A Theory of Fault-Based Testing

LARRY J. MORELL

Abstract—A theory of fault-based program testing is defined and explained. Testing is *fault-based* when it seeks to demonstrate that prescribed faults are not in a program. It is assumed here that a program can only be incorrect in a limited fashion specified by associating *alternate expressions* with program expressions. Classes of alternate expressions can be infinite. Substitution of an alternate expression for a program expression yields an alternate program that is potentially correct. The goal of fault-based testing is to produce a test set that differentiates the program from each of its alternates.

A particular form of fault-based testing based on symbolic execution is presented. In *symbolic testing* program expressions are replaced by symbolic alternatives that represent classes of alternate expressions. The output from the system is an expression in terms of the input and the symbolic alternative. Equating this with the output from the original program yields a *propagation equation* whose solutions determine those alternatives which are not differentiated by this test. Since an alternative set can be infinite, it is possible that no finite test differentiates the program from all its alternates. Circumstances are described as to when this is decidable.

Index Terms—Error-based testing, fault-based testing, fault propagation, program testing, program verification, symbolic execution.

I. Introduction

TESTING is a method of program verification that deduces from execution information that a program possesses required properties. The role of testing theory is to study this deductive process and to investigate the conditions under which it can be accomplished. Testing may be viewed in two stages. First, a *testing strategy* is developed which describes how testing is to be performed; i.e., how the input data is to be selected, what information is to be collected, and how to collect and analyze this information. Second, the strategy is applied to a given program, resulting in a *test* of that program.

A testing theory must provide a means of judging the quality of either stage of testing. Intuitively, a good test provides convincing evidence that a program is correct. A testing strategy could therefore be considered good if it produces good tests when applied to a particular program. This approach has been taken by Howden [1] who defines a *reliable* test to be one whose success implies program correctness. The disappointing result, however, is that a reliable test is not attainable in general. It is thus necessary to try a different approach to defining the quality of a test or a testing strategy.

Manuscript received September 10, 1987; revised March 26, 1990. This work was supported in part by the Air Force Office of Scientific Research under Contract F49620-83-K-0018.

The author is with the Department of Computer Science, Hampton University, Hampton, VA 23668.

IEEE Log Number 9036264.

The intent of a quality measure is to determine when one test (or testing strategy) is better than another. It is therefore desirable to have gradations of goodness, with a reliable test being the ultimate. Structural coverage measures such as statement coverage, branch coverage, path coverage, or TER coverage [2], [3] illustrate such a gradation, but even when these criteria are satisfied they do not imply correctness. Howden showed, for example, that executing all statements does not guarantee that a program is correct [1].

The means taken here of assessing the quality of tests is to compare them on the basis of how many faults each demonstrates are not in the program, i.e., on how many faults each test *eliminates*. This choice satisfies the gradation mentioned above since the highest quality test by this approach is one which eliminates all faults, implying the program is correct. Similarly the quality of a strategy is determined solely by what faults it is guaranteed to eliminate.

Testing is *fault-based* when its motivation is to demonstrate the absence of prespecified faults. This paper develops a fault-based testing meta-theory that

1) defines fault-based testing,

2) provides strategies by which fault-based testing can be applied, and

3) proves properties about the strategies concerning what faults can and cannot be eliminated.

The theory defines the mechanism whereby the faults can be defined, and once defined, how they are eliminated. It is a meta-theory in the sense that it applies once the fault categories have been chosen, and is therefore independent of any particular class of faults.

The difference between this approach and that of conventional testing can be understood by reflecting on the "information content" of a test. In the traditional point of view a test that fails to reveal an error bears no useful information. For example, Myers [4], states that "since a test case that does not find an error is largely a waste of time and money, the descriptor 'successful' seems inappropriate" [4]. Thus, the information content of a test is measured by the failures it causes.

A different perspective is taken here. It is argued that every correct program execution contains information that proves the program could not have contained particular faults. The fault-based theory developed here treats successful tests as indicative of the absence of prespecified faults. It thus measures the information content in a successful execution by the set of eliminated faults. This work therefore focuses on methods for proving the absence of prespecified faults from traces of correct executions.

It should be noted that "proving the absence of prespecified errors (faults)" is not to be interpreted as "look for arbitrary errors (faults)." The "look for arbitrary errors" viewpoint is both imprecise and inadequate [5]. It is imprecise because every testing method can in some sense be interpreted as looking for errors. It is inadequate since it provides no guidance as to when to stop testing. If no errors are found, how long does one keep looking? If several errors are found, does one keep looking? On the other hand, proving the absence of specified faults is both precise and adequate (in the sense of [5]). Since the faults are prespecified it is clear what is and what is not being accomplished. Having proven the absence of these faults is the clear stopping criterion. Furthermore, if a prespecified fault has not been eliminated, areas of code are therefore identified for further investigation.

The information content for a given test (in the fault-based sense) is fixed, but the amount of that information perceived varies with the amount of analysis performed. As a trivial example, a single successful test indicates that no executed loop is missing a loop control variable reassignment. It also demonstrates that the program does not always assign the wrong name to the output file. Probing further, it may be possible to deduce that no executed loop contains an off-by-one fault by analyzing intermediate values computed along the path. Deeper analysis thus leads to better discernment of the information content of the test, namely, the types of faults eliminated.

The theory developed here for fault-based testing is adapted from mutation testing [6]. Expressions are supposed incorrect in a limited manner as defined by a set of *alternatives*, which contains legal replacements for the expression. The mutation model is extended here in a significant way: alternative sets are allowed to be infinite. This entails developing a fault-based testing theory that does not rely on the ability to generate and execute all alternatives, as is done in mutation testing. Fault-based testing therefore must prove the absence of infinitely many faults based on finitely many executions. Theorems asserting the absence of prescribed faults must be generated and proven. *Symbolic testing*, a fault-based testing method for generating these theorems, is developed and discussed. It has the advantage of being introspective in that symbolic testing can be used to investigate properties of fault-based testing schemes, including itself.

Fault-based testing therefore intertwines elements from both formal verification and traditional testing. This approach varies considerably from how testing and formal verification have been related in the past. The tendency has been to place testing and proof-of-correctness as separate activities, as antagonists in the game of demonstrating program correctness. For example, Gerhart and Yelowitz [7] recommend testing to check the veracity of a proof. Goodenough and Gerhart [8] propose that testing can establish the basis step in an inductive proof. Geller [9] suggests that testing can be used to prove assertions which would otherwise be tedious to prove by hand. However, there has been no concerted attempt to combine testing and verification into a unified theory.

The fault-based testing theory developed here integrates testing with program verification by exploiting the advantages of both. Properties are proved about the program or its alternative sets which in conjunction with program execution imply the absence of certain faults.

Section II gives a basic theory of testing and demonstrates the need for fault-based testing, which is defined and discussed in Section III. Symbolic testing, a particular style of fault-based testing, is described in Section IV and is used to discuss the theoretical limitations of fault-based testing. Particular emphasis is given to the decidability of when an infinite alternative set can be eliminated by a finite test. Section V relates the present work to the literature. Section VI summarizes and suggests future research areas.

II. A General Theory of Testing

In this section a general theory of testing is presented that is both descriptive and prescriptive. The fundamental components of testing theory are precisely defined and the relevant notation is introduced. Unsolvable problems are addressed and for the areas in which general testing is limited, direction is given for a more specialized theory.

A. Basic Components and Relationships

During the process of software creation many types of mistakes can occur. To distinguish among these the IEEE conventions [10] are adopted here. An *error* is a mental mistake by a programmer or designer. It may result in textual problem with the code called a *fault*. A *failure* occurs when a program computes an incorrect output for an input in the domain of the specification. For example, a programmer may make an error in analyzing an algorithm. This may result in writing a code fault, say a reference to an incorrect variable. When the program is executed it may fail for some inputs.

The fundamental components in this testing theory are programs, specifications, and test sets. Every program P defines a function called its *program function*, which consists of the set of input–output pairs computed by P. If P is a program, then its program function is denoted by $[P]$, as suggested by Mills [11]. The *domain* of the program function, written dom $([P])$, is all points for which P halts. If $(x, y) \in [P]$ then $[P]\ (x) = y$ and $x \in$ dom $([P])$. $[P]\ (x)\downarrow$ indicates that the program function is defined for x; i.e., P on input x halts and outputs $[P]\ (x)$. $[P]\ (x)\uparrow$ indicates that the program function is not defined for x; i.e., P on input x does not halt.

A *specification* S is a string that represents a recursive relation with a recursive domain. The relation is called the *specified relation* and is denoted by $[S]$. Requiring the domain of the specified relation to be recursive avoids the awkward circumstance of not knowing whether a program is supposed to halt on a given input. Requiring the specified relation to be recursive enables the specification to act as an *oracle*, a decision procedure which can decide whether or not any proposed input–output pair is in the specified relation. Both properties are necessary since neither one implies the other [12].

The central concept relating programs to specifications is that of correctness. A program P is *correct* with respect to a specification S if and only if dom $([P] \cap [S]) =$ dom $([S])$ [13]. Thus, a program is correct with respect to a specification if and only if it computes results for the entire domain of interest, and all the results it computes are as specified.

It is convenient to be able to discuss a given program P, a specification S, and a domain of interest D as a whole. Such a triad $< P, S, D >$ is called an *arena*. By default the third component is the entire domain of $[S]$. It is convenient to have the domain explicitly named, especially for focusing attention on the operation of the program over a portion of the input space. D is our source of test data; thus any finite subset of D is called a *test set*. A test set is called *successful* if and only if for each $x \in T$, $(x, [P](x)) \in [S]$.

Much attention has been given to demonstrating the correctness of a program with respect to a specification. There are two general approaches to this problem, proof-of-correctness and testing. In proof-of-correctness, a formal proof is given to show that the program computes what is specified. Different specification methods induce different proof techniques, but in each the proof is inductive and based on the program text rather than on program execution.

Since testing is based on program execution and has little inductive power it is difficult to integrate testing with proof-of-correctness. Earlier attempts have placed testing in a subservient role, e.g., as proving the basis step of an inductive proof [8] or as a means of simplifying complex proofs [9]. It is possible, however, to view verification from a noninductive perspective. As shown below, such an approach is theoretically weaker than the inductive approach, but has advantages not supported by the inductive approach.

B. An Alternate Approach to Correctness

An alternate approach to demonstrating program correctness is one of proof-by-contradiction: showing the program is not incorrect. The essential idea is to demonstrate that the program never produces the incorrect output for each valid input. The following definitions are given to facilitate discussion of this approach.

Definition: For an arena $G = < P, S, D >$, $x \in D$ is *successful* if and only if $[P](x)\downarrow$ and $(x, [P](x)) \in S$, otherwise x is a *failure*. A test set T for G is successful if and only if each $x \in T$ is successful for G. The set of all successful points for G is called the *successful* set of G. The set of all failure points for G is called the *failure set* of G.

A test set is successful, therefore, if and only if the program computes the correct output as determined by the specified relation for each point in the set. Clearly if the test set is both exhaustive (covers the entire input domain) and successful, the program being tested is correct.

Definition: The *defined failure set* of an arena G is all members of the failure set for which $[P]$ is defined. The *undefined failure set* of an arena G is all members of the failure set of G for which $[P]$ is undefined.

Definition: A halting test set T for an arena $G = < P, S, D >$ is *reliable* if and only if: T successful for G implies P correct with respect to S.

Given these definitions, the following theorem follows directly.

Theorem: A program P is correct with respect to a specification S if and only if P's failure set is empty.

Exploring the relationship between testing and program correctness may be viewed as investigating the characteristics of the failure set. Can the failure set be a random collection of points, or must it conform to some pattern? The limitations of this approach become immediately apparent when the following is realized.

Theorem: The failure set of an arena is not necessarily recursively enumerable.

Proof: For every program it is possible to construct an arena in which the failure set is exactly the set of points on which the program does not fault. (This is accomplished by choosing the degenerate specification S in which any output is considered correct. In such an arena, the only way to fail is to not halt.) Since some programs do not have recursively enumerable halting sets [14], it follows that not all the constructed arenas for these programs have recursively enumerable failure sets.

When faced with trying to deduce information about classes which include nonrecursively enumerable sets, not much can be done. For example we have the following.

Theorem: A reliable test set exists but can neither be generated nor recognized for an arbitrary arena.

Proof: If a program is correct then any subset of the domain of the specification is a reliable test set. If the program is incorrect, then there must be some input x in the failure set. The set $\{x\}$ is therefore vacuously reliable. Hence a reliable test set always exists. Suppose a reliable test set can be generated for an arbitrary arena. Then a program R can be constructed for the function which determines program equivalence

$$[R](P, S) = \begin{cases} 1 & \text{if } [P] = [S] \\ 0 & \text{otherwise} \end{cases}$$

by generating a reliable test set for P with respect to S, executing P on this test set, and checking agreement with S. If they agree, then they are equivalent; produce 1. Otherwise, produce 0. However if program equivalence is solvable, then it would be decidable whether or not an arbitrary function is the zero function. In particular it would be possible to determine whether or not

$$f_P(k) = \begin{cases} 1 & \text{if } [P](P)\downarrow \text{ in less than } k \text{ steps} \\ 0 & \text{otherwise} \end{cases}$$

is everywhere zero for an arbitrary program P. But f_P is everywhere zero if and only if $[P](P)\uparrow$. Thus, the ability to generate a reliable test for P with respect to S implies the ability to decide whether or not P will halt when supplied itself as input. This, of course, cannot be done, leaving the conclusion that reliable tests cannot be generated for arbitrary programs and specifications.

Thus, a reliable test set cannot be generated. Neither can it be recognized, because then a generator could be constructed by producing successively larger test sets until the recognizer declares one to be reliable.

C. Research Direction

Since failure sets need not be recursively enumerable, to obtain a testing theory with practical import it is necessary to make some assumptions about the failure sets of the arenas under consideration. For example, a positive result can be obtained when it is possible to decide when the program halts.

Theorem: For any area $G = < P, S, D >$, if dom $([P])$ is recursive then the failure set, defined failure set and the undefined failure set are recursive.

Proof: Since dom $([P])$ is recursive it is decidable whether or not P is undefined for any input x. Thus, since dom $([S])$ is recursive, the undefined failure set is recursive. If P is defined for an input x, then x is either in the successful set or in the defined failure set. To decide which, compute $[P]\ (x)$. Since the specified relation is recursive, it is decidable whether or not $(x, [P]\ (x))$ is in $[S]$ for any x in dom $([P])$. If $(x, [P]\ (x))$ is in $[S]$ then x is a member of the successful set, otherwise it is member of the defined failure set.

This theorem represents an example of incorporating realistic assumptions about programs and specifications into a theory about testing. It is not unrealistic to assume that program halting is decidable for most practical programs. For example, in many situations a program that does not compute an output within a specified time limit is considered incorrect. Since halting within a specified time limit is decidable the above theorem is then applicable. Unfortunately, even in the presence of decidable halting reliable test sets still cannot be generated or recognized, since the proof of the earlier theorem still holds. Thus to demonstrate correctness via testing, it is necessary to restrict the failure set even more.

Assumptions about the failure set implicitly limit the class of programs considered likely to meet a given specification. Without any assumptions, testing must distinguish the original program from all programs. This is impossible, as shown by the above result on reliable test sets. Conversely, by suitably restricting the class of programs that must be considered (and implicitly restricting the resulting failure set), testing can sometimes demonstrate correctness under the assumptions.

The class of alternate programs can be specified by defining transformations to be applied to the program under test. If the transformations represent typical programmer mistakes, the goal of testing is then to show that the programmer did not make any of these mistakes. Such testing is called *fault-based* and is the method formalized and discussed in the following sections.

III. Fault-Based Testing

Fault-based testing is closely related to the alternate way of showing program correctness mentioned above: show that the program is not incorrect. If a program has limited potential for being incorrect, then a test set demonstrates correctness when it shows the potential is not realized. The means of specifying incorrectness taken here is to define potential faults for program constructs.

A *location* in a program denotes some expression of the program. Exactly what constitutes an expression is language dependent. An *alternate expression* (an *alternative*) is an expression f which can be legally substituted[1] for the expression designated by l. The resulting program is called an *alternate program* and is denoted by P_f^l. A set of alternatives for an expression is called an *alternative set.* An alternative set always contains the original expression unless otherwise stated.

A *fault-based arena* encapsulates the important components of fault-based testing, namely an arena along with a designated set of locations and alternate expressions for those locations. A fault-based arena is therefore a 5-tuple, $E = < P, S, D, L, A >$, where $< P, S, D >$ is an arena, $L = (l_1, l_2, \cdots, l_n)$ is an n-tuple of locations in P, and $A = (A_1, A_2, \cdots, A_n)$ where each A_i is an alternative set associated with location l_i, $1 \leq i \leq n$. The inclusion of the set L allows us to focus on particular locations within the program under test, for example, those locations identified with more critical sections of code. The set A sharpens our focus even more, by limiting consideration to only specified faults at the chosen locations. Note that there is no implied restriction on the size of either set. On one extreme there can be just one location with one associated alternative (our earlier example); on the other extreme, infinitely many alternatives may be allowed, at every place in the program where they may be legally substituted.

The set of all alternate programs for E comprises those programs generated by substituting one or more alternatives in their respective locations in the program P. It is denoted by P_E. The program P is called the *original* program. When discussing an expression at a particular location, the expression is called the *designated expression.*

The common characteristic to all fault-based schemes is that they use information extracted from the execution to make such deductions; they vary on how much of the information they use. The semantic operator [] captures only the input–output behavior of the program. What is needed is a similar operator which captures the additional information from which to deduce that an alternate program is behaving differently. We therefore let $< P >$ denote the behavior of the program P, where behavior may include any run-time characteristic of P on an input, for example its output, its intermediate states, its running time, or its space consumption. Thus $< P >$ need not even subsume $[P]$. The only required connection between

[1]A substitution is legal if and only if the resulting program compiles without error.

$[P]$ and $<P>$ is that $<P>$ must be defined whenever $[P]$ is defined. For the purpose of this paper, however, we will adopt the view that $<P> = [P]$.

In fault-based testing test data is sought that *distinguishes* (or *differentiates*) the original program from its alternate programs, where "distinguishes" depends upon the choice of interpretation of $<P>$. To make this concrete, we say for a program P and $x \in \text{dom}([P])$, *x distinguishes* (or *differentiates*) P from a program R if and only if

$$<P>(x) \neq <R>(x), \text{ or}$$
$$<R> \text{ is not defined for } x.$$

Differentiation can be extended to both test sets and sets of programs. A set $T \subseteq \text{dom}([P])$, *T distinguishes P* from a program R if and only if there is an x in T such that x distinguishes P from R. For an arbitrary set of programs $\boldsymbol{R}$, program P, and set T, *T distinguishes P* from $\boldsymbol{R}$ if and only if for each program R in $\boldsymbol{R}$, T distinguishes P from R.

A. Properties of Fault-Based Testing

This section explores those properties it would be advantageous for a fault-based testing arena to possess. First to be discussed are properties necessary for a program to be proved correct. Other desirable properties are then discussed which, if possessed, reduce the complexity of testing.

To better understand the context and restriction of fault-based testing, it is useful to consider under what circumstances fault-based testing would ensure correctness. That is, what assumptions must be satisfied by a fault-based arena to ensure no faults remain in the code after distinguishing all nonequivalent alternatives? It should be recalled, however, that ensuring correctness is not the primary goal of fault-based testing. It is useful, however, to know when correctness does occur.

There are two necessary and sufficient conditions for fault-based testing to ensure that a program is correct with respect to its specification:

1) the fault-based arena must be alternate-sufficient, and

2) coupling does not occur for the test set.

Each of these is defined and discussed below.

Fault-based testing can only produce conclusions about programs associated with the fault-based arena. Thus, if a program is to be declared correct via fault-based testing it must be either the original program or one of its alternates. Multiple alternates, formed by several substitutions, can also be considered if coupling does not hold (as described below). Hence, one of these programs associated with the arena must be correct for fault-based testing to make such a conclusion. A fault-based arena which contains a correct program is called *alternate-sufficient*. Formally, a fault-based arena $\boldsymbol{E} = <P, S, D, L, A>$ is *alternate-sufficient* if and only if there exists $P \in \boldsymbol{P_E}$ such that P is correct with respect to S.

Alternate-sufficiency captures the idea that at least one alternate program is correct. Clearly, there are fault-based arenas which are not alternate-sufficient. Fortunately, there are fault-based arenas which are alternate-sufficient as well. For example, it may be known that a certain library program exists and is correct (shown, perhaps, by formal proof), but its name has been forgotten. A test set can be developed which selects the desired program from the library. Another example is when it is known that the specification can be implemented by a program from a given class, as in the case of algebraic program testing [15].

It would, of course, be quite useful to have an algorithm to decide whether or not an arbitrary fault-based arena is alternate-sufficient. Unfortunately, no such algorithm exists.

Theorem: It is undecidable whether or not an arbitrary fault-based arena is alternate-sufficient.

Proof: Consider the fault-based arena $\boldsymbol{E} = <P, S, D, \phi, \phi>$. In this arena there are no locations or alternate sets, making P the only member of $\boldsymbol{P_E}$. Therefore the arena is alternate-sufficient if and only if P is correct with respect to S. There is no algorithm for deciding the correctness of a program, hence there is no algorithm for deciding if the arena is alternate-sufficient.

The noneffective nature of alternate-sufficiency means that it must be assumed in general. This affects fault-based testing in much the same manner as the halting problem affects conventional testing: it has no impact until it is proven false. Discovery that the assumption is not satisfied identifies additional faults, which, of course contributes to the goal of fault-based testing.

Fault-based testing frequently proceeds on the assumption that alternates can be precluded one at a time without side effects. Alternatives are *coupled* if each is individually precluded by a test, but the combination is not precluded by the same test. If no set of alternatives are coupled, then distinguishing all single alternate programs distinguishes all multiple alternate programs. This assumption therefore allows for a broader class of programs to be distinguished by a test set. The absence of coupling can be inferred from the structure of the program in some cases. It is impossible for two alternatives to interact if they do not appear on the same path, or if they do not affect the same parts of the data state. A formal description of coupling can be found in [16]. In [12] it is shown that there is no algorithm for deciding whether or not coupling exists in an arbitrary fault-based arena.

It is clear from the above that fault-based arenas must possess some rather strong properties in order to conclude correctness from fault-based testing. Other desirable properties are now discussed.

If each alternative set is finite then E is called *bounded* else it is *unbounded*. Unbounded arenas are more likely to encompass potential faults. Techniques for handling unbounded arenas are therefore desirable. A fault-based arena $\boldsymbol{E} = <P, S, D, L, A>$ is *finitely distinguishable* (or simply *finite*) if and only if there is a finite test set T

for E such that T distinguishes P from P_E. If no finite test set distinguishes all alternative programs, then conventional test sets, being finite, are ineffectual. Thus other methods of testing, such as symbolic execution [17], [18], must be employed. Such a scheme is discussed in the next section.

In a bounded fault-based arena, there are finitely many locations and alternatives, hence there are finitely many alternate programs. Therefore, we have the following theorem.

Theorem: Every bounded fault-based arena is finitely distinguishable.

Mutation testing [6] is an example of a bounded fault-based arena and is therefore finitely distinguishable. In mutation testing the assumption that the arena is alternate-sufficient is called the *competent programmer hypothesis* [6]. This states that a competent programmer will write a program that is syntactically close to the correct program, i.e., that the program will fall into a restricted class of programs, one of which is correct. Since the mutation testing arena is bounded, all alternate programs can be generated. The set of alternate programs is unfortnately huge, and the undecidable program-equivalence problem remains.

The limitation of mutation testing to bounded fault-based arenas is quite restrictive. It is certainly desirable to extend fault-based testing to unbounded fault-based areans. This, however, cannot be done in general since an unbounded fault-based arena can require an infinite test set to differentiate all the alternate programs. An example is given later. Fortunately, this is not always the case. Consider the program:

```
print 5
```

Suppose the alternative expressions are "any constant" and the location is that of the constant 5. Then the infinite set of alternate programs is the set of all constant programs. Any nonempty finite test set differentiates the original program from all alternate programs. Thus, we have proven the following.

Theorem: There are unbounded fault-based arenas that are finitely distinguishable.

A second problem with mutation testing relates to the phenomenal number of mutants that can be generated for even a simple program. Mutation testing [6] assumes that simple faults are linked to complex faults in such a way that a test set which differentiates a program with simple faults, differentiates programs containing complex faults. This assumption (dubbed the "coupling effect") reduces the number of alternate programs that must be generated. It is not verifiable since "simple" and "complex" are not precisely defined. Under the interpretation that "simple" means "single mutation" and "complex" means "multiple mutation" the coupling effect does not universally hold but does probabilistically hold under certain circumstances [16]. Empirical evidence supports this assumption as well [6], [19].

Mutation testing provides a good model for exploring bounded, and hence finitely distinguishable fault-based testing arenas. The next section discusses a fault-based testing scheme called symbolic testing that allows for unbounded arenas that need not be finitely distinguishable. Properties of symbolic testing are discussed in the succeeding section.

IV. Symbolic Testing

The previous section presented a model for understanding fault-based testing. It suggested that the absence of faults may be deduced from the behavior of a program on a test case, but it provided no specific guidance on how such inference can be made. This section presents a fault-based testing strategy called *symbolic testing* which achieves exactly that. First symbolic testing is formalized and illustrated. Ramifications of the method are then developed for a simple language. Since fault-based testing allows unbounded arenas, undecidable issues are discussed in the next section for the infinite alternative set of constant substitutions. An extension of results for constant substitution to other unbounded arenas is discussed. The last section discusses the complications that arise in the presence of multiple paths.

A. Motivation and Basic Definitions

Symbolic execution [17] underlies the idea of symbolic testing. The forte of symbolic execution is its ability to model infinitely many executions by a single symbolic execution. Historically this capability has been used to model the parallel execution of one program on an infinite set of inputs. This is accomplished by using a symbolic input to represent all inputs which follow a given path. In such a system, execution proceeds as usual until an input statement is reached. A symbolic input is then obtained and stored as the value of the appropriate variable. Whenever the variable is referenced, its symbolic value is introduced into the computation. This can result in a symbolic expression being stored in another variable. For example, in

```
read x
y := x + 1
print y
```

if the symbolic input X is read, then the value of y on output is $X + 1$. A symbolic input represents all inputs that follow a particular path. A symbolic execution system therefore determines the function computed by any given path by expressing the output in terms of the symbolic input.

Symbolic testing generalizes symbolic execution by providing the ability to simultaneously execute infinitely many alternate programs. What is necessary is to modify this ability to execute infinitely many instances of the same program in order to be able to execute infinitely many alternatives to the same program. For symbolic execution the key lies in encoding infinitely many inputs by a single *symbolic input*. For symbolic testing the key lies in encoding infinitely many alternatives by a single *symbolic alternative*. Interpretation of a program containing a symbolic alternative proceeds as usual until the expression containing the symbolic alternative is reached. At that point it is necessary to simulate the result of all alternatives represented by the symbolic alternative. For example, consider the program

```
read (x, y)
x := x * y + 3
write (x * 2)
```

Symbolic testing can be used to ensure that no mistake was made in selecting the constant 3 as follows. First the constant 3 is replaced by a symbolic alternative, say F, producing the following transformation:

```
read (x, y)
x := x * y + F
write (x * 2)
```

Note that this transformation denotes infinitely many alternate programs, one for each constant that can replace F. Upon executing the transformation with input $x = 5$ and $y = 6$, the second statement will introduce into the program state a *symbolic value* of $30 + F$ into the data state of the program. A symbolic value is created as the value of the symbolic alternative in much the same way as a symbolic value is created for an input in conventional symbolic execution. As x is then used in succeeding statements the symbolic value F will propagate through the program and ultimately will appear in the output, which is $(30 + F) * 2$ in this case.

The output succinctly describes the impact any of the denoted alternatives would have on the output of the program. The original program computes 66 for input (5, 6) and we would confirm its correctness against the specification. Other outputs that would have occurred had a substitution been made for 3 are determined by substituting that value for F in $(30 + F) * 2$.

Recall that our goal is to demonstrate that no other constant could have been substituted for 3 and have gone undetected for our test set. But for a substitution to detected for input $x = 5$ and $y = 6$ would mean it would have to produce some value other than 66. Thus we need to determine those values of F which would have caused an output other than 66 to be computed. Thus we need to solve the equation

$$(30 + F) * 2 = 66$$

for F. A little algebra yields

$$30 + F = 33, \quad \text{so } F = 3.$$

Thus no constant other than 3 can be substituted for 3 and go undetected on this test case. Thus the test set $\{(5, 6)\}$ distinguishes P from $\boldsymbol{P_E}$ where $\boldsymbol{P_E}$ contains all alternate programs produced by substituting any constant for 3 in P.

To formalize the above ideas, we introduce the notion of a symbolic arena. The components of such an arena consist of those of a fault-based arena, augmented to allow for symbolic values and symbolic alternatives. Formally, a *symbolic arena* $S = < P, S, D, L, A >$ is a fault-based arena $< P', S, D, L, A >$ augmented in the following ways:

1) dom $([P])$ = dom $([P'])$ $\cup$ an infinite set of *symbolic inputs*.

2) $P = P'$, except for the replacement of one expression in P' with a *symbolic alternative*. Referencing such an alternative introduces a unique *symbolic value* into the computation.

Including symbolic inputs (a subset of symbolic values), enables conventional symbolic execution and therefore allows for infinitely many inputs to be represented in a single execution. Including symbolic alternatives enables the representation of another infinite class of executions: that of an infinite set of alternate programs. This is achieved by creating a program called a *transformation* which represents all alternate programs induced by an alternative set. This transformation is created by replacing a designated expression with a symbolic alternative. The symbolic alternative "stands in" for members of the alternative set. The transformation is then symbolically executed along a chosen path, as discussed above.

The transformation produced by replacing the expression at location l in P with symbolic alternative F is denoted by $\boldsymbol{P}^l_F$. This transformation represents a class of alternate programs; it is not an alternate program itself. The transformation $\boldsymbol{P}^l_F$ represents all alternate programs $\boldsymbol{P}^l_e$, for all e in the alternative set.

In the example given above the output of the transformation can be used to deduce what alternatives could not have been substituted and remain undetected for a particular test set. This is done by comparing the output of the original program to the symbolic output of the transformation. Those values of the symbolic alternative which satisfied the inequality would be distinguished by the test set. The inequality established is called a *propagation assertion*, the solution of which is the set of values that alternatives could not assume and remain undetected by the test set. Formally, for $x \in$ dom $([P])$, a *propagation assertion* for transformation $\boldsymbol{P}^l_F$ is

$$[P](x) \neq [\boldsymbol{P}^l_F](x).$$

The negation of a propagation assertion is called a *propagation equation*. If any solution to a propagation equation is assumed by one the alternatives for a given input, the corresponding alternate program would not be distinguished by that input. Propagation assertions and equations are classified by whether the input is symbolic or actual. If x is a symbolic input then the assertion (or equation) is *general*, otherwise it is *specific*. If the symbolic form of the computation for a class of alternate programs can be determined for an input x, then this form can be equated to the computation of the actual program. A propagation assertion involves both the symbolic fault and the actual or symbolic input values.

The symbolic alternative in the propagation equation serves a dual role, as a free variable over the value space of the designated and as a free variable over the alternate set. As a free variable over the value space, the symbolic alternate represents those values which, if substituted for the designated expression, would yield alternate programs that are not distinguished by the input value. As a free variable over the alternate set, the symbolic alternate represents those alternatives which, if substituted for the designated expression, would yield alternate programs that are not distinguished by the input value.

Once this solution set has been found the symbolic fault can function as a free variable over the alternate set in equations known as *fault equations*. The solution set of a fault assertion comprises those alternatives which, if substituted for the designated expression would produce programs that are not differentiated by the input.

For example, consider the following program to swap two integers:

```
read x, y
x := x + y
y := x − y
x := x − y
print x, y
```

which produces on input 2, 3:

```
x: 3
y: 2
```

Suppose that the reference to y in the first assignment statement is incorrect. Replacing this expression with symbolic alternative F yields the program

```
read x, y
x := x + F
y := x − y
x := x − y
print x, y
```

Reexamining the program for the same input (2, 3) produces the symbolic output:

$$x\text{: } 2 + F - (2 + F - 3)$$
$$y\text{: } 2 + F - 3.$$

The output encodes the functions computed by all the alternate programs represented by the (unspecified) alternative set. One propagation equation is produced by equating the two values computed for y:

$$2 + F - 3 = 2.$$

Treating F as a free variable over the value space and solving, we get $F = 3$. Thus, $\{3\}$ is the solution set to the propagation equation. In this case replacing y with any expression that computes 3 yields an alternate program that is not distinguished by the input (2, 3).

Once this solution set to the propagation equation has been found the symbolic fault can function as a free variable over the alternate set in equations known as *fault equations*. The solution set of a fault equation comprises those alternatives which, if substituted for the designated expression would produce programs that are not differentiated by the input. In this case no alternate that evaluates to a constant other than 3 in the data environment defined by the original execution could be substituted in the original program and not be differentiated by this test. For example, substituting $x + 2$ for F would be detected for the input (2, 3), but substituting $x + 1$ would not.

Consider then some potential alternative sets for the designated expression. If F represents the class of constant substitutions, then no constant other than 3 could be substituted and the output be correct for this input. Thus, all constants other than 3 are eliminated by this test. An additional input is therefore necessary to eliminate the alternate program formed by substituting 3 for y.

If F represents the class of variable substitutions, then an inspection of the state at the designated expression reveals that no variable other than y has the value 3. Thus the test eliminates this alternative set. If F represents the class of "variable + constant" faults, then the only possible substitutions are $x + 1$ and $y + 0$ since the alternative must evaluate to 3 in the current state, which has $x = 2$ and $y = 3$. The second alternative is equivalent to the original expression. An additional input is required to eliminate the first alternative.

Another propagation equation can be generated by equating the two computed values for x:

$$2 + F - (2 + F - 3) = 3.$$

This however simplifies to $3 = 3$ and thus no alternative programs can be eliminated based on this test. This curious result occurs exactly when the symbolic output, though containing references to a symbolic alternative, simplifies to a constant. Thus, despite the fact that the presence of a fault contributes to the output, its contribution cancels itself.

Executing the program for input (8, 5) yields the following set of equations:

$$8 + F - 5 = 8$$
$$8 + F - (8 + F - 5) = 5.$$

Again the second equation simplifies to $5 = 5$. The first equation simplifies to $F = 5$. This eliminates the class of constant substitutions because no constant can evaluate to 3 on one execution and 5 on another. It also eliminates the class of variable + constant substitutions because the only possible substitution this time is $x + (-3)$ which is not equivalent to the previously determined substitution.

Therefore the test set $\{(2, 3), (8, 5)\}$ is sufficient to eliminate all the aforementioned alternatives.

The following is an example of a symbolic execution using symbolic inputs (B, C, T).

```
read $b, c, t$
$t := b + c * b$
$b := b * t - c$
$c := c + b * t$
print $t, b, c$
```

The resulting output is:

t: $B + (C * B)$

b: $B * (B + (C * B)) - C$

c: $C + ((B * (B * (C * B)) - C) * (B + (C * B)))$.

Introducing a symbolic alternative into the first assignment statement

$$t := b + c * F$$

and reexecuting yields:

t: $B + (C * F)$

b: $B * (B + (C * F)) - C$

c: $C + ((B * (B + (C * F)) - C) * (B + (C * F)))$.

Note that the output is expressed in terms of the symbolic inputs and the symbolic alternative.

Generating the propagation equation for t yields:

$$B + CF = B + CB$$

thus $F = B$ (provided $c \neq 0$). Symbolic alternative F must therefore have as its value B. Thus, only alternatives which evaluate to B in the data environment will not be differentiated.

The propagation equation for b yields the same conclusion:

$$B(B + CF) - C = B(B + CB) - C$$

thus

$$F = B$$

If c were the only output, then evaluation would be harder. The propagation equation is

$$C + ((B(B + CF) - C) * (B + CF))$$
$$= C + ((B(B + CB) - C) * (B + CB)).$$

Simplifying produces

$$(F - B)(2B^2C + BC^2(F + B) - C^2) = 0.$$

Solving this for F yields:

$$F = B \quad \text{and} \quad 2B^2C - C^2 + BC^2F + B^2C^2 = 0.$$

For this propagation condition there is a solution other than the default solution of $F = B$, namely

$$F = (-B^2C + C - 2B^2)/(BC).$$

The division operator in this solution is algebraic and not integer division. If F must be an integer, the solution holds only for those inputs which produce an even division. Examples of this are when $b = 1$, and c a member of $\{-3, 1, 3\}$. Thus, for these inputs the original program would not be distinguished from the alternate program containing the alternate statement:

```
$t := b + c$
    $* (-1 * b * b * c + c - 2 * b * b)/(b * c)$
```

or any algebraic equivalent.

B. Example

The techniques of the previous section will now be illustrated on a simple program (Fig. 1) to compute the area of $f(x) = x^2 + 1$ for x between a and b. Before proceeding, however, a lemma will prove useful in the analysis that follows.

Lemma [20]: If $f(x_1, x_2, \cdots x_n)$ and $g(x_1x_2, \cdots x_n)$ denote two multinomials in n variables that agree on an n-dimensional subset that is infinite in each coordinate, then $f = g$ for all points.

Executing ComputeArea (Fig. 1) for symbolic input A, B, I produces

$$(A * A + 1) * (B - A)$$

assuming $B \geq A$ and $A + I > B$ (i.e., skipping the loop). Introducing an assignment fault in statement 3:

$$\text{area} := F$$

and following the same path produces

$$F + (A * A + 1) * (B - A).$$

Forming the general propagation equation:

$$F + (A^2 + 1)(B - A)$$
$$= (A^2 + 1)(B - A)$$

and solving for F gives

$$F = 0.$$

Clearly, if F represents the class of constant substitutions, then the only solution is $F = 0$. Suppose, however, that F denotes the set of all polynomials in a of degree $\leq M$. Call this class of polynomials $\boldsymbol{P}^+(M)$. In this case, the fault equation is written as $F(a) = 0$ to indicate that the alternative set consists of expressions involving the variable a. Clearly each member of $\boldsymbol{P}^+(M)$ which has no real zero has been differentiated. But more importantly, by the above lemma all members of $\boldsymbol{P}^+(M)$ other than the zero polynomial have been differentiated, because the solution $F(a) = 0$ must hold for infinitely many values of a since the path domain allows infinitely many a's. Thus if the symbolic output for this path is verified to be

```
Program ComputeArea (input, output);
var a, b, incr, area, v : real;
begin
1  read (a,b,incr); {incr > 0}
2  v := a*a + 1;
3  area := 0;
4  while a + incr ≤ b do begin
5      area := area + v * incr;
6      a := a + incr;
7      v := a*a + 1
   end;
8  incr := b - a;
9  if incr ≥ 0 then begin
10     area := area + v * incr;
11     writeln ('area by rectangular method:', area)
   end else
12 writeln ('illegal values for a=', a, ' and b=', b)
end.
```

Fig. 1. Compute the area for $x^2 + 1$ over the interval $[a \,..\, b]$.

correct according to the specification, then all nonzero polynomial substitutions for 0 are eliminated.

In the case when only real outputs and not symbolic outputs can be verified, the remaining alternate expressions can be eliminated by algebraic testing [15]. In this case many $M + 1$ inputs that satisfy the path condition and differ in the a component at the designated expression need be chosen [20].

Let us now consider a different location and class of alternate expressions. Suppose it is suspected that something is wrong with the expression found in statement 5:

$$\text{area} := F$$

To reach this statement requires selecting a different path. Executing CalculateArea without the symbolic fault on symbolic inputs A, B, I satisfying path condition $A + I \leq B \,\&\, A + 2I > B$ yields symbolic output of area:

$$(A^2 + 1)I + [(A + I)^2 + 1](B - A - I).$$

With the symbolic fault inserted at statement 5 the symbolic output is:

$$F + [(A + I)^2 + 1](B - A - I).$$

Forming the propagation equation and simplifying yields

$$F = (A^2 + 1)I.$$

Again it is clear that no constant substitution is possible, since for the specified path neither $A^2 + 1$ nor I need be 0. A more plausible situation would be that F is a different multinomial in the variables *area*, v, and *incr*. Let $F(\text{AREA}, V, \text{INCR})$ denote this multinomial. In this case we have the fault equation:

$$F(0, A^2 + 1, I) = (A^2 + 1)I$$

since 0, $A^2 + 1$, and I are the values of *area*, v, and *incr*, respectively, at the point in execution when the symbolic fault is first encountered.

Now due to the value of 0 in the first parameter position, many possible substitutions for F arise of the form

$$F(x, y, z) = g(x, y, z) + yz \qquad (1)$$

where $g(x, y, z) = 0$ whenever $x = 0$. For example, the alternate expression (area * incr + value * incr) would not be detectable on this path. (Zeil calls expressions such as g, *blindness expressions* since input data are blind to the presence of such expressions [30].)

These blindness expressions can be eliminated by executing the loop twice. The resulting propagation equation is

$$\begin{aligned} F(F(0, A^2 + 1, I);\ & (A + I)^2 + 1; I) \\ & + [(A + 2I)^2 + 1][B - A - 2I] \\ &= (A^2 + 1)I + [(A + I)^2 + 1]I \\ & + [(A + 2I)^2 + 1][B - A - 2I] \end{aligned}$$

which simplifies to

$$\begin{aligned} F(F(0, A^2 + 1, I);\ & (A + I)^2 + 1; I) \\ &= (A^2 + 1)I + [(A + I)^2 + 1]I \qquad (2) \end{aligned}$$

substituting $F(0, A^2 + 1, I) = (A^2 + 1)I$ into this equation and rewriting as (1) yields

$$\begin{aligned} F((A^2 + 1)I,\ & (A + I)^2 + 1; I) \\ &= g((A^2 + 1)I, (A + I)^2 + 1, I) \\ &\quad + [(A + I)^2 + 1]I. \end{aligned}$$

Equating this with (2) and simplifying gives

$$g((A^2 + 1)I, (A + I)^2 + 1, I) = (A^2 + 1)I.$$

Thus, $g(x, y, z)$ agrees with the multinomial x on a three-dimensional subset that is infinite in each dimension, so by the lemma, $g(x, y, z) = x$ and $F(x, y, z) = x + yz$ must be the function computed by any multinomial substituted for F. This is, of course, exactly the function of the multinomial already in the program. Note, again, that this conclusion is based on the assumption that the symbolic outputs have been verified as being correct. Otherwise it is again necessary to resort to algebraic testing [15].

As a final example consider analyzing a problem which affects the control flow. Introduction of a symbolic fault into the predicate at line 4 will produce different output only if fewer or additional loop iterations result. Suppose we are concerned with an off-by-constant fault in statement 4 represented by

$$\text{while } a + \text{incr} + F \leq \text{do begin.}$$

Assume the original program loops zero times, i.e., $A + I > B$. The alternate programs will loop additionally if and only if $A + I + F \leq B$. Thus, $F < 0$. The loop is an accumulator loop [21] with a monotonic increasing accumulation function, hence additional iterations produce different output for the loop accumulation variable (*area* in this case) [22]. Thus any general propagation equation formed will have no solutions. This implies that any value of $F < 0$ has been eliminated. A similar analysis for values of $F > 0$ which cause loops to iterate fewer times enables the conclusion that $F \leq 0$. Hence $F = 0$. Algebraic arguments again enable the elimination of any multinomial in program variables of a given degree.

C. Theoretical Considerations

Although it was the case that all the alternates considered in the previous section were eliminated, it is natural to ask whether or not this is always possible. This section characterizes the computability limitations on fault-based testing. Results from this section are thus applicable to any fault-based testing scheme that seeks to ensure the elimination of designated faults. Detailed proofs can be found in [12]. The goal here is to overview these results.

The first issue addressed is the relationship between specific propagation equations and general propagation equations. How are specific propagation equations related to general propagation equations? Each is developed from one program path. The general propagation equation represents the computation of all inputs in the path domain. Thus, the general propagation equation encompasses the specific propagation equation in that the specific equation results from substituting a member of the path domain for the symbolic input in the general equation. This is formally stated in the following theorem.

Theorem: The solution set for a general propagation equation for path P is the intersection of the solution sets of all the specific propagation equations generated from P.

Proof: A solution to a general propagation equation is a value that could be substituted for the designated expression that would not be differentiated, no matter what input is selected from the path domain of path P. Thus it would remain undifferentiated for any particular input and hence would be in the solution set for any specific (nongeneral) propagation equation generated from this path. Conversely, any solution to all the specific propagation equations must indeed solve the general propagation equation since each specific propagation equation can be derived from the general propagation equation by substituting the input values for the symbolic inputs.

As a result of this theorem it is clear that a specific propagation equation for input x may have more solutions than a general propagation equation for the path determined by x. This satisfies our intuition since a specific propagation equation embodies less information from the program and hence is less powerful in eliminating alternatives. As illustrated in the previous section, however, it is frequently the case that specific propagation equations are more easily solved than general propagation equations. Thus, both provide a useful means for eliminating alternatives.

A second area of interest is the decision problems associated with infinite alternative sets. An infinite alternative set implies infinitely many alternate programs. It is possible that an arena with such an alternative set is not finite, i.e., that no finite test set exists which differentiates the original program from all its alternates. It would be advantageous to know when this occurs. This problem is discussed below for arenas with restricted programs and alternate sets.

The program class has the following features.

1) Computation: Arithmetic expressions using +, −, *, / (integer division).

2) Control: If and while statements.

3) Input/Output: The program begins with a read statement for a single variable that is executed once at program invocation. The program terminates with a sequence of print statements that are executed only once when the program halts.

4) Data: All inputs, outputs, and internal values are integers. Division by zero is treated as a fatal error: no output is produced and the program function is undefined for that point.

The class is general enough to compute all computable functions.

The alternative set chosen is that of "all constants." Thus, a program expression may be replaced by an arbitrary constant. This is the only infinite class whose members can be evaluated independent of where they are located. In Section III-B it was shown that a bounded arena is finite, i.e., an arena with finite alternative sets requires only a finite test set to differentiate the original program from its alternates. When an arena is unbounded, however, it is possible that no finite test set suffices. An earlier example showed that this is not the case for a very particular (location, alternative set) pair. The discussion below applies to more general circumstances in which unbounded arenas are finite.

Two factors influence whether an arena is decidably finite for constant substitutions. One is the unrestricted use of the division operator. The other is alternatives which affect the program flow. It is shown that the presence of either inputs that infinite arenas can result, and that it is undecidable when they do. It is also shown that the absence of both produces finite arenas.

Consider program transformations which can affect the flow of the program.

Definition: A transformation P_F^l of P is *domain independent* for an input x if and only if the conditions along the path taken by x in P evaluate the same as the conditions along the path followed by x in P_F^l, for all alternatives represented by F. P_F^l is *domain independent* for a set T if and only if it is domain independent for each member of T. If no inputs are specified, then the input set dom ([P]) is implied. If a transformation is not domain independent, then it is *domain dependent*.

Domain independence captures the intuitive notion of a program change affecting the flow of control. As might be expected, there is no algorithm for deciding if a transformation is domain independent for any x. There are, nevertheless, algorithms in data flow theory [23] which calculate all variables (and hence all expressions) that have the potential of affecting a given expression. If the given expression is a boolean condition then these algorithms pinpoint those expressions which may affect program flow. Hence they also determine those which cannot affect program flow. The undecidable issue lies in the middle ground—those expressions which have the potential of affecting the program flow, but do not.

For example, consider a symbolic arena with program

```
if x = 1 then y := 1
else y := x * x
```

and alternative set "all integers" substituted for the constant 1 in the expression "$x = 1$." The program transformation for this alternative set is

```
if x = F then y := 1
else y := x * x
```

Clearly the transformation is domain dependent. The curious thing is that for every alternate program there is a single input which differentiates it from the original. Furthermore, no input differentiates more than one alternate program. Thus no finite test set exists which differentiates all its alternate programs. The arena is therefore infinite. The following has therefore been proved.

Theorem: An arena can be infinite for domain dependent transformations representing constant substitutions.

Domain dependent transformations can introduce unwanted difficulty for error-based testing. But how great is this difficulty? For instance, even though they can produce infinite arenas, perhaps it is decidable when this occurs. If so, other techniques (such as static error-based testing [22]) can be tried. Unfortunately, this cannot be decided [12].

Theorem: There is no program Q that computes a function satisfying

$$[Q]\,(P, P_F^l) = \begin{cases} T & \text{if the arena is finite} \\ 0 & \text{otherwise} \end{cases}$$

where T is a nonempty set which differentiates P from P_e^l, for all integers e, P satisfies the program restrictions above, and P_F^l represents a class of constant substitutions which are domain dependent.

For domain independent transformations it appears there may be hope of dealing with infinite arenas. Since these transformations do not affect the program control flow, the same path must be followed in the original and all alternate programs. The functions computed by each of these paths is a rational form and the equivalence of such forms is known to be decidable by a finite test set [24]. However, the question at hand is whether a finite test set always exists to differentiate a rational form from infinitely many such forms. In [12] it is shown that a conditional statement can be replaced by assignments using rational forms. Thus, the example above can use rational forms and hence the previous two theorems apply.

Theorem: An arena can be infinite for domain independent transformations representing constant substitutions.

Theorem: There is no program Q that computes a function satisfying

$$[Q]\,(P, P_F^l) = \begin{cases} T & \text{if the arena is finite} \\ 0 & \text{otherwise} \end{cases}$$

where T is a nonempty set which differentiates P from P_e^l, for all integers e, P satisfies the program restrictions above, and P_F^l represents a class of constant substitutions which are domain independent.

The previous two theorems rely on the fact that rational forms can be used to simulate the effect of conditional statements, thereby introducing hidden branching in what would seem to be straight-line code. Multinomials cannot be used for such simulation. The difference, of course, between a multinomial and a rational form is the allowance in the latter of the division operator. It appears then that if there is any hope of producing decidable results for unbounded arenas, division must be restricted. The following theorem states that this is exactly the case.

Theorem: There is a program Q that computes the function

$$[Q]\,(P, P_F^l) = \begin{cases} T & \text{if the arena is finite} \\ 0 & \text{otherwise} \end{cases}$$

where T is a nonempty set which differentiates P from P_e^l, for all integers e, P satisfies the program restrictions above, P_F^l represents a class of constant substitutions which are domain independent and the general propagation equation for a feasible path through l involves no division operators.

The above theorems parallel the result cited earlier concerning reliable sets. Call a finite set that differentiates an infinite arena an *alternate-sufficient reliable set*. The following therefore holds.

Theorem: Alternate-sufficient reliable sets may exist for an arena, but they can be neither recognized nor generated.

The results about constant substitutions are extendible to other infinite sets. The key difference between the set of constants and another infinite alternative set (say expressions of the form $ax + b$, a and b constants) is that of difficulty of evaluation. The latter require knowledge of the program state for evaluation while the former do not. However there is a link between the two that is important. If a constant is eliminated by a test set, then all expressions which evaluate to that constant are also eliminated. This is the argument that underlies the use of fault equations mentioned earlier.

V. Related Work

The theory of fault-based testing presented here is strongly influenced by prior work in practical and theoretical testing. This section overviews how these contributions relate to the current framework. It is also hoped that the theory presented here provides insight into previous thought.

The general theory of testing in Section I is derived from [1]. The concept of a reliable test set is given there as well as a discussion of its unsolvability. The means of defining specification and correctness is taken from [13]. The approach of using failure sets to discuss essential unsolvabilities in general testing is original.

Fault-based testing is related, but not equivalent, to *error-based* testing as espoused by Weyuker and Ostrand [25]. Testing is *error-based* when it is performed with the aim of eliminating errors. In their approach, a program's input is partitioned into a set of disjoint classes called *subdomains*. These subdomains are then shown to be *revealing* for certain errors; i.e., if any member of a subdomain reveal a designated error, then all members of the subdomain reveal the error. In this manner an error is eliminated if the program is correct for an input from a subdomain that is revealing for that error. Only specified errors are eliminated.

The difficulty with Weyuker's and Ostrand's approach is that it does not specify the means of designating errors and hence it is impossible to systematically apply their ideas or to prove any properties of what they envision as error-based testing. The means of eliminating errors is therefore explained by example rather than by providing a method to follow independent of the error categories. They do indicate, however, that for every supposed error in the program, test data must be developed that eliminates the error. In order to do this a proof must be given that certain inputs eliminate particular errors. What is unclear is whether there is any general technique for doing such proofs or what exactly constitutes a proof. There is also no comment on whether such proofs are always possible, and if not, when they are possible. It is therefore impossible to determine for what categories of errors the proofs can be automated.

While the study of error-based testing is motivated by the goal of eliminating errors, the desire here to formally capture and explore this process has replaced the concept of "error" with "fault." This approach greatly benefited from two testing systems which have used the idea of alternative sets. The first such system [26] illustrates how a compiler can be used to check the adequacy of a test. The tester supplies both the program and a finite sample of expected input–output behavior. The compiler then ensures that the given behavior is not matched by any "smaller" program. Designated program expressions are substituted with a finitely many alternate expressions which are defined by built-in rules of the compiler. When a program expression is encountered for which alternatives are defined, its alternatives are evaluated and compared with the value of the original expression. Those which evaluate differently are marked. After all test data has been executed, the unmarked alternate expressions are printed by the compiler as warning messages, indicating that either the test input–output pairs are inadequate, or equivalent simpler programs have been found.

Mutation testing is the second system based on the concept of eliminating alternatives. In all fairness, adherents of mutation testing do not consider it to be fault-based testing. Elimination of faults is not their goal; they view their technique as a program exercising tool which has the agreeable side effect of ensuring that "complex errors" are not present in a program. Regardless of this orientation, it is clear that mutation testing serves well as a formal model for fault-based testing.

The concept of analyzing conditions under which faults manifest themselves is introduced in [27] and formalized in [28]. The obvious disadvantage (as evidenced in compiler-based testing) is that alternatives can be rejected which are in fact correct. Propagation is introduced here as a means of rectifying this situation. That propagation is necessary is granted by all; few, however, show any inclination to prove propagation. The assumptions of domain testing [29] typify this attitude. A predicate is considered adequately tested when inputs are found which results in states which lie "close" to the boundary described by the predicate. This is the creation condition. It is then assumed that coincidental correctness does not occur; i.e., if any input follows a wrong path it will produce a program failure. Only recently has any emphasis been given to proving propagation, and then only propagation between internal points on a path. In perturbation testing [30] the effect of computational statements on predicates is formally proved. The extension to proving that the effect persists to program output is not taken however.

Though the technique has been available for nearly two decades, the use of symbolic execution as a means of proving propagation is unique here. Limited use of symbolic execution (although apparently unrecognized by the authors) appears in [30] and [31]. The former models the computation of programs whose predicates are linear transformations of the input variables. These linear transformations are nothing more than a restricted form of symbolic execution. The latter proves propagation properties of a restricted class of recursive Lisp programs by computing the functions taken by various paths.

VI. Conclusions

Fault-based testing is a realistic approach to the inherent limitations on testing. The best that can be deduced by testing is to discern those faults that could not be present in the program, on the basis that if they were present, the test execution would have been different. To view the goal of testing as to "look for errors" is misleading in two ways. First, it implies that one can search for errors without a notion of what errors might be found. Second, such a goal is unattainable in the presence of a correct program.

A model of fault-based testing was presented based on the notion of fault-based arenas. The approach taken to the discussion of fault-based testing has been theoretically oriented. This orientation has affected the presentation in the following ways.

1) Fault-based testing has been theoretically motivated. The fact that failure sets may not be recursively enumerable leads to the conclusion that testing can only be sufficient when the failure set is restricted.

2) Many of the theorems are pessimistic. In order to describe the theoretical limitations of fault-based testing, the worst-case is often used to show some undecidable result. This does not mean that there can never be any solution—it just means that there is no general solution.

This work implies there may be a "program-verification" continuum. The defining characteristic of where a given verification technique lies is the information content of the technique. Here information content is taken to be set of faults that it demonstrates are not in a given program. On one end of the continuum is formal verification which demonstrates program correctness without executing the program. Absolute correctness can be achieved since the full semantics of the programming language is available to the technique and hence all faults can be eliminated. Fault-based testing lies in the middle due to the assumption that an alternate sufficient arena is available. Certain prescribed faults are shown to be eliminated using information derived from symbolic execution of the program. Below symbolic testing on this continuum are the various structural program coverage measures [2]. These allow deduction that certain simple faults are not in the program, such as unreachable code. At the far end of the spectrum is conventional black-box testing where input-output pairs are simply compared to a specification. The only information gained here is whether or not certain inputs are in the failure set; no extension can be made to other points in the domain.

Several important open questions remain. First, though the techniques described here are applicable to demonstrating the absence of combinations of faults, combinatorial explosion quickly overtakes the process. Therefore extrapolating results derived from the elimination of single alternatives to combinations of alternatives needs investigating. Preliminary investigations suggests that this may be a second-order effect, but more work is needed [16]. Second, domain dependent transformations are difficult to analyze because of the large number of alternate paths along which propagation can occur. Techniques for estimating the likelihood of propagation through a class of paths are currently being investigated. Finally, since software reliability [32] is intimately connected to the absence of faults, techniques that demonstrate the absence of faults should impact the assessment of reliability. The fault-based model discussed in this paper is being investigated as the basis for a white-box model of software reliability.

References

[1] W. E. Howden, "Reliability of the path analysis testing strategy," *IEEE Trans. Software Eng.*, vol. SE-2, no. 3, Sept. 1976.

[2] E. Miller, M. Paige, J. Benson, and W. Wisehart, "Structural techniques of program validation," in *Dig. Papers COMPCON 74*, Spring 1974, pp. 161-164.

[3] M. R. Woodward, D. Hedley, and M. A. Hennell, "Experience with path analysis and testing programs," *IEEE Trans. Software Eng.*, vol. SE-6,. no. 3, pp. 278-286, May 1980.

[4] G. J. Myers, *The Art of Software Testing*. New York: Wiley, 1979.

[5] E. J. Weyuker, "Axiomatizing software test data adequacy," *IEEE Trans. Software Eng.*, vol. SE-12, no. 12, pp. 1128-1138, Dec. 1986.

[6] R. DeMillo, R. J. Lipton, and F. G. Sawyer, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, pp. 34-41, Apr. 1978.

[7] S. L. Gerhard and L. Yelowitz, "Observations of fallibility in applications of modern programming methodologies," *IEEE Trans. Software Eng.*, vol. SE-2, no. 3, Sept. 1976.

[8] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Software Eng.*, vol. SE-1, no. 2, pp. 156-173, June 1975.

[9] M. Geller, "Test data as an aid in proving program correctness," *Commun. ACM*, vol. 21, pp. 368-375, May 1978.

[10] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Standard 729-1983, 1983.

[11] R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming Theory and Practice*. Reading, MA: Addison-Wesley, 1979.

[12] L. J. Morell, "A theory of error-based testing," Ph.D. dissertation, Dep. Comput. Sci., Univ. Maryland, Tech. Rep. TR-1395, Aug. 1984.

[13] H. D. Mills, "Function semantics for sequential programs," in *Proc. Inform. Processing 80*, 1980.

[14] W. S. Brainerd and L. H. Landweber, *Theory of Computation*. New York: Wiley, 1974.

[15] W. E. Howden, "Algebraic program testing," *Acta Inform.*, vol. 10, 1978.

[16] L. J. Morell, "A model for code-based testing schemes," in *Proc. Fifth Annu. Pacific Northwest Software Quality Conf.*, Oct. 1987, pp. 309-326.

[17] L. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 215-222, Sept. 1977.

[18] W. E. Howden, "Symbolic testing and the DISSECT symbolic evaluation system," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 266-278, 1977.

[19] A. J. Offutt, "The coupling effect: Fact or fiction?" in *Proc. Third Symp. Software Testing, Analysis, and Verification (TAV3)*, Dec. 13-15, 1989, pp. 131-140.

[20] J. H. Rowland and P. J. Davis, "On the use of transcendentals for program testing," *J. ACM*, vol. 28, no. 1, pp. 181-190, Jan. 1981.

[21] S. K. Basu, "A note on the synthesis of inductive assertions," *IEEE Trans. Software Eng.*, vol. SE-6, no. 1, pp. 32-39, Jan. 1980.

[22] L. J. Morell and R. G. Hamlet, "Error propagation and elimination in computer programs," Dep. Comput. Sci., Univ. Maryland, Tech. Rep. TR-1065, July 1981.

[23] M. Hecht, *Flow Analysis of Computer Programs*. New York: Elsevier North-Holland, 1977.

[24] S. Ibarra and B. Leninger, "Straight-line programs with one input variable," *SIAM J. Comput.*, Jan. 1982.

[25] E. J. Weyuker and T. J. Ostrand, "Theories of program testing and the appliction of revealing subdomains," *IEEE Trans. Software Eng.*, vol. SE-6, no. 3, pp. 236-246, May 1980.

[26] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Trans. Software Eng.*, vol. SE-3, no. 4, July 1977.

[27] K. Foster, "Error sensitive test analysis (ESTA)," in *Dig. Workshop Software Testing and Test Documentation*, Dec. 1978.

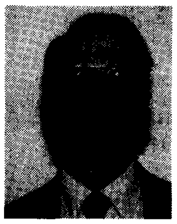
[28] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 371-379, July 1982.

[29] L. J. White and E. I. Cohen, "A domain strategy for computer program testing," *IEEE Trans. Software Eng.*, vol. SE-6, no. 3, pp. 247-257, May 1980.

[30] S. J. Zeil, "Testing for perturbation of program statements," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 335-346, May 1983.

[31] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Theoretical and empirical studies on using program mutation to test the function correctness of programs," *Principles Program. Lang.*, pp. 220-233, 1980.

[32] C. V. Ramamoorthy and F. B. Bastani, "Software reliability—Status and perspectives," *IEEE Trans. Software Eng.*, vol. SE-8, no. 4, pp. 354-371, July 1982.



Larry J. Morell received the B.A. degree in mathematics and computer science from Duke University, Durham, NC, in 1974, the M.S. degree in computer science from Rutgers University, New Brunswick, NJ, in 1976, and the Ph.D. degree from the University of Maryland in 1983 in the area of program testing.

He was an Assistant Professor at the College of William and Mary, and is now an Associate Professor at Hampton University, Hampton, VA. His current research efforts are in program testing, reliability, verification of expert systems, and specification of software tools.