Testing for Equivalent Algebraic Terms – EQUATE[1]

Steven J. Zeil

COINS Technical Report 85-04
February 1985
Revised: November 1985

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

---

[1] This work was supported by grants MCS-8303320 and DCR-8404217 from the National Science Foundation, 84M103 from Control Data Corporation, and SCEEE-PDP/85-0037 from the Southeastern Center for Electrical Engineering Education. This paper has been submitted to the IEEE Transactions on Software Engineering.

Abstract

This paper introduces a new testing strategy, EQUATE testing. EQUATE represents an attempt to merge the strengths of perturbation testing and mutation testing in order to provide a testing strategy that offers support for data and functional abstraction, that detects a wide variety of simple faults, and that also provides good coverage of combinations of those simple faults. 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 each pair of these expressions from one another at every test location.

Keywords: Testing, Perturbation Testing, Mutation Testing

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 [3,6,9,11,16,17,18]. This is hardly surprising, since few areas of theoretical research would make much progress without similar simplifying assumptions, at least in their beginning stages. Perhaps the most common restriction for testing strategies has been to restrict their operation to numeric data and to the standard arithmetic operators. Unfortunately, as support for data and functional abstraction becomes more widely available both in language constructs and in design methodologies, this restriction becomes less palatable.

This paper introduces a new testing strategy, EQUATE testing. EQUATE represents an attempt to merge the strengths of perturbation testing [19,20,21] and mutation testing [1,2,4,5,12] while also attempting to develop a method which is more compatible with the ideas of modularity and abstraction which underlie much of software engineering. The primary goals advanced for this testing method are:

1. to provide a testing strategy that is useful for modules employing varying levels of data and functional abstraction;

2. to provide a testing strategy that both detects a wide variety of simple faults and also provides good coverage of combinations of those simple faults.

The next section describes EQUATE in its "pure" form, the form most suited to understanding its power and capabilities. Section III discusses the practical problems involved in implementing EQUATE and some important shortcuts which greatly reduce the required computations. Section IV describes its relation to other testing strategies, especially perturbation and mutation testing. Finally, section V describes some possible avenues for future research.

II. Conceptual View of EQUATE Testing

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, in effect specifying certain paths or classes of paths to be executed [10,13,14,15]. 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 to this idea is proposed: *Each valued object (variables, constants, and expressions) in the module*

should take on values that, on at least one execution, can be distinguished from those of any other such object and from those of any constant (not necessarily appearing in the module). 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.

The EQUATE strategy therefore 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 then separated into equivalence classes. Each equivalence class will contain only those terms with equal values when evaluated at that test location. Thus, each time that a test location is reached, a partition is defined on the set of terms. A new set of equivalence classes are formed from the intersection of the partitions obtained each time that same test location is reached. Any two terms remaining in the same class have been equal for every test. Subsequent test data should be chosen to give those terms non-equal values at that location.

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,[2] for example, has the expression set shown in figure 2. Testing is required to continue until, at each test location, all members of the expression set have taken on distinct values at least once and therefore have been separated into different equivalence classes.

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 been separated from the expression set term that generated it.

---

[2] In this listing, and elsewhere in this paper, □ is used to denote a single blank character.

```
package Variable_Length_Strings is
    type VString is private;
    function String_to_VString (S: string) return VString;
    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;
private
    .
    .
    .
end Variable_Length_Strings;
    .
    .
    .
with Variable_Length_Strings; use Variable_Length_Strings;
procedure Compress_Double_Blanks (S: in out VString) is

    I: integer;

begin
    I := 1;
    while  I < Len(S) loop
        if Mid(S,I,2) = String_to_VString("□□") then
            S := Left(S,I-1) & Right(S,Len(S)-I);
        else
            I := I + 1;
        end if;
    end loop;
end Compress_Double_Blanks;
```

Figure 1: String Manipulation module.

Integer: I 1 Len(S) 2 I-1 Len(S)-I I+1

String: "□□"

VString: S Mid(S,I,2) String_to_VString("□□")
Left(S,I-1) Right(S,Len(S)-I)
Left(S,I-1)&Right(S,Len(S)-I)

Boolean: I<Len(S) Mid(S,I,2)=String_to_VString("□□")

Figure 2: Expression Set for String Manipulation module.

---

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. This procedure will be called *operand substitution*. Thus, for the procedure in figure 1, some of the new terms would be I+1<Len(S), I−1<Len(S), Mid(S,Len(S)−I,2), and Mid(Left(S,I−1)&Right(S,Len(S)−I),I,2). Testing is required to continue until each of these terms has been separated from 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.)

The inclusion of the expression set and the values of the expression set terms by EQUATE is clearly related to the goal of distinguishing each valued object in the program (i.e., each expression set term) from every other object and from any constant (the values taken on by the expression set terms). To understand the role of the substitution terms, it is necessary to return to the question of what it means to distinguish one object from another.

Clearly a necessary condition for distinguishing two objects is that the strings of bits representing the values of the 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. Thus 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 data 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 operations on X, X is distinguished from Y only if f(X) is distinguished from f(Y). If, for example, X were a floating point number and the 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 operation 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 operation 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 operation f, then X and Y are distinguished if $X \neq Y$.

While this way of limiting the set of operations on expression set terms is a natural solution, it is not the only reasonable approach. The very name "operand substitution" suggests the possibility of "operator substitution" as an additional or as an alternate source of terms. Some data types (e.g. integers) may have a massive number of legal operators and functions. It may be quite common that only a fraction of these would be reasonable for any given module. (Consider, for example, the possibility of pulling some abstract data type from a library of frequently-used routines. A considerable number of operations on that type may have been added for use by other projects that required a fuller set of operations than the module currently under test.) Operator substitution would therefore be most useful if operators could be grouped into classes of related functions from which substitutes would be chosen. Such classes might be suggested by the programmer based upon a perceived relationship among the operators (e.g. the relational operators, or the set of standard trigonometric functions). Where the language or environment supports the grouping of related functions (e.g. the ADA package), such groups could be taken as the basis for classification. This form of operator substitution may be considered a likely candidate as a fourth component (possibly optional) of EQUATE's set of terms, but has not yet been explored further.

Many of the operations on a given object X may involve other parameters besides X. If we wish to distinguish X from Y, and the operation $f(X,Z)$ appears in the module, what requirement on f is imposed in order to distinguish X from Y? Clearly it would be impractical to try all possible values for the second parameter of f. A natural solution is to use exactly those expressions that actually appeared in the code invoking $f(X,Z)$. In effect, the actual operation on X is $g_z(x) = f(x,Z)$, and we are requiring $g_z(X)$ to be distinguished from $g_z(Y)$.

The arguments presented here describe exactly the set of terms identified earlier as the operand substitution terms. It should also be clear from this discussion why each operand substitution term is only required to be distinguished from the expression set term it was from which it was derived and not from all other substitution terms. Substitution will usually be responsible for the bulk of the terms to be employed. Indeed, operand substitution may appear to generate a prohibitively large number of new terms. It is possible that, for some applications, testing with the expression set and constants without substitutions would prove adequate. This would be most likely for very large modules, where the number and variety of expressions and subexpressions is much greater. Keep in mind, however, that this section is presenting the conceptual view of EQUATE without regard to computational efficiency. Section III will introduce means of reducing the number of substitutions to be handled at any given moment.

To conclude this section, consider some possible errors and the way in which EQUATE might lead to their detection. Figure 3 shows the body of the 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. There are a variety of places where "off-by-1" errors could occur. Consider the expression Right(S,Len(S)−I). EQUATE would catch such an error by requiring at least one test on which Right(S,Len(S)−I) gives a value different from Right(S,Len(S)−(I+1)) and Right(S,Len(S)−(I−1)) at test location 4.1. Similar requirements occur for all expressions involving I. One might legitimately object that, although EQUATE would catch off-by-1 errors in this module, it will not do so in other modules where "+1" and "-1" operations do not occur as part of the expression set. It is this author's

```
. . . . . .
    begin
1.1-
        I := 1;
1.2-
. . . . . .
2.1-
        while  I < Len(S) loop
. . . . . .
3.1-
            if Mid(S,I,2) = String_to_VString("□□") then
. . . . . .
4.1-
                S := Left(S,I-1) & Right(S,Len(S)-I);
4.2-
. . . . . .
            else
5.1-
                I := I + 1;
5.2-
. . . . . .
            end if;
6.1-
        end loop;
. . . . . .
7.1-
    end Compress_Double_Blanks;
```

Figure 3: Test Locations.

contention, however, that this type of error is most likely to occur in precisely those modules where the constituent operations do include incrementing and decrementing by 1, operations which EQUATE will then use to advantage.

Another interesting error would occur if the operation "I := I + 1;" were moved outside of the IF construct, so that it is performed each time through the loop. Such an error would be detectable only with test data containing a sequence of three or more consecutive blanks. Such test data would be required by EQUATE since, in order to separate it from the constant denoting its first value, every term in the expression set must take on at least two distinct values. Thus the boolean expression Mid(S,I,2)=String_to_VString("□□") must take on both true and false values at every test location. For this to occur at location 4.2, the procedure must be tested with data containing a string of more than two blanks. This error illustrates the usefulness of basing the testing requirements at each location on all the terms, rather than just the terms appearing at that particular location.

Other interesting test cases will arise from the fact that this module has expressions that count up from 0 and that count down from the length of S. The principal examples of these are I and Len(S)-I. EQUATE will require asymmetric strings in order to

distinguish terms like Left(S,I) from Right(S,Len(S)-I) or Mid(S,I,2) from Mid(S,Len(S)-I,2). An additional requirement will be that the patterns of double blanks be asymmetric to distinguish Mid(S,I,2)=String_to_VString("□□") from Mid(S,Len(S)-I,2)=String_to_VString("□□") at various locations throughout the module. These requirements test the direction of the scan performed by the module and its relation to the substring extraction operations.

Many of the test cases required by EQUATE compare quite favorably with intuitive notions of good testing practice, such as rule-of-thumb procedures for exercising loops. For example, the fact that I<Len(S) must take on both true and false values at location 2.1 means that test data is required both to enter and to bypass the loop. The test case that bypasses the loop must be one for which the length of S is no greater than 1; thus the module is required to be tested with inputs of minimal size. The empty string must be used as a test case since I and Len(S) can only take on different values at location 7.1 if the loop is never entered and if Len(S)≠1. Tests are also required that execute the loop more than once since I<Len(S) must at least once be true at location 6.1. In fact since I<Len(S) must take on both true and false values at locations 4.2 and 5.2, tests will be required that both repeat the loop after each branch of the "if" and that leave the loop immediately after each of those branches.

III. Implementation Considerations

In this section, the EQUATE scheme is reviewed with an eye towards the issues and problems involved in implementing it with reasonable efficiency and with minimal restrictions on the programming language constructs to which it may apply. This section therefore discusses the ways in which EQUATE is defined for practical programming languages, some algorithmic shortcuts to improve efficiency, and the types of supporting analysis and tools required for use with EQUATE. This section then concludes with an extended example of the use of EQUATE on the program presented earlier in figure 1.

Programming Language Constructs

The major activity associated with EQUATE is the determination of whether a pair of terms are equivalent in the current program state. A definition of what it means for two terms to be equivalent is therefore important. The major criterion for equivalence is that the values of the two terms must be represented by the same string of bits (or other underlying representation). Additional criteria for equivalence may be imposed by a particular language. In languages with strong typing, two terms can be judged equivalent only if they have the same type. In fact, a pair of terms having different types may be considered to initially lie in separate classes provided that there is no possibility for operand substitutions involving the two types. Additional provisions must be made for derived and constrained types, if they exist in the language.

Another difficulty with determining equivalence of terms is the possibility that a term may be undefined or illegal at some points in the testing process. A term is *undefined* at some point during execution if any variable referenced by that term has not been assigned a value since the start of the module or since any statement which "undefined" that variable. A term is *illegal* in some program state if the process of evaluating that term involves some prohibited operation (such as division by zero) which would cause a run-time

error. Two terms will be considered *equivalent* in some program state if both return the same data type and if neither is illegal in that state and if either is undefined or both have equal values in that state. The idea behind this definition is that terms that, if evaluated at that point, would cause a run-time error may be considered to have been distinguished from all legal terms, but a term whose value is simply unknown cannot be guaranteed to be distinct from any defined term. If the module being tested is intended for use in an environment where references to undefined variables are detected and flagged as errors, then such undefined terms are actually illegal and may be considered to be distinguished from all defined terms.

Other practical problems with EQUATE involve determining a reasonable interpretation of various language constructs. For example, references to array elements/record fields are probably best treated as expressions involving an indexing/selection operator with the array/record as one operand and the indices/field as the remaining operands. A more substantial problem is posed by procedure calls. Clearly the parameters of a procedure call are themselves members of the expression set and so will enter into the testing requirements. The procedure call itself is more problematic. One possibility is to regard it as if it were an I/O statement and to be satisfied simply with testing based upon its operands. For many user-defined data types, however, the choice between implementing the principal operations as functions or as procedures may, for whatever reason, be resolved in favor of procedures. Ideally, such decisions should not substantially alter the testing requirements, but in this case the use of procedures would substantially reduce the set of operations available to EQUATE for generating new terms. A better solution, therefore, is to treat a procedure call such as PROC(x,y,z) with input parameter x, output parameter z, and in-out parameter y as an extended assignment statement "y := arg2(PROC(x,y,z)); z := arg3(PROC(x,y,z))", with the assignments being simultaneous. The advantage of treating procedures in this fashion is that the procedures themselves become part of the expression set, and substitutions of procedure parameters become subject to testing.

Supporting Analysis and Tools

EQUATE is not intended to be applied manually, nor even as a stand-alone testing tool. Instead, the tools and support facilities of a good programming environment are presumed available. These include a parser to generate the expression set, tools for data flow analysis and symbolic manipulation of expressions, the ability to break execution and save the current module state for later restoration (after the evaluation of expressions that may include calls to user-defined functions, that may in turn have various side-effects), and possibly a simple theorem prover.

Some terms may always be undefined at particular test locations. If, for some term, there is no legal path from the start of the module to a test location on which all variables appearing in that term are defined, then there is no point to including that term in the testing criteria for that location. The majority of these terms can be identified via static data flow analysis.

Some terms may always be equal to some other term (including a constant) at a particular test location. Identifying such equivalences and treating the resulting group of terms and constants as a single item might lead to some run-time savings, but, more importantly, it would allow a testing tool to avoid telling the tester to seek data that

would distinguish two indistinguishable terms. Detecting such equivalences automatically might seem to require a sophisticated theorem prover, but in fact a great deal can be done with simpler tools that examine no more than one basic block at a time. Most functionally equivalent terms will result from assignments such as "X := f(Y)" or from passing through conditional statements such as "if X = f(Y)". At the test location immediately following this statement, the terms X and f(Y), both of which should be in the expression set, will be equivalent no matter what data is employed. This equivalence will continue to hold for all succeeding test locations within the same block until either X or Y is redefined. The same will be true for any terms of the form g(X,Z) and g(f(Y),Z). Most such equivalences can be determined via symbolic execution of the basic block, combined with rudimentary simplification of those expressions that involve the standard arithmetic operators.

The prospect of introducing additional analysis into what may appear to be an already expensive testing method may be somewhat discouraging, but note that the data flow and symbolic analyses described above operate on the static form of the module and hence are done only once at the start of testing. There are, however, substantial advantages to allowing some of these analyses to be done throughout the testing process. In particular, a simplifier or theorem prover will require simplification rules or axioms for the abstract data types appearing in the module under test. If these are not supplied as part of a formal (algebraic or axiomatic) specification of the module, then the tester should be allowed to supply such rules during testing. The EQUATE terms can provide motivation and guidelines for such rules, as will be demonstrated shortly.

Efficient Implementation of EQUATE

The use of data flow analysis and symbolic evaluation can also help to improve the efficiency of EQUATE by combining certain classes from several test locations within some block into a single test location at the start of that block. As an example of this, consider test locations 1.1 and 1.2 in figure 3. The intervening statement assigns a value to the variable I but leaves the other variables unchanged. Let X and Y be any two terms. If neither X nor Y uses the variable I, then X and Y will be in the same equivalence class at location 1.2 if and only if they are in the same equivalence class at location 1.1. If, therefore, we have an equivalence class at 1.2 in which none of the terms use I, then we can ignore that class because the tests required to distinguish its terms from one another will also be required at 1.1.

A generalization of this technique for eliminating certain classes involves back-substituting for the redefined variables, thereby collapsing the testing criteria for several locations into an expanded set of terms at the start of the block. For example, the terms affected by the assignment statement between 1.1 and 1.2 are I, I+1, I−1, Len(S)−I, Mid(S,I,2), Left(S,I−1), Right(S,Len(S)−I), Left(S,I−1)&Right(S,Len(S)−I) and I<Len(S). These are distinct at 1.2 from each other and from the remaining terms of the expression set if the terms 1, 1+1, 1−1, Len(S)−1, Mid(S,1,2), Left(S,1−1), Right(S,Len(S)−1), and Left(S,1−1)&Right(S,Len(S)−1) are distinct from those other terms when evaluated at 1.1. If these terms (and the associated operand substitutions) are added to the set of terms for location 1.1, then nothing at all needs to be done to monitor testing at location 1.2. The use of back-substitution may save a tremendous amount of computation if the percentage of terms affected by a single statement is small, although in the worst case it would substantially increase the amount of computation.

It is possible for back-substitution to make the testing criteria more stringent by forcing the testing of terms which previously would not have been considered together. For example, the terms I from 1.1 and Len(S)−1 from 1.2 do not properly belong together, and before the use of back-substitution there would not have been a requirement for test data to distinguish the two. Such a requirement can be avoided by associating with each term a list of the locations at which it is valid, and reporting only those equivalences involving terms from the same locations. In fact, it may be a good idea to report equivalences as if back-substitution were not being employed, listing separate classes for different locations, so that the user is still presented with the conceptual view of EQUATE.

Perhaps the single most expensive-looking aspect of EQUATE, both in terms of execution time and space, is the sheer number of terms generated by operand substitution. This number can be dramatically reduced by delaying the substitution process. Consider two variables X and Y, and an expression set term f(X,Z). By operand substitution, we get a new term f(Y,Z) and a requirement for test data distinguishing f(X,Z) from f(Y,Z). Note however, that f(X,Z)=f(Y,Z) whenever X=Y. Consequently we need not begin to check f(Y,Z) until we have first distinguished X from Y. Now suppose further that f(X,Z) consists of a single operator or function with X as one of its operands (though Z may actually be an arbitrarily complex expression) and that there is a term in the expression set g(f(X,Z),W). When X is distinguished from Y, we need not automatically consider the term g(f(Y,Z),W). Instead, we only generate the operand substitution terms like f(Y,Z) involving a single operation on X and Y. The more complicated substitution term g(f(Y,Z),W) need only be generated when f(X,Z) is distinguished from f(Y,Z) (which may be immediately true or may occur only after additional tests). Note also that the term f(Y,Z) will cease to be of interest when this occurs, except in as much as its value is required as an operand to g, and so need no longer appear in any equivalence classes.

Thus, when any two terms are distinguished, we generate the operand substitution terms involving a single operation applied to one of the newly distinguished terms. If each of these new terms is equivalent to its corresponding term from the expression set, then the process stops. If a new term is not equivalent, then that term and the original one are used to generate another level of operand substitution terms. This continues until an equivalence is found or until neither of the newly distinguished terms is a subexpression of a member of the expression set. Halting is guaranteed because the terms created in this manner are growing more and more complicated, and so we cannot continue indefinitely forming new expressions in this manner before they cease to be members of the expression set.

An Example of EQUATE

To illustrate the ideas presented in this section, consider again the procedure in figure 3. We will step through the testing of this module as guided by EQUATE, taking advantage of prior data flow analysis, delayed operand substitution and the use of a symbolic evaluator that can accept new simplification rules interactively and has some very limited theorem proving abilities for dealing with relational expressions. These capabilities have been implemented by the author in a prototype EQUATE system, which was used to generate the equivalence classes shown here.

Suppose that the module in figure 3 is tested with input data S="a□□bc". At location 1.1, the only expression set terms that can possibly be defined are S, Len(S) and String_to_VString("□□"), so the resulting equivalence classes are {S "a□□bc"}, {Len(S) 5} and {String_to_VString("□□") "□□"}. Thus we know that we need an additional test where S has a different value and a different length. The third class would be of interest only if the String_to_VString function could return different values given the same inputs. While such behavior may be prohibited in mathematical functions, it is far from unknown in most programming languages (The most common example would probably be functions associated with input from external devices). It is useful to be able to flag "pure" functions and procedures like String_to_VString so that such classes are not reported. Assuming this is done, since the remaining two classes at this location do not provide much information about the required test data, we move on to location 1.2.

The non-trivial equivalence classes (i.e. those containing at least two terms, with at least one of those belonging to the expression set) for location 1.2 are:

{ S "a□□bc" }

{ Len(S) 5 }

{ Len(S)-I 4 }

{ Mid(S,I,2) "a□" }

{ Left(S,I-1) "" Left(String_to_VString("□□"),I-1) Left(Mid(S,I,2),I-1)
Left(Right(S,Len(S)-I),I-1) Left(Left(S,I-1)&Right(S,Len(S)-I),I-1) Left(S,I-2)
Left(S,I-(Len(S)-I)) Left(S,I-Len(S)) }

{ Right(S,Len(S)-I) Left(S,I-1)&Right(S,Len(S)-I) □□bc
Right(Left(S,I-1)&Right(S,Len(S)-I),Len(S)-I) }

{ I<Len(S) true I+1<Len(S) I-1<Len(S) I<Len(Mid(S,I,2))
I<Len(Right(S,Len(S)-I)) I<Len(Left(S,I-1)&Right(S,Len(S)-I)) }

{ Mid(S,I,2)=String_to_VString("□□") false Mid(S,I,1)=String_to_VString("□□")
Mid(S,Len(S),2)=String_to_VString("□□") Mid(S,I-1,2)=String_to_VString("□□")
Mid(S,Len(S)-I,2)=String_to_VString("□□") Mid(S,I,2)=S S=String_to_VString("□□")
Mid(Left(S,I-1),I,2)=String_to_VString("□□") Mid(S,I,Len(S))=String_to_VString("□□")
Mid(S,I,I-1)=String_to_VString("□□") Mid(S,I,Len(S)-I)=String_to_VString("□□")
Mid(S,I,2)=Left(S,I-1) Mid(S,I,2)=Right(S,Len(S)-I) Mid(S,I,2)=Left(S,I-1)&Right(S,Len(S)-I)
Left(S,I-1)=String_to_VString("□□") Right(S,Len(S)-I)=String_to_VString("□□")
Left(S,I-1)&Right(S,Len(S)-I)=String_to_VString("□□") }

The first four classes are, like those at 1.1, reflections of the need for each expression to take on non-constant values. Since this requirement is trivially easy to satisfy at this location, we proceed to consider the fifth class. Since I is always equal to 1 at this location, the expression set term Left(S,I-1) simplifies to Left(S,0). This class therefore indicates that Left(S,0) should be distinguished from the empty string and from a variety of other terms involving Left. Since the operation Left(S,0) represents a special case returning the empty string, most of the terms in this class are redundant and could be eliminated if

the simplifier knew that Left(X,N) → "" whenever N≤0.[3] Giving this rule to the simplifier reduces this class to

{ Left(S,I-1) Left(S,I-Len(S)) Left(S,I-(Len(S)-I)) }.

The further inference that Left(S,1-Len(S)) must always return an empty string is too subtle for the current simplifier, but, once discovered by the tester, it can be entered as a further axiom. Thus the class is reduced to { "" Left(S,I-(Len(S)-I)) }. The latter string simplifies to Left(S,2-Len(S)), which can be non-empty only if Len(S)=1, so we know know that a test case consisting of a single character is required. This seems specific enough for us to propose a second test run, this time with input S="x". Running this test case and taking the two new axioms into account causes both equivalence classes at 1.1 to become trivial, and reduces the classes at 1.2 to:

{ Left(S,I-1)&Right(S,Len(S)-I) Right(S,Len(S)-I)
Right(Left(S,I-1)&Right(S,Len(S)-I),Len(S)-I) }

{ I<Len(S) I+1<Len(S) I<Len(S)-I I<Len(Mid(S,I,2)) I<Len(Right(S,Len(S)-I))
I<Len(Left(S,I-1)&Right(S,Len(S)-I)) }

{ Mid(S,I,2)=String_to_VString("□□") false Mid(S,I,1)=String_to_VString("□□")
Mid(S,Len(S),2)=String_to_VString("□□") Mid(S,I-1,2)=String_to_VString("□□")
Mid(S,Len(S)-I,2)=String_to_VString("□□") S=String_to_VString("□□")
Mid(Left(S,I-1),I,2)=String_to_VString("□□") Mid(S,I,Len(S))=String_to_VString("□□")
Mid(S,I,Len(S)-I)=String_to_VString("□□") Mid(S,I,2)=Left(S,I-1)
Mid(S,I,2)=Right(S,Len(S)-I) Mid(S,I,2)=Left(S,I-1)&Right(S,Len(S)-1)
Right(S,Len(S)-I)=String_to_VString("□□")
Left(S,I-1)&Right(S,Len(S)-I)=String_to_VString("□□") }

Examining the first class shows a need for two more simplification rules. The first is that ""&X → X for any string X; the second is that Right(Right(X,N),N) → Right(X,N) for any X and N. Moving on to the second class, we have I<Len(S), which simplifies to 1<Len(S) and which has so far been equivalent to I+1<Len(S) and to I<Len(S)-I, both of which simplify to 2<Len(S). These two can be distinguished by any test where S is of length 2. Rather than immediately choose a string of this form, we proceed with the examination of the classes to see if a more specific requirement can be found. The term I<Len(Mid(S,I,2)) is inherently equivalent to I<Len(S) when I=1. Proving this is beyond the capability of the current simplifier, so it is given as a new axiom. The term I<Len(S) will be distinguished from I<Len(Right(S,Len(S)-I) when Len(S)=2, so there is no new information to be gained from this class.

---

[3] Because of the inherent symmetry of the Left, Right, and Mid operations, this rule naturally suggests similar rules for Right(X,N) and Mid(X,J,N). We will assume henceforth that such rules are always supplied to simplifier together with any new rules specifically noted during the discussion.

Moving to the final class, we find that the simplified expression set term Mid(S,1,2)=String_to_VString("□□") has been false for both test cases used so far. This suggests that at least one test is required where S begins with a pair of blanks. Combining this with the earlier requirement of a test where Len(S)=2 suggests that the module be executed with S="□□". This test, and the new simplification rules, leaves a single class at location 1.2:

{ Mid(S,I,2)=String_to_VString("□□") Mid(S,Len(S)-I,2)=String_to_VString("□□")
S=String_to_VString("□□") Mid(S,I,Len(S))=String_to_VString("□□") }

The first term in this class can be distinguished from the second by any string that has a pair of blanks at one end but not the other and can be distinguished from the other terms only by a test where S begins with a pair of blanks but is of length greater than 2. Again, since these are fairly general conditions, we elect to examine other classes to get additional guidance. Since there are no more classes at location 1.2, we move on to location 2.1. Because this test location is reached just prior to each evaluation of the while-loop condition, this location has been reached with many more program states than has location 1.2. It is therefore not surprising that very few classes are left here. The classes remaining at location 2.1 are:

{ Left(S,I-1) Left(Left(S,I-1)&Right(S,Len(S)-I),I-1) }

{ Right(S,Len(S)-I) Right(Left(S,I-1)&Right(S,Len(S)-I),Len(S)-I) }

These classes can be disposed of by adding the new simplification rules Left(Left(X,N1)&Right(Y,N2),N1) → Left(X,N1) when N1≤Len(X) and Right(Left(X,N1)&Right(Y,N2),N2) → Right(Y,N2) when N2≤Len(Y), and by noting that 1≤I≤Len(S)+1 at every location but 1.1. This last rule is different from the ones employed so far, in that it is not an axiom on the data types manipulated by the program, but instead represents a statement about this module that cannot be determined via purely local information. A theorem prover capable of global analysis of the code would have little trouble proving that I≥1 and could also determine that I≤Len(S)+1 given a few axioms regarding the length of the strings returned by Left and Right. In the absence of such a powerful tool, the system should allow the tester to enter such rules. These rules are distinct from the data type axioms in two important aspects: 1) these rules may be tied to specific program locations, while the axioms apply throughout the program, and 2) these rules are specific to this program, while the axioms could be stored with the modules they describe, allowing them to be used during the testing of other programs that call those modules (We have assumed during this example that this is the first time the VString package has been used, and so no axioms are available from previous tests).

Location 3.1 is executed almost as often as is 2.1, and so it is not surprising that 3.1 has no non-trivial equivalence classes remaining. Interestingly, location 4.1 also has no non-trivial classes remaining, even though this location has so far been reached only twice during testing. Location 4.2, however, has the following classes:

{ Left(S,I-1) Left(S,Len(S)-(I+1)) }

{ Right(S,Len(S)-I) Right(S,Len(S)-2) }

{ Mid(S,I,2) Mid(S,I,I) Mid(S,Len(S)-I,2) }

{ I<Len(S) 1<Len(S) I+1<Len(S) I<Len(Left(S,I-1)&Right(S,Len(S)-I)) 2<Len(S) }

{ Mid(S,I,2)=String_to_VString("□□") false Mid(S,1,2)=String_to_VString("□□")
Mid(S,Len(S),2)=String_to_VString("□□") Mid(S,I-1,2)=String_to_VString("□□")
Mid(S,I+1,2)=String_to_VString("□□") S=String_to_VString("□□")
Mid(Left(S,I-1),I,2)=String_to_VString("□□")
Mid(Right(S,Len(S)-I),I,2)=String_to_VString("□□")
Mid(Left(S,I-1)&Right(S,Len(S)-I),I,2)=String_to_VString("□□")
Mid(S,I,1)=String_to_VString("□□") Mid(S,I,Len(S))=String_to_VString("□□")
Mid(S,I,I-1)=String_to_VString("□□") Mid(S,I,I+1)=String_to_VString("□□")
Mid(S,I,2)=Left(S,I-1) Mid(S,I,2)=Right(S,Len(S)-I) Mid(S,I,2)=Left(S,I-1)&Right(S,Len(S)-I)
Left(S,I-1)=String_to_VString("□□") Right(S,Len(S)-I)=String_to_VString("□□")
Left(S,I-1)&Right(S,Len(S)-I)=String_to_VString("□□") Mid(S,2,2)=String_to_VString("□□")
Mid(S,I,Len(S)-I)=String_to_VString("□□") }

Examining the first of these classes, the equality of Left(S,I-1) to the term Left(S,Len(S)-I-1) at this point in the program basically means that, each time this location has been reached, the Ith character has been the one just to the left of the center of S. Choosing a test case where the double blanks are well away from the center of the string should remedy this situation. The second class imposes a requirement that, on at least one test, there should be more than two characters following a pair of blanks. The third class requires a pair of double blanks not occurring at the second character, in a string with more than two characters, thus distinguishing Mid(S,I,2) from Mid(S,I,I), and repeats the requirement for an off-center pair of blanks to distinguish Mid(S,I,2) from Mid(S,Len(S)-I,2).

The fourth class shows that, so far, I<Len(S) exactly when 1<Len(S). These two terms can be distinguished only by reaching this location with I=Len(S)>1, which in turn means that S must end in a pair of blanks and have at least one additional character. If S has at least 2 additional characters, then 2<Len(S) will also be distinguished from I<Len(S). The term I+1<Len(S) can be distinguished from I<Len(S) only if this location is reached with I+1=Len(S), requiring S to have a pair of blanks followed by exactly one character. Combining these requirements implies that S must end in three blanks.

Combining all the requirements encountered since the last choice of test data suggests a new test of the form S="□□x□□□". This leaves the following class at 4.2:

{ Mid(S,I,2)=String_to_VString("□□") Mid(S,I,Len(S))=String_to_VString("□□")
Mid(S,I,I-1)=String_to_VString("□□") Mid(S,I,I+1)=String_to_VString("□□")
Mid(S,I,I)=String_to_VString("□□") }

This class can be eliminated by a test where a string of blanks of length 3 or more occurs at a position beginning other than with the first two characters in the string and being followed by more than two characters. Moving on to location 5.1, the only equivalence classes at this location are:

{ I<Len(S) 2<Len(S) }

{ Mid(S,I,2)=String_to_VString("□□") Mid(S,I,I)=String_to_VString("□□")
Mid(S,Len(S),2)=String_to_VString("□□") Mid(S,I-1,2)=String_to_VString("□□")
Mid(S,I,2)=S S=String_to_VString("□□") Mid(Left(S,I-1),I,2)=String_to_VString("□□")
Mid(S,I,1)=String_to_VString("□□") Mid(S,I,Len(S))=String_to_VString("□□")
Mid(S,I,I-1)=String_to_VString("□□") Mid(S,I,Len(S)-I)=String_to_VString("□□")
Mid(S,I,2)=Left(S,I-1) Mid(S,I,2)=Right(S,Len(S)-I) Mid(S,I,2)=Left(S,I-1)&Right(S,Len(S)-I)
Left(S,I-1)=String_to_VString("□□") Right(S,Len(S)-I)=String_to_VString("□□")
Left(S,I-1)&Right(S,Len(S)-I)=String_to_VString("□□") Mid(S,1,2)=String_to_VString("□□")
Mid(S,I,1+I)=String_to_VString("□□") }

In the first class, the term I<Len(S) must always be true at this location, so we must find data for which 2<Len(S) is false at this location. This can occur only if this location is reached with Len(S)=2, so we need a test with S of length 2 but not consisting of two blanks. This test condition is incompatible with the still outstanding requirement for a test with a string of at least 3 blanks, so we cannot combine the two. Running the test S="cc" eliminates the first class at location 5.1 but leaves the second class unchanged.

In the second class, the expression set term Mid(S,I,2)=String_to_VString("□□") must be false at this location, so it can be distinguished from the other terms only if those other terms can become true. A little examination shows that the rules Mid(X,J,N)≠Y when Len(Y)>N or Len(Y)>Len(X)-J or Mid(X,J,Len(Y))≠Y, Mid(X,J,N) → "" when J>Len(X), and Len(Left(X,N))≤N reduces this set to:

{ Mid(S,I,2)=String_to_VString("□□") Mid(S,I-1,2)=String_to_VString("□□")
Mid(S,1,2)=String_to_VString("□□") Left(S,I-1)=String_to_VString("□□")
S=String_to_VString("□□") Mid(S,I,2)=Left(S,I-1) Mid(S,I,2)=Right(S,Len(S)-I)
Mid(S,I,2)=Left(S,I-1)&Right(S,Len(S)-I) Right(S,Len(S)-I)=String_to_VString("□□")
Left(S,I-1)&Right(S,Len(S)-I)=String_to_VString("□□") }

The first term is the expression set term, always false at this location. The next four terms essentially ask whether the program has been correct on the first I characters. We certainly would hope that these three terms are inherently false, but proving it would require a complete proof of correctness for this program. The sixth term can be true for an S whose first and last two characters are identical and not double blanks. The seventh is true if the third and fourth characters of S are identical to the last two and are not both blanks. The eighth term is satisfied when S consists of any three identical non-blank characters, and the final two terms can be made true by any string of the form "X□□" where X is any non-blank character. Combining these requirements suggests test data "aaaa□□aaaa", "xxx" and "x□□".

With this test data, there are no remaining classes at locations 5.1, 5.2, or 6.1. The final location, 7.1, has the following classes:

{ I Len(S) }

{ Len(S)-I 0 }

{ Mid(S,I,2) Mid(S,I,I) Mid(S,I,1) Mid(S,I,Len(S)) Mid(S,I,1+I) }

{ Left(S,I-1) Left(S,I-1)&Right(S,Len(S)-I) }

{ Right(S,Len(S)-I) "" Right(S,Len(S)-(I+1)) Right(String_to_VString("□□"),Len(S)-I)
Right(S,Len(Mid(S,I,2))-I) Right(Mid(S,I,2),Len(S)-I) Right(S,Len(Left(S,I-1))-I)
Right(Left(S,I-1),Len(S)-I) Right(S,Len(Right(S,Len(S)-I))-I)
Right(S,Len(Left(S,I-1)&Right(S,Len(S)-I))-I) Right(S,I-1) }

{ I<Len(S) false I<Len(Mid(S,I,2)) I<Len(Right(S,Len(S)-I))
I<Len(Left(S,I-1)&Right(S,Len(S)-I)) }

{ Mid(S,I,2)=String_to_VString("□□") false Mid(S,1,2)=String_to_VString("□□")
Mid(S,2,2)=String_to_VString("□□") Mid(S,I-1,2)=String_to_VString("□□")
Mid(S,Len(S)-I,2)=String_to_VString("□□") S=String_to_VString("□□")
Mid(Right(S,Len(S)-I),I,2)=String_to_VString("□□")
Mid(Left(S,I-1)&Right(S,Len(S)-I),I,2)=String_to_VString("□□")
Mid(S,I,Len(S)-I)=String_to_VString("□□") Mid(S,I,2)=Right(S,Len(S)-I)
Left(S,I-1)=String_to_VString("□□") Right(S,Len(S)-I)=String_to_VString("□□")
Left(S,I-1)&Right(S,Len(S)-I)=String_to_VString("□□") Mid(S,I,I-1)=String_to_VString("□□") }

An examination of the first two classes and a quick scan of the others show that a number of terms are listed only because of the need to distinguish I from Len(S). At this point in the program, I=Len(S) unless S is the empty string. This immediately suggests the empty string as a new test case, after which we can henceforth assume that I=Len(S). The third class can also be simplified by introducing the rule that Mid(X,J,N) → Mid(X,J,Len(X)-J+1) when N>Len(X)-J+1. The classes at 7.1 are then reduced to:

{ Right(S,Len(S)-I) Right(S,Len(Mid(S,I,2))-I) Right(Left(S,I-1),Len(S)-I) }

{ I<Len(S) I<Len(Mid(S,I,2)) I<Len(Left(S,I-1)&Right(S,Len(S)-I)) }

{ Mid(S,I,2)=String_to_VString("□□") Mid(S,1,2)=String_to_VString("□□")
Mid(S,2,2)=String_to_VString("□□") Mid(S,I-1,2)=String_to_VString("□□")
Mid(S,Len(S)-I,2)=String_to_VString("□□") S=String_to_VString("□□")
Left(S,I-1)&Right(S,Len(S)-I)=String_to_VString("□□") }

None of these classes contain terms that can be distinguished, so the completed test set is "a□□bc", "x", "□□", "□□x□□□", "aaaa□□□aaaa", "xxx", "x□□", and "". This test set includes tests that bypass the module's loop, that execute each internal branch within the loop exactly once and then exit the loop, and that require multiple executions of the loop both alternating and not alternating the internal branches taken on successive iterations of the loop.

In some ways, the above example has exaggerated the difficulty of using EQUATE. We have assumed that no prior axioms/simplification rules were known for the functions and procedures called by the module under test, and that each such rule would be proposed only when examination of the classes revealed a need for that rule. Many of these rules, if not already available, could have been proposed at the start of testing based upon our knowledge of the routines involved. Similarly, in the example we added new test data only after rigorous examination of the classes left by earlier tests, proceeding one test location at a time. In section II, however, we were able to describe much of the required test data based upon a cursory examination of the expression set terms. Furthermore, we

noted during that discussion that much of the test data required by EQUATE corresponds to good intuitive rules for testing loops and other program constructs. This suggests that a more profitable approach to EQUATE testing would be to generate a number of test cases based upon a high-level examination of the expression set and upon less rigorous testing guidelines. Running these test cases should considerably reduce both the number and the size of the remaining classes, making the subsequent detailed examination of those classes much easier. EQUATE can then be used to indicate the remaining gaps in the test coverage.

IV. Relation to Other Testing Strategies

This section discusses the relation of EQUATE to a variety of earlier testing strategies. The strategies considered are mutation testing [1,2,4,5,12], perturbation testing [19,20,21], Simpler Expression Coverage [9], and the testing subsystem of DAISTS [7]. These strategies are compared to EQUATE in terms of their support for abstraction, their ability to detect simple faults, and their ability to detect combinations of simple faults, in accordance with the goals stated in the introduction. Before discussing these other strategies, we begin by reviewing EQUATE's performance in these areas.

EQUATE's support for abstraction stems from its emphasis on the operations applied to each object. EQUATE makes no distinction between those objects whose data types are user-defined and those whose types are language-supplied primitives. As a result, it can be expected to perform equally well with modules at any level of data and functional abstraction.

EQUATE's ability to detect simple faults comes in part from the fact that EQUATE requires the subexpressions appearing in each statement to be distinguished from a wide range of alternative expressions, thus eliminating the possibility that the given subexpression should have been replaced with any of those alternatives.

Of course, at any given test location EQUATE not only requires the expressions appearing at that location to be distinguished from those alternatives, but requires many of those alternatives to be distinguished from one another. In fact, the set of terms to be distinguished is independent of the test location and hence independent of the statements, expressions, and structures appearing near that test location. This independence of the local testing criteria from the local syntactic structure is crucial to the detection of combinations of simple faults (and thus to the detection of complex faults), although this relationship may not be immediately apparent.

To demonstrate the importance of independence from local syntactic structure, consider the following thought experiment, which is centered on the design of a (wholly impractical) hypothetical testing strategy. The key idea will be to directly generate, on the first execution of any statement containing one or more expressions, all those expressions that would yield the same results when evaluated in that program state as would those already appearing in the module. For subsequent executions of the same statement, we can keep track of how many of these alternatives continue to mimic the original expressions either by evaluating each remaining alternative and comparing its value to the original or by repeating the generation of equivalent alternatives and taking the intersection of the

different sets of alternatives.

To generate the alternative expressions, we will employ repeated mutations according to one of two rules.

1. Any expression or subexpression E_1 can be replaced by E_2 if it is axiomatically true for the data type returned by E_1 and E_2 that $E_1 = E_2$.

2. Any variable X in an expression can be replaced by a constant denoting its current value, and any constant can be replaced by a variable whose current value is denoted by that constant.

We allow up to N substitutions using these two rules. Clearly, as N approaches infinity, this strategy generates all those expressions that, when evaluated in the current program state, would be equivalent to the original expression.[4] Using integer expressions as an example, an expression X+1 encountered at a point in the execution where the program state were described by {X=2, Y=0, Z=2} might go through a series of mutations X+1 → X+1+0 → X+1+0*Z → X+1+Y*Z → 2+1+Y*Z → 3+Y*Z or X+1 → 2+1 → Z+1 → 1*(Z+1) → (0+1)*(Z+1) → (Y+1)*(Z+1) or simply X+1 → 2+1 → 3. The generated expressions may be either more or less complex than the original, may appear very similar to the original or may look to be completely unrelated; the only constant is that they will have the same value in the current program state as does the original expression.

Consider the properties of this strategy as N increases. We will show that the sets of alternative expressions generated for different expressions, for different program states, and for different locations in the module using at most N substitutions all become increasingly identical. Consider first the set of integer expressions. Let f and g be any two expressions. Then for any expression E_1 appearing in the module, there exist an infinite number of expressions E_2 (e.g. $E_2 = E_1+(f-g)$) such that $E_2 \neq E_1$ only in states where $f \neq g$. Given any two expressions f and g that have the same value in the current program state, as N increases, the probability that at least one such E_2 will be generated also increases.

The expression E_2 can be distinguished from the original expression E_1 only if f can be distinguished from g. Since f and g are arbitrary expressions, as N becomes arbitrarily large the set of alternatives generated by this hypothetical testing strategy can be distinguished from the original expression if and only if every pair of expressions are distinguished from each other. The conclusion, then, is that as N increases, the actual testing requirements become increasingly independent of the form of the original statement.

Similar behavior will occur, not only with integer expressions, but with expressions returning any data type for which at least one operation exists, or can be constructed, that has both an identity element and an inverse. It is likely to be approximately true for many other data types. In particular, mixed mode operations on two or more types (e.g. an operation PUSH mixing inputs of types STACK and ELEMENT) tend to guarantee that, if

[4] In effect, we have defined a new version of mutation testing where the mutation rules are drawn from the data type axioms and where arbitrary combinations of the mutations may occur in any expression.

this property holds for one type, then it will hold for the other.

The point of this exercise is not to seriously propose the above testing strategy (although it clearly has a strong relation to EQUATE), but to establish that the goal of detecting combinations of primitive faults in any statement is tied to the idea that the testing criteria at any location should be independent of the local syntactic structure at that location. It is for this reason that EQUATE employs the same set of terms at every test location.

What types of faults are best detected by criteria that are independent of the local syntax? The above arguments would suggest that such independence is particularly useful for detecting faults that improperly simplified the module, omitting parts of calculations that may be important only in certain program states. One example would be the omission of an entire assignment statement. A necessary condition for detection the omission of a statement $X := f(Y)$ is that the expressions X and $f(Y)$ take on different values at the location where the statement should have appeared. Local-syntax-dependent strategies such as [1,9,12] would usually enforce such a condition only if X or $f(Y)$ were used in one of the statements immediately surrounding that location. EQUATE, however, would enforce this condition regardless of the appearance of the surrounding statements, provided that it recognized X and $f(Y)$ as useful terms.

The basic design of EQUATE stems from an effort by the author to combine the strengths of two existing testing strategies, perturbation testing, and mutation testing. These two strategies, though they have very different origins and motivations, share a common structure, which in turn is partially reflected in EQUATE. Both strategies choose a set of test locations and postulate a set of possible faults at each of those locations. Each time that a test location is reached during testing, these strategies determine the subset of those possible faults which, if they were present, would still not affect the module output. The intersection of all the subsets obtained in this manner at a given test location forms a set of faults which would have escaped detection for all the tests done so far. By providing a listing of these as-yet-untested faults, both strategies provide guidance for the selection of new test data, encouraging the choice of tests targeted at those specific faults.

Mutation Testing

Mutation testing places a test location at every statement. The postulated set of faults consists of single applications of a variety of primitive substitutions called *mutations*. The mutations are chosen to model faults that are believed to be common in the programming language being employed. As each test location is reached, the program state induced by the statement actually appearing at that location is compared to the state which would result from the mutated versions of that statement. Those mutated statements which result in the same state as the original correspond to faults which would escape detection on that particular execution.[5] The intersection of this sets of faults with those from other tests is computed implicitly by only evaluating those mutated statements which have escaped all

[5] The form of mutation testing described here is known as *weak mutation testing* [12] or *weak statement mutation testing* [2], to distinguish it from the much more expensive *strong mutation testing*, which requires each mutation to cause a change in the module's eventual output rather than just a change in the current program state.

previous tests.

For example, if mutation testing were performed on the program fragment shown in figure 4, the mutated versions of the second statement might include:

A:=2.*A+B; B:=2.*A+B; C:=2.*A+B; X:=1.9*A+B; X:=2.1*A+B; X:=2.+A+B; X:=2.-A+B; X:=2/A+B; X:=2.**A+B; X:=2.*B+B; X:=2.*C+B; X:=2.*X+B; X:=2.+B; X:=A+B; X:=2.*abs(A)+B; X:=2.*-abs(A)+B; X:=2.*A-B; X:=2.*A*B; X:=2.*A/B; X:=(2.*A)**B; X:=2.*A; X:=B; X:=2.*A+A; X:=2.*A+C; X:=2.*A+X; X:=2.*A+abs(B); X:=2.*A+-abs(B).

If the module were tested with inputs (1.,0.,0.), the set would be reduced to:

X:=2/A+B; X:=2.**A+B; X:=2.+B; X:=2.*abs(A)+B; X:=2.*A-B; X:=2.*A; X:=2.*A+C; X:=2.*A+abs(B); X:=2.*A+-abs(B).

On a further test with inputs (0.,1.,1.), these remaining versions (but not the ones already eliminated) would be evaluated, leaving:

X:=2.*abs(A)+B; X:=2.*A+C; X:=2.*A+abs(B).

A major advantage of mutation testing is its wide applicability. Because the mutations are defined for the given language in terms of the syntax and primitive objects of that language, mutation testing can be applied to essentially any module in that language. This advantage, however, is somewhat mitigated by the fact that defining the mutations in terms of the language primitives results in many mutations being useful only for routines and data structures at a relatively low level of abstraction. Common mutations such as "if the variable X occurs in some statement, then replace X by X+1" are of limited use because "+" is only defined for certain data types, and "+1" is defined for an even smaller set of data types. Modifying the rule to read "if X is an integer variable..." misses the point, since it is of no help in devising a more general testing method. The end result is that mutation testing is far more rigorous for statements manipulating language-defined data types than for statements manipulating user-defined data types, a sign of weak support for abstraction compared to EQUATE.

For arithmetic expressions over integers or real numbers, it is difficult to compare the simple faults detected by mutation testing to those detected by EQUATE. Mutation testing includes certain rules specifically aimed at such expressions (e.g. replace an existing integer expression *E* by *E*+1). Since EQUATE would not duplicate such rules (unless similar expressions, in this case *E*+1 or other +1 operations already appeared in the code), mutation testing can require certain tests that EQUATE would not. On the other hand, since EQUATE uses operand substitution terms that can be more complex than any of the

```
input A, B, C;
X := 2. * A + B;
.
.
.
```

Figure 4: To Be Tested With (1.,0.,0.) and (0.,0.,1.).

mutation testing primitives, EQUATE may require certain tests not required by mutation testing.

If, however, we consider expressions involving user-defined data types rather than the language's primitive data types, comparisons are much simpler. Then the weak mutation rules described in [2,12], as well as the strong mutation rules from [1,4,5] that would be applicable to weak mutation testing, collapse to just a few rules. Mutations would be generated to 1) substitute each variable name for every existing variable reference, 2) to substitute constants for each variable reference, 3) to substitute different variable names for the variable on the left-hand side of an assignment statement, and 4) to delete the statement entirely. EQUATE subsumes all four of these mutation rules. Consider a statement that computes the value of an expression $f(X)$ and either writes it to some output file or stores it at Y. The first mutation rule would alter the expression by substituting other variables for X. Thus $f(X)$ would have to take values different from, for example, $f(Z)$ at least once. In this situation, X, Z, and $f(X)$ would be in the expression set and therefore $f(Z)$ would be an operand substitution term that EQUATE would require to be distinguished from $f(X)$. Similarly, EQUATE would satisfy the second rule by requiring X to take on at least two distinct values, and in fact would go even further by requiring $f(X)$ to take on at least two distinct values. The third mutation rule would replace assignments of the form $Y := f(X)$ by assignments such as $Z := f(X)$. This mutation fails to change the program state only if Y, Z, and $f(X)$ are all equal. Since all three will be in the expression set, EQUATE will require them to take on distinct values. The final mutation rule can only be satisfied if the values $f(X)$ computed by the statement is stored in some object Y that already had that value, a situation avoided by EQUATE's requirement that Y and $f(X)$ be distinguished. Thus, for non-primitive data types, EQUATE appears to subsume weak mutation testing, thus confirming the claim that EQUATE offers better support for abstraction and showing as well that EQUATE detects a wider range of simple faults in abstract programs.

In terms of detecting combinations of faults, there is a tremendous contrast between the approaches taken by mutation testing and by EQUATE. Mutation testing exhibits a strong dependence upon local syntactic structure, explicitly assuming that the possibility of combinations of faults can be ignored. It seems likely then, that EQUATE should offer significantly better performance in detecting combinations of faults.

Perturbation Testing

In perturbation testing, the test locations occur at each statement containing an arithmetic expression. The postulated set of possible faults consists of a set of error expressions which might be added to the existing expression. Usually a sufficiently rich set is chosen to allow for the possibility that an error expression could completely subtract away the existing expression and add in something of equivalent complexity. Whatever set is actually chosen, perturbation testing actually considers all faults which could be formed as linear combinations of that set, and hence considers all possible combinations of such faults within a given statement. Each time that the test location is reached, it is possible to compute, from the program state at that time, the set of error expressions that would evaluate to zero in that state and that would therefore escape detection. (Even though there are usually an infinite number of these untested faults, a finite description can be provided.) The intersection of this set with the set of faults previously left untested is then computed explicitly. Both of these steps are accomplished via the solution of a single system

of linear equations.

For example, at the second statement of the module in figure 4, the simplest form of perturbation testing would consider alternate versions of this statement that had the form $X := aA + bB + cC + d$, a form which includes most of the alternatives considered by mutation testing. When the test data (1.,0.,0.) is used, perturbation testing would then conclude that the correct form of the second statement might actually be

$$X := 2.*A + B + \alpha(A-1.) + \beta(B) + \gamma(C).$$

for any values of α, β, and γ without the difference being detected by that test. The tester would then be advised to seek test data so that A−1., B, and/or C were non-zero. On running the further test (0.,1.,1.), perturbation testing would report that the correct form of the statement might still be

$$X := 2.*A + B + \alpha(A+B-1.) + \beta(B-C).$$

If the addition of (A−1.) or (B) or (C) to the given expression are considered to be the simple faults treated by perturbation testing, then the addition of $\alpha(A-1.)+\beta(B)+\gamma(C)$ or $\alpha(A+B-1.)+\beta(B-C)$ must be considered to be combinations of those simple faults. In fact, these expressions stand for an infinite number of possible combinations, all of which are explicitly monitored by perturbation testing.

Perturbation testing offers essentially no support for abstraction. The nature of the calculations constituting the perturbation analysis prevents its use with data other than real numbers and integers. Its ability to detect faults in those statements to which it is applicable is, however, very strong. For expressions over integers and floating point numbers, in a program employing only the normal arithmetic operators, any alternative expression generated by EQUATE would also be tested by perturbation testing. Perturbation testing is therefore more rigorous in this situation. If, however, the expressions include calls to user-written functions or to operators and functions other than the normal arithmetic operators, then it is likely that EQUATE will generate operand substitution terms that would not be tested by perturbation testing.

The most important strength of perturbation testing is its ability to deal with combinations of faults, explicitly tracking all possible linear combinations of its primitive faults in any given statement. As a consequence of this part of the analysis, the local testing criteria perturbation testing imposes are essentially identical, independent of the testing location. This aspect of perturbation testing was the inspiration for the corresponding property of EQUATE. Thus EQUATE cannot compete with perturbation testing in modules employing only integers and floating point numbers and only arithmetic operators, but for all other programs EQUATE's wider applicability and support for abstraction give it the edge.

Simpler Expression Coverage

Hamlet proposes a strategy in [9] that selects a set of terms superficially similar to EQUATE's. His strategy requires each expression in the module to be distinguished from all possible simpler expressions (involving those variables and operators appearing anywhere in the code) at the location where the original expression occurs. Thus, at the second

statement in figure 4, he would require 2.*A to have a different value from 2., A, B, and C, and 2.*A+B to have a different value from 2., A, B, C, 2.+2., 2.+A, 2.+B, 2.+C, A+2., A+A, A+B, A+C, B+2., B+A, B+B, B+C, C+2., C+A, C+B, C+C, 2.*2., 2.*A, 2.*B, 2.*C, A*2., A*A, A*B, A*C, B*2., B*A, B*B, B*C, C*2., C*A, C*B, C*C, and possibly other terms if there are any other variables and operators used in the rest of the program. On executing the test data (1.,0.,0.) then, Hamlet's strategy would report that 2.*A had yet to be distinguished from 2. and that 2.*A+B had yet to be distinguished from 2., 2.+B, 2.+C, A+A, B+2., C+2., 2.*A, and A*2. On executing the second test (0.,1.,1.) he would report that the second statement had been completely tested (assuming that there were no other terms generated because of variables and operators elsewhere in the program).

The testing of modules containing abstract, user-defined data types was not a concern addressed by Hamlet. Because, however, his strategy uses the operators appearing in the code to generate terms, it does adjust itself to such modules automatically. Many of its terms, however, may be poor choices. Simpler expression coverage generates every possible expression (up to some level of complexity) that can be formed using every operator and variable appearing in the program. If the language does not provide strict encapsulation facilities, or if the programmer chooses not to use them, many of the resulting expressions may involve the use of inappropriate operators with some objects. The EQUATE approach would seem to be more economical.

A comparison between the simple faults detected by simpler expression coverage and by EQUATE is somewhat complicated. In simpler expression coverage, any expression E is compared to terms including subexpressions of E, subexpressions of E with some operand substitutions, and other expressions bearing no relation to E except the fact that they are simpler. There is substantial overlap between this set of terms and the set that EQUATE would generate. EQUATE would also require E to be distinguished from all subexpressions of E, and EQUATE's operand substitution terms would have a substantial overlap with those of Hamlet. EQUATE would not check E against the many non-expression set terms simpler than but unrelated to E and not actually appearing in the source code. It is not clear how valuable such terms really are during testing. EQUATE would, however, compare E to many terms that were not simpler than E. Hamlet's strategy is most suited to faults that make the expressions more complicated, so that the possibly correct alternatives to be considered during testing are simpler than the expression actually appearing in the code. Such a bias may not be reasonable unless there is reason to believe that the majority of faults tend to be faults of over-complication. There is some evidence that the opposite is more often true for those faults that tend to missed during testing [8].

Clearly Hamlet's strategy will be effective at detecting complex or combined faults that result in simpler expressions without introducing any new variables, constants, of operators. Is this sufficient to guarantee detection of other combined faults, especially faults that result from over-simplification of the code? Since simpler expression coverage only requires an expression to be distinguished from the simpler alternatives at the location where that expression occurs, this strategy exhibits a moderate level of dependence on local syntactic structure (though considerably less than mutation testing). The arguments presented earlier would suggest that this represents a weakness, relative to EQUATE.

DAISTS

The DAISTS system of Gannon et al. implements a methodology for the specification, implementation, and testing of abstract data types. The specification language emphasizes axiomatic definition of the abstract type. The testing criterion monitored by DAISTS is that each subexpression appearing in the code of the specification must take on at least two distinct values. Thus if the expression 2. * A + B appeared in some axiom of a specification and if the implementation were tested with data that forced the quantities A and B in that axiom to take on values 1. and 0., respectively, DAISTS would report that A, B, 2.*A, and 2.*A+B had so far taken on only a single value each (1., 0., 2., and 2., respectively). If another test were performed on which the quantities A and B took on values 0. and 1., respectively, then DAISTS would be satisfied since each of the four expressions would have taken on two distinct values.

Of the four testing strategies being discussed here, DAISTS exhibits the strongest concern for abstraction. EQUATE is, however, more flexible since DAISTS is tied to the concept of axiomatic specification. If that mode of specification is not employed, DAISTS is not applicable. Note that, while axioms are of use to EQUATE in detecting inherently equivalent terms, DAISTS emphasizes axioms governing the module(s) being tested while EQUATE uses mainly axioms governing the modules called by the one being tested. Finally, while DAISTS is designed for use with fully encapsulated implementations of abstract data types, EQUATE can be employed with modules employing primitive, unencapsulated user-defined types, or encapsulated abstract data types.

DAISTS' testing criterion is the simplest one considered in this paper. Its requirement that each expression and subexpression appearing in the specification of an abstract data type should be forced to take on at least two different values has a direct analogue in EQUATE where the same is required of each expression and subexpression in the module. The two requirements are not identical, since one is imposed on the specification and the other on the code. Since the basic testing problem, that of increasing confidence in the equivalence of the specification and the code, involves a symmetric relation between the specification and the code, there is not likely to be any reason for either purely code-based or purely specification-based criteria to be more effective than the other. (In fact a combination of the two is probably called for).

Of course, the requirement that each object take on more than just one value during testing is only a fraction of the total testing requirement imposed by EQUATE, so it seems likely that EQUATE will detect a much wider variety of both simple and combined faults than would DAISTS.

V. Directions for Future Work

It is imperative that testing strategies be developed that can operate on programs containing a variety of abstract, user-defined data types. This paper describes an approach to this problem, the EQUATE testing strategy. This strategy attempts to offer support for data and functional abstraction, to detect a wide variety of simple faults, and to provide good coverage of combinations of those faults.

The EQUATE testing strategy is at an early stage in its development and there are still a number of basic issues to be addressed. A particularly important subject for future study is the way in which EQUATE compares to other testing strategies. While some informal discussion on this subject has been presented here, a more rigorous study involving more testing strategies is merited.

EQUATE presents a schema for testing that may permit construction of a whole family of useful testing methods because of the variety of possibilities for sets of terms. The expression set seems to be an obvious choice as a source of terms, but after that a number of alternatives open up. Operand substitution terms have been shown in this paper to be particularly important to the goal of support of abstraction. An option to be explored further is the use of operator substitution terms, where the substitutions are based upon language features for encapsulation of abstract data types. Other sources of new terms might include specifications associated with the module being tested and those modules that it calls, programmer-supplied assertions appearing in the module, terms from other modules in the same package as the module being tested, or even terms from the modules called by the one under test.

More drastic alterations of EQUATE that might be of interest would include examining the equivalence of larger portions of the abstract syntax tree than simply expressions and subexpressions. Statements or groups of statements could be subjected to a similar analysis. It is interesting to compare this idea to what would happen if EQUATE in its present form were used with functional or applicative languages where an entire module may be considered to be a single expression.

At present, a prototype implementation of EQUATE exists that is capable of generating the relevant classes at a single test location for a limited set of data types and of detecting the majority of inherently equivalent terms in those classes. A more complete implementation is anticipated in the near future.

References

1. 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

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

3. 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

4. R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, v. 11, no. 4, April 1978, pp. 34-41

5. R. A. DeMillo, F. G. Sayward, and R. J. Lipton, "Program Mutation: A New Approach to Program Testing," *State of the Art Report on Program Testing*, 1979, Infotech International

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

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

8. R. L. Glass, "Persistent Software Errors," *IEEE Transactions on Software Engineering*, vol. SE-7, no. 2, 162-168, March 1981

9. 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

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

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

12. 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

13. 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

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

15. 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

16. 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

17. S. J. Zeil and L. J. White, "Sufficient Test Sets for Path Analysis Testing Strategies", *Proceedings of the 5th International Conference on Software Engineering*, IEEE Computer Society, pp. 184-191, 1981

18. S. J. Zeil, *Selecting Sufficient Sets of Test Paths for Program Testing*, Ph.D. dissertation, 1981, Ohio State University, also technical report OSU-CISRC-TR-81-10

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

20. 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

21. S. J. Zeil, *Perturbation Testing for Domain Errors*, COINS Technical Report 83-38, University of Massachusetts, December 1983, revised February 1984