conclude that $N_s = O(L_p^2)$. The total number of terms associated with a program is therefore given by $2*N_e+N_s = O(L_p^2)$.

Complexity of the Basic EQUATE Algorithm

Each of the expression set terms and the operand substitution terms must be evaluated and then examined for equivalences when a test location is reached. Since any subexpression of one of these terms is also in the term set, exactly one operator invocation per term is required in order to evaluate the entire term set. The time complexity of the evaluation of terms at each test location is therefore $O(L_p^2)$, measured in number of operator invocations. Of course, since the operators may include user-defined functions, we cannot, a priori, assign a fixed cost per operator invocation. As a result, the total cost of evaluation at each test location is subject to considerable variation. As will be seen later, however, there are shortcuts that can substantially reduce the number of operator invocations required at each test location.

Once the terms have been evaluated, the classes representing the history of term equivalences at the current test location must be examined. There are two cases to consider in representing the term equivalences.

The first case is that of the expression set terms. Each of these terms must be distinguished from each of the others. Consider a set of such terms that have not yet been distinguished from one another. In the worst case, where all the terms were of the same type and no tests had yet reached this test location, there would be N_e terms in this set.

Assume that we can perform an ordering operation on the underlying representation of the values of these terms such that we can sort the terms by value using this ordering operation and such that equivalent values would be adjacent in the resulting sorted list. (Such an ordering operation should usually be possible at the word/bit level even if no such operation is actually defined on the expression type.) Then this set of expression set terms could be separated into equivalence classes in $O(N_e \log N_e)$ comparisons. If no such ordering operation were available, the separation could still be achieved in $O(N_e^2)$ comparisons.

The second case concerns the constant and substitution terms. These terms need not be compared to each other, but only to the expression set terms from which they were derived. Thus we need a maximum of N_e comparisons of constant terms and N_s comparisons of substitution terms. The total number of comparisons is therefore $O(L_p^2)$. The overall time complexity of EQUATE is therefore no worse than $O(L_p^2)$ operator invocations and $O(L_p^2)$ comparisons each time that a test location is encountered. In practice, the number of terms and hence the number of operator invocations and comparisons tends to shrink quite rapidly with repeated executions of the same test location, because substitution terms may be ignored once they have been distinguished from the related expression set terms.

The space complexity of EQUATE can be determined quite simply. By representing each term using only the name of its root operator and a set of pointers to the terms comprising its operands, the entire term set can be stored in space linearly proportional to the number of terms and to the number of operands of each operator. Thus the term set requires $O(L_p^2)$ space.

The term equivalences at each test location can be represented as a set of equivalence classes. EQUATE can ignore any classes that do not contain at least one expression set term, so the maximum number of classes at a test location is N_e. In fact, the most convenient representation may well be to associate one class with every expression set term, so that the tester's goal becomes distinguishing each term in that class from the expression set term it is associated with. Since constant and substitution terms are associated with a particular expression set term, they need appear only in that expression set term's class. Hence each constant and substitution term need appear only once. If, however, X and Y are two expression set terms, and X and Y have so far always had equal values, then X would be in Y's class and Y would be in X's. (The two classes are not necessarily identical, since each class may contain constant or substitution terms that must be

distinguished from X or Y but not both.) The redundancy of having the X/Y relationship represented in two different classes opens up the possibility of an expression set term appearing in up to N_e classes. If this information were truly redundant, we could, of course, simply choose to represent it only once, selecting only a single class to hold a given equivalence.

The single situation in which an expression set term may really need to appear in more than one equivalence class is when that term is undefined. EQUATE distinguishes between terms that are undefined and terms that are illegal. A term is *illegal* if its evaluation or reference would cause an immediate error. Division by zero, on most machines, would be illegal. A term is *undefined* if its value is unknown but may be legal. For example, the value of a variable prior to its first definition is undefined, unless the execution environment tests for such not-yet-defined variables and raises an error when they are accessed, in which case that variable would be illegal. An illegal object is considered to be distinguished from all other objects (since it is presumed that the error action serves to identify the term). An undefined object, however, cannot be guaranteed to have been distinguished from any legal terms. Consider then the situation where an equivalence class {*A*, *B*, *C*} is examined with *A* and *B* legal but unequal and *C* undefined. EQUATE would maintain that *A* and *B* should be separated, since they have just now been distinguished, but *C* has not yet been distinguished from either *A* or *B*. Consequently we are left with equivalence classes {*A*, *C*} and {*B*, *C*}, with *C* appearing in both classes.

In the worst case then, the term equivalences may require $O(N_e^2)+O(N_s)$ storage, making the total storage requirements for EQUATE $O(L_p^2)$ per test location, or $O(L_p^3)$ for the entire program.

Algorithmic Shortcuts

There are two refinements to a straightforward implementation of EQUATE that can offer substantial computational savings. The first of these is *delayed substitution*. The key idea behind delayed substitution is that the operand substitution term *f(X,Z)* need not be evaluated or compared to the expression set term *f(X,Y)* until the terms *Y* and *Z* have been distinguished from one another. Similarly, the substitution term *g(W,f(X,Z))* need not be evaluated or compared to the expression set term *g(W,f(X,Y))* until *f(X,Z)* has been distinguished from *f(X,Y)*, and so on for more complicated terms. Although delayed substitution does not alter the worst case performance of EQUATE, preliminary experience with a prototype implementation indicates that it offers substantial time and space savings

in normal use.

The second refinement is *symbolic back-substitution*. Consider two test locations separated by an assignment statement $X := f(Y)$. Any pair of terms that do not involve X will be distinguished from one another at the later test location if and only if they are distinguished from one another at the earlier one. A term $g(X)$ will be distinguished from another term at the later location if and only if the term $g(f(Y))$ would have been distinguished from the other term at the earlier test location.

By back-substitution of $f(Y)$ for X we can generate new terms that, if added to the term set of the earlier test location, completely eliminate the need to monitor the equivalences at the later test location. In most cases, we should be able to collapse all of the test locations in a basic block into a single location at the beginning of that block via back-substitution. The penalty for doing this is, of course, the need to process a larger number of terms at that one test location. In addition, some additional bookkeeping may be required to be sure that we do not require terms to be distinguished from one another that would never otherwise have occurred at the same test location.

The computational savings offered by back-substitution may be contingent on the property that most statements affect only a small portion of the term set. Suppose that a given statement changes the value of $q(N_e + N_s)$ terms, where $0 \le q \le 1$. Back-substitution across this statement would create $q(N_e + N_s)$ new terms at the earlier test location. This means a net savings of $(1.-q)(N_e+N_s)$ operator invocations during the evaluation of terms, and of $(1.-q)N_e$ comparisons of constant terms and of $(1.-q)N_s$ comparisons of operand substitution terms. The only part of the EQUATE processing that might be negatively affected would be the comparison of the expression set terms (an expression set term altered by back-substitution must be treated as a new expression set term). Here we are faced with $(1.+q)N_e\log((1.+q)N_e)$ comparisons at the earlier location after back-substitution as opposed to $N_e\log(N_e)$ comparisons each at the earlier and the later location if back-substitution were not employed. Even this represents a net savings for back-substitution for reasonably small values of q. For a very small module with N_e=10, for example, back-substitution results in fewer comparisons between expression set terms if $q \le 0.63$. If N_e=100, back-substitution is cheaper if as much as 77% of the terms are redefined in the intervening statement, and the threshold continues to rise with increasing N_e. When the differences in the other evaluations and comparisons are added in, it seems unlikely that back-substitution would ever fail to provide savings in a practical program.

Detecting Inherent Equivalences

A major concern in a practical implementation of EQUATE is to avoid swamping the person using it with massive, unintelligible lists of expressions. While EQUATE has the advantage that the expressions it deals with are closely related to ones appearing in the code and so should be individually easy to understand,[2] the sheer number of expressions may be overwhelming, especially since many of them are likely to be inherently equivalent and hence of negligible value in guiding the selection of new test data.

For this reason, some means of weeding out inherently equivalent terms is highly recommended. A relatively simple automatic theorem prover should be capable of capturing most of these terms. The author's prototype system has been quite successful by using a symbolic expression simplifier to reduce each term to its "simplest" form and then comparing pairs of simplified terms for identity. When two terms are shown to be inherently equivalent, the more "complex" term can be discarded since it will be distinguished from other terms if and only if its simpler, equivalent version is distinguished from those terms.

Since EQUATE is intended for use with programs employing a variety of data types, many of them user-defined, someone must supply axioms to the prover describing those types. This can occur during the specification, the design, the coding, or even during the actual testing process itself. Obviously if an axiomatic specification of the data types employed is available, it can serve as input to the prover. On the other hand, the person conducting the testing may, while examining the sets of terms listed by EQUATE, notice that a pair of terms that have not been distinguished are in fact inherently equivalent and may decide to add a new axiom that would permit the equivalence to be picked up by the prover during subsequent test runs.

The number of proofs of equivalence attempted by EQUATE depends upon the number of terms per class and the amount of effort we are willing to expend to reduce those classes. If, for example, we simply attempt to prove each term in a class equivalent to the expression set term that class is associated with, then the number of proofs attempted is one less than the number of terms in the class. If, on the other hand, we

[2] To preserve this property, it seems advisable to keep any back-substitution invisible, since the back-substitution expressions may be much more complex and the reasons for their appearance at a particular test location may be hard to fathom. Hence, such expressions should be displayed in their original form, associated with their original test locations, even if they are not manipulated that way internally.

wish to eliminate all redundant information from the class, then attempting to prove the equivalence of each term to each other term in the class would lead to a quadratic rise in the number of proofs. The approach actually taken in the EQUATE prototype is an intermediate one: Each term in the class is simplified, and any two terms are considered inherently equivalent if and only if their simplified forms are identical. This reduces the problem of detecting inherently equivalent terms to a single simplification per term. The prototype simplifier is capable of accepting rewriting rules for user-defined types, including conditional rules (e.g. $A*B<A*C$ can be simplified to $B<C$ if it can be proven that $A>0$.)

IV. An Example of EQUATE Testing

To illustrate the ideas presented in the previous sections, we will step through the testing of the module in figure 1 as guided by EQUATE. Figure 3 shows the body of the

```
. . . . . .
    begin
1.1–    I := 1;
1.2–    loop;
. . . . . .
2.1–        N := Len(Target) – Len(Search) + 1;
2.2–    exit when I > N;
. . . . . .
3.1–        If Mid(Target,I,Len(Search)) = Search then
. . . . . .
4.1–            Target := Left(Target,I–1) & Replacement
4.2–                    & Right(Target,N–I);
. . . . . .
            else
5.1–            I := I + 1;
5.2–
. . . . . .
            end if;
6.1–    end loop;
. . . . . .
7.1–    end Substitute;
```

Figure 3: Test Locations.

procedure from figure 1 with labels indicating the test locations and dotted lines separating the basic blocks. The first digit in each label denotes the block number, and the second indicates the position within the block.

The author has implemented a prototype EQUATE system, which was used to generate the equivalence classes that will be discussed in this example. It had been the author's intention to seed this module with an error (by moving the assignment to *N* to a position just before the loop), but such seeding turned out to be unnecessary because, quite unintentionally, the module already contains a pair of errors.

Suppose now that the module in figure 3 is tested via the call *Substitute('abcde','cd','**')*. At location 1.1, the only expression set terms that can possibly be defined are the parameters *Target*, *Search*, and *Replacement*, their lengths, and the constant *1*. The equivalence classes at location 1.1 are therefore:

{ *Target;* *'abcde'* }

{ *Search;* *'cd'* }

{ *Replacement;* *'**'* }

{ *Len(Target);* *5* }

{ *Len(Search);* *2;* *Len(Replacement)* }

This set of classes tells us that we should try test data that gives some other value to each parameter, that uses strings of different lengths for each parameter, and that gives *Search* and *Replacement* different lengths from each other. On the basis of these requirements we might add a second test, *Substitute('xyz','abc','*')*. This eliminates all equivalences at location 1.1, so we move on to the next location. Location 1.2 has the following equivalence classes:

{ *Len(Search);* *Len(Mid(Target,I,Len(Search)))* }

{ *Left(Target,I−1);* *Left(Target,I−Len(Search));*
Left(Target,I−(Len(Target)−Len(Search)+1)) }

{ *Mid(Target,I,Len(Search))=Search;* *false;*
Mid(Target,Len(Target),Len(Search))=Search; *Mid(Target,Len(Search),Len(Search))=Search;*
Mid(Target,I,1)=Search;
Mid(Target,Len(Target)−Len(Search)+1,Len(Search))=Search;
Mid(Target,I−1,Len(Search))=Search; *Mid(Target,I+1,Len(Search))=Search;*
Mid(Target,I,1),=Search; *Mid(Replacement,I,Len(Search))=Search;*
Target=Search; *Mid(Target,I,Len(Search))=Replacement;*

Mid(Target,I,Len(Search))=Left(Target,I−1); *Replacement=Search;*
Mid(Target,I,Len(Target)−Len(Search))=Search;
Mid(Target,I,Len(Target)−Len(Search)+1)=Search;
Left(Target,I−1)=Search; *Mid(Target,I,I+1)=Search;*
Mid(Target,I,Len(Replacement))=Search }

In the first of these classes, *Len(Search)* can be made different from *Len(Mid(Target,I,Len(Search))* only by a test where *Len(Search)<Len(Target).*

In the second class, the expression set term *Left(Target,I−1)* at this location simplifies to *Left(Target,0)* and so represents an empty string. This class offers a good example of the process of elimination of inherently equivalent terms. Prior to passing it through the simplifier/theorem prover the second class had been:

{ *Left(Target,I−1);* *'';*
Left(Target,I−Len(Target)); *Left(Target,I−Len(Search));*
Left(Target,I−(Len(Target)−Len(Search)+1)); *Left(Target,I−(I+1));*
Left(Target,I−Len(Replacement)); *Left(Target,I−(I+Len(Replacement)));*
Left(Search,I−1); *Left(Replacement,I−1);*
Left(Mid(Target,I,Len(Search)),I−1); *Left(Left(Target,I−1)&Replacement,I−1)* }

Most of these terms were shown to be inherently equivalent to the first one by use of the rule that *Left(S,N)* can be simplified to the empty string if it can be proven that $N \leq 0$. Since the first term lies in the expression set and the others do not, the first term was retained and the others were discarded. Only the two terms that could not be proven equivalent to the first one are left for examination by the tester.

Returning to the reduced version of the second class, since *I* is equal to 1 the second term in this class, *Left(Target,I−Len(Search))*, can be non-empty only if *Len(Search)=0* and *Len(Target)>0*. The term *Left(Target,I−(Len(Target)−Len(Search)+1))* simplifies to *Left(Target,Len(Search)−Len(Target))*, which is non-empty only if *Len(Target)<Len(Search)* and *Len(Target)>0*. Combining these requirements suggests test cases *Substitute('xyz','','•')* and *Substitute('xy','abc','•')*. The first of these tests should also help to reduce the final class, which owes its large size in part to the fact that the expression set term *Mid(Target,I,Len(Search))=Search* has only taken on a single value, *false*.

Executing the first of these new tests reveals one of the errors in this module, the failure to halt when *Search* is empty. We might, upon reflection, decide that substitutions for the empty string should not be allowed and therefore modify the routine as shown in figure 4. This modification adds two new test locations, 0.1 and 8.1, and two new expression set terms, *IsEmpty(Search)* and *not IsEmpty(Search)*. Repeating the previous tests

```
    begin
0.1-    If not IsEmpty(Search) then
1.1-        I := 1;
1.2-        loop;
2.1-            N := Len(Target) - Len(Search) + 1;
2.2-        exit when I > N;
3.1-            If Mid(Target,I,Len(Search)) = Search then
4.1-                Target := Left(Target,I-1) & Replacement
                            & Right(Target,N-I);
4.2-
            else
5.1-            I := I + 1;
5.2-
            end if;
6.1-        end loop;
7.1-        end if
8.1-    end Substitute;
```

Figure 4: Modified Substitute Routine.

---

leaves the following class at location 1.1:

{ *IsEmpty(Search);* *IsEmpty(Replacement);* *IsEmpty(Target)* }

Since *IsEmpty(Search)* must be false at this location, this class can be reduced only by making the other terms true, suggesting the test *Substitute("","x","")*.

The tests done so far reduce the classes at location 1.2 to:

{ *Mid(Target,I,Len(Search))=Search;* *false;*
Mid(Target,Len(Target),Len(Search))=Search; *Mid(Target,Len(Search),Len(Search))=Search;*
Mid(Target,I,I)=Search;
Mid(Target,Len(Target)-Len(Search)+1,Len(Search))=Search;

Mid(Target,I−1,Len(Search))=Search; *Mid(Target,I+1,Len(Search))=Search;* ....
Mid(Target,I,1),=Search; *Mid(Replacement,I,Len(Search))=Search;*
Target=Search; *Mid(Target,I,Len(Search))=Replacement;*
Mid(Target,I,Len(Search))=Left(Target,I−1); *Replacement=Search;*
Mid(Target,I,Len(Target)−Len(Search))=Search;
Mid(Target,I,Len(Target)−Len(Search)+1)=Search;
Left(Target,I−1)=Search; *Mid(Target,I,I+1)=Search;*
Mid(Target,I,Len(Replacement))=Search }

This class has been unchanged because none of the tests done so far have satisfied the condition *Mid(Target,I,Len(Search))=Search* at this test location. The only test for which this condition was true with *I*=1 was the test with the empty Search string, a test which now does not reach this location. Thus we are now forced to construct a test in which the Search string occurs at the start of the Target string, suggesting the test *Substitute('xyx','x','•')*. This reduces the class at location 1.2 to:

{ *Mid(Target,I,Len(Search))=Search;* *Mid(Target,I,1)=Search;*
Mid(Target,Len(Target),Len(Search))=Search; *Mid(Target,Len(Search),Len(Search))=Search;*
Mid(Target,Len(Target)−Len(Search)+1,Len(Search))=Search;
Mid(Target,I,Len(Replacement))=Search }

In essence, this class argues that the last test was too much of a special case. It states that the target string began with the search string (*Mid(Target,I,Len(Search))=Search*) exactly when the first character of the target string was the whole search string (*Mid(Target,I,1)=Search* and *Mid(Target,Len(Search),Len(Search))=Search*) and when the target string also ended in the search string (*Mid(Target,Len(Target),Len(Search))=Search* and *Mid(Target,Len(Target)−Len(Search)+1,Len(Search))=Search*) and when the lengths of the search and replacement strings were equal (*Mid(Target,I,Len(Replacement))=Search*). We can eliminate all of these terms by testing with *Substitute('abcde','ab','•')*. After these tests there are no remaining equivalences at location 1.2.

The classes at location 2.1 can involve many more terms than the classes at the earlier locations because 2.1 is the first location where *N* can be defined and where *I* can take on values other than *1*. Because, however, location 2.1 occurs within the loop, it is reached far more often than are the earlier locations. As a result, relatively few equivalences remain at 2.1. The equivalence classes are:

{ *IsEmpty(Search);* *IsEmpty(Left(Target,I−1)&Replacement&Right(Target,N−I))* }

{ *I>N;* *I>Len(Target)−Len(Search)+1* }

In the first class, *IsEmpty(Search)* must always be false at this location, so the second term

must be made true. The second term represents a test to see if the entire *Target* could be replaced by the empty string. To make this term true, we need a test where *Replacement* is empty and where *Target* and *Search* have the same length. The second class states that the loop exit condition from the previous iteration, *I>N*, has always been equivalent to the new exit condition, *I>Len(Target)−Len(Search)+1*, for the current iteration. In order to distinguish these two terms, we need a test where a substitution reduces the length of *Target* by enough to immediately force a loop exit if *I* is currently less than or equal to *N* or where a substitution increases the length of *Target* enough to force additional iterations if *I* is currently greater than *N*. Both of these classes can therefore be eliminated by the test *Substitute('xyz','xyz','')*.

There are no remaining equivalences at locations 2.2 and 3.1. This is not particularly surprising since these locations are reached almost as often as is 2.1 and with very similar states. More surprisingly, there are no remaining equivalences at location 4.1 even though this location has not been reached very often. Location 4.2, however, has two equivalence classes remaining:

```
{ I>N;                                          Len(Target)−Len(Search)+1>N;
  Len(Target)−Len(Search)>N) }

{ Mid(Target,I,Len(Search))=Search;              false;
  Mid(Target,Len(Search),Len(Search))=Search;    Mid(Target,I,I)=Search;
  Mid(Target,I−1,Len(Search))=Search;            Mid(Target,N−I,Len(Search))=Search;
  Mid(Target,I+1,Len(Search))=Search;            Mid(Target,I,1)=Search;
  Mid(Target,I,N)=Search;                       Mid(Target,I,Len(Target))=Search;
  Mid(Replacement,I,Len(Search))=Search;         Target=Search;
  Mid(Right(Target,N−I),I,Len(Search))=Search;
  Mid(Target,I,Len(Left(Target,I−1)&Replacement))=Search;
  Mid(Target,I,Len(Left(Target,I−1)&Replacement&Right(Target,N−I)))=Search;
  Mid(Target,I,Len(Target))=Search;             Mid(Target,I,Len(Target)−Len(Search))=Search;
  Mid(Target,I,Len(Target)−Len(Search)+1)=Search;
  Mid(Target,I,N−I)=Search;                     Mid(Target,I,I+1)=Search;
  Left(Target,I−1)=Search;                      Right(Target,N−I)=Search;
  Left(Target,I−1)&Replacement=Search;
  Left(Target,I−1)&Replacement&Right(Target,N−I)=Search;
  Mid(Target,Len(Target)−Len(Search),Len(Search))=Search;
  Mid(Target,I,I−1)=Search;                     Mid(Target,I,Len(Left(Target,I−1)))=Search;
  Replacement=Search;                           Mid(Target,I,Len(Replacement))=Search;
  Mid(Left(Target,I−1)&Replacement,I,Len(Search))=Search) }
```

In the first class, *I>N* must always be false at this location, so we must attempt to make the other terms true. The second term can be true only when a substitution increases the length of *Target*, and the third term can only be true if the increase is by more than one

character. The second class appears to be discouragingly large until we note that the first term has always been false at this location and so has taken on only a single value. If we can make the first term true, most of the others should disappear. This first term is simply the test to see if *Search* appears at the *I*th position in *Target*. For this condition to be true at this location, we must make two successive substitutions without incrementing *I*. This in turn requires that *Replacement* begin with *Search*, suggesting the test *Substitute('abcde','bc','bcdef')*. Executing this test, however, reveals a second error in the module, a failure to terminate whenever *Search* occurs in *Replacement*. The correction involves adding the assignment *I := I + Len(Replacement)* just after location 4.2.

Although this assignment adds two new terms, *Len(Replacement)* and *I+Len(Replacement)*, to the expression set and alters the number of times that some of the locations are reached, no new equivalences are introduced at locations 0.1, 1.1, 1.2, 2.1, 2.2, 3.1, or 4.1. At location 4.2 a single class remains:

```
{ Mid(Target,I,Len(Search))=Search;
  Mid(Target,Len(Search),Len(Search))=Search;          Mid(Target,I,I)=Search;
  Mid(Target,N-I,Len(Search))=Search;                  Mid(Target,I,N-I)=Search) }
```

This class illustrates that the only circumstances under which a match occurred immediately after a substitution has been when two of *I*, *Len(Search)*, and *N-I* have been equal. This can be relieved by the use of the test case *Substitute('xyyxyyx','y','yx')*. This eliminates the equivalence class at 4.2. There are no classes left at 4.3, so we move on to the locations in block 5.

The test locations in block 5 are reached only when there is no match at the *I*th position of *Target*. There are two classes remaining at location 5.1. The first of these is:

```
{ IsEmpty(Search);          IsEmpty(Replacement);
  IsEmpty(Left(Target,I-1)&Replacement) }
```

We can only reach this location if *IsEmpty(Search)* is false, but the other terms in this class can be made true by a test such as *Substitute('xx','y','')*. The other class remaining at location 5.1 is:

```
{ Mid(Target,I,Len(Search))=Search;          Replacement=Search;
  Mid(Target,I-1,Len(Search))=Search;        Mid(Target,I,Len(Search))=Replacement;
  Left(Target,I-1)=Search;                  Left(Target,I-1)&Replacement=Search;
  Left(Target,I-1)&Replacement&Right(Target,N-I)=Search;
  Mid(Target,1,Len(Search))=Search) }
```

The expression set term *Mid(Target,I,Len(Search))=Search* must always be false when we

reach this location, so we must attempt to make the other terms become true. The second term can obviously be made true by the test *Substitute('y','x','x')*. The third term can be made true by *Substitute('xy','x','x')* (the term becomes true when *I*=2). The fourth term is made true by *Substitute('x','y','x')*. As it happens, these tests also eliminate the remaining terms of this class. Furthermore, there are no remaining classes at locations 5.2 or 6.1.

Location 7.1 has two remaining classes. The first is

```
{ Right(Target,N-I);          Right(Target,Len(Replacement)-I) }
```

Since this location is reached only when *I*>*N*, the first term must always evaluate to the empty string. The second term can be non-empty by testing with a replacement string longer than the final value of *I*. This suggests the test *Substitute('x','y','xyz')*.

The second class at location 7.1 is

```
{ Mid(Target,I,Len(Search))=Search;          Target=Search;
  Mid(Replacement,I,Len(Search))=Search;     Mid(Target,I-1,Len(Search))=Search;
  Left(Target,I-1)=Search;                  Left(Target,I-1)&Replacement=Search;
  Mid(Target,Len(Target),Len(Search))=Search) }
```

The second term can be distinguished from the first by the test *Substitute('x','x','x')*, and the third term can be distinguished from the first by the test *Substitute('','x','x')*. These tests also serve to eliminate the other terms in this class.

There are no remaining equivalence classes at location 8.1, so the test set is now finished. The complete test set is summarized in figure 5. It is worth noting that the two errors uncovered by these tests were not found by serendipity, but that the conditions for forcing their discovery were logically implied by EQUATE's rules for distinguishing terms.

V. Conclusions

In view of the importance of data and functional abstraction to modern methods of software design and implementation, testing strategies supporting and taking advantage of user-defined types and operations are badly needed. The EQUATE testing strategy represents a means of determining when a statement has been exercised through a sufficient range of program states, where these states are revealed through the values of the data and operations on data employed throughout the module under test.

Target	Search	Replacement
'abcde'	'cd'	'**'
'xyz'	'abc'	'*'
'xyz'	''	'*'
'xy'	'abc'	'*'
''	'x'	''
'xyx'	'x'	'*'
'abcde'	'ab'	'*'
'xyz'	'xyz'	''
'abcde'	'bc'	'bcdef'
'xyyxyyx'	'y'	'yx'
'xx'	'y'	''
'y'	'x'	'x'
'xy'	'x'	'x'
'x'	'y'	'x'
'x'	'y'	'xyz'
'x'	'x'	'x'
''	'x'	'x'

Figure 5: Complete Test Set for Substitute Routine.

EQUATE selects a set of terms to be distinguished at each test location. This set consists of all expressions and subexpressions appearing in the module under test (the expression set), the constants denoting values taken on by the expression set terms during execution, and the expressions that can be formed by replacing a subexpression of any expression set term by a second expression set term (the operand substitution terms). The expression set terms must each be distinguished from one another. The constants and substitution terms must be distinguished from the related expression set term.

The time complexity of EQUATE, measured in terms of the number of function evaluations required on reaching any test location, is at worst $O(L_p^2)$ where L_p is the length of the program. The space complexity of EQUATE is at worst $O(L_p^3)$. Significant speedups and space savings can be obtained via delayed substitution and symbolic back-substitution, although the worst case order of complexity is not affected.

Current work on EQUATE is concentrating on expansion of the prototype implementation and on the evaluation of its effectiveness on a variety of programs, as compared to a number of other testing strategies.

References

1. B. W. Boehm, "Software and Its Impact: A Quantitative Assessment," *Datamation*, May 1973

2. T. A. Budd, "Mutation Analysis: Ideas, Examples, Problems and Prospects," *Computer Program Testing*, B. Chandrasekaran and S. Radicchi (eds.), North-Holland Publishing Co., 1981, pp. 129-148

3. T. A. Budd, *The Portable Mutation Testing Suite*, University of Arizona technical report TR 83-8, March 1983

4. L. A. Clarke, J. Hassell, and D. J. Richardson, "A Close Look at Domain Testing," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, 380-390, July 1982

5. K. A. Foster, "Error Sensitive Test Cases Analysis (ESTCA)," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 3, 258-264, May 1980

6. J. Gannon, P. McMullin, and R. Hamlet, "Data-Abstraction Implementation, Specification, and Testing," *ACM TOPLAS*, vol. 3, no. 3, 211-223, July 1981

7. R. G. Hamlet, "Testing Programs with the Aid of a Compiler," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 4, 279-290, July 1977

8. W. E. Howden, "Methodology for the Generation of Program Test Data," *IEEE Transactions on Computers*, vol. C-24, no. 5, 554-560, May 1975

9. W. E. Howden, "Algebraic Program Testing," *Acta Informatica*, vol. 10, 53-66, 1978

10. W. E. Howden, "Weak Mutation Testing and Completeness of Test Sets," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, July 1982, 371-379

11. J. W. Laski and B. Korel, "A Data Flow Oriented Program Testing Strategy," *IEEE Transactions on Software Engineering*, vol. SE-9, no. 3, 347-354, May 1983

12. S. J. Ntafos, "On Required Elements Testing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 6, 795-803, November 1984

13. S. Rapps and E. J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 4, 367-375, April 1985

14. R. N. Taylor, L. A. Clarke, L. J. Osterweil, J. C. Wileden, and M. Young, "ARCADIA: A Software Development Environment Research Project," *IEEE Computer Society Second International Conference on Ada Applications and Environments*, April 1986

15. L. J. White and E. I. Cohen,"A Domain Strategy for Computer Program Testing," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 3, 247-257, May 1980

16. S. J. Zeil, "Testing for Perturbations of Program Statements," *IEEE Transactions on Software Engineering*, SE-9, No. 3, May 1983, pp. 335-346

17. S. J. Zeil, "Perturbation Testing for Computation Errors," *Seventh International Conference on Software Engineering*, March 1984, IEEE, also University of Massachusetts Technical Report 83-23, July 1983

18. S. J. Zeil, *Testing for Equivalent Algebraic Terms - EQUATE*, COINS Technical Report 85-04, February 1985, University of Massachusetts

19. S. J. Zeil, "The EQUATE Testing Strategy", *Workshop on Software Testing*, July 1986, Banff Canada, IEEE