A Mathematical Framework for the Investigation of Testing

JOHN S. GOURLAY, MEMBER, IEEE

Abstract—Testing has long been in need of mathematical underpinnings to explain its value as well as its limitations. This paper develops and applies a mathematical framework that 1) unifies previous work on the subject, 2) provides a mechanism for comparing the power of methods of testing programs based on the degree to which the methods approximate program verification, and 3) provides a reasonable and useful interpretation of the notion that successful tests increase one's confidence in the program's correctness.

Applications of the framework include confirmation of a number of common assumptions about practical testing methods. Among the assumptions confirmed is the need for generating tests from specifications as well as programs. On the other hand, a careful formal analysis of the usual assumptions surrounding mutation analysis shows that the "competent programmer hypothesis" does not suffice to ensure the claimed high reliability of mutation testing. Hardware testing is shown to fit into the framework as well, and a brief consideration of it shows how the practical differences between it and software testing arise.

Index Terms—Hardware testing, mutation analysis, path analysis, software reliability, software testing, specifications, testing theory.

I. Introduction

PRACTICALLY every person who has written a computer program has felt the need to test it before feeling confident that it works. Even people who have devoted their lives to the study of formal verification of programs will admit that testing is an extremely valuable practical activity. At the same time, however, very little is known about testing from a theoretical point of view. There is no agreement upon definitions of even such basic terms as "error" and "test." More complex concepts, such as the power of a particular test or of testing in general, can only be discussed by appeals to intuition. In the world of practicing programmers, we see millions of dollars spent annually on testing. It is seen as an indispensable part of computer program development. In the world of the theoretician, however, there seems to be nothing to indicate why, or even if, the money is well spent. The gulf between these two worlds represents a significant challenge to computer science.

The goal of practical program verification is to be able to prove all programs correct before they are put into use. Were this goal attained, testing would become an obsolete activity. It seems, however, to be as elusive as ever, and so we conclude that testing will be with us for some time to come. Even taking an extremely optimistic view, that verification could largely make testing obsolete in 15 or 20 years, any improvements in testing during the interim would still be quite valuable.

A general theoretical framework for testing would ideally do a number of things. Practically speaking, testing cannot hope to settle the correctness problem unequivocally (for essentially the same reason that no algorithm exists for verifying programs.) It is this limitation that has in the past relegated testing to a low position in the priorities of theoreticians. Any theory of testing, therefore, must provide a more useful criterion for judging testing than whether or not it is capable of verification. Such a criterion should, nevertheless, retain verification as an ideal limiting case, because it is the need for program correctness that motivates testing after all. The criterion for judging testing would ideally provide a way of comparing methods of testing with each other. Some approaches to testing are obviously better than others, and a useful theory of testing should confirm the obvious relationships, and provide a principle for deciding less obvious ones.

A second issue appropriately addressed by testing theory is that of program reliability. In a mathematical sense, programs are either right or wrong, and it makes little sense to talk about the reliability of a program as if it were capable of developing faults like a piece of hardware. Programs usually have undiscovered errors, however, so the idea of reliability has an intuitive appeal. It seems related to the frequent, but poorly understood, assertion that a successful test increases one's confidence in the correctness of a program. It seems reasonable to hope that a definition of the power of a method of testing might coincide with the degree of confidence imparted by a successful test.

Attempts have already been made at gaining a better theoretical understanding of testing. As we will see in Section III, however, these prior attempts have been limited in a number of ways, among them that they are conceptually different enough that they apparently cannot draw from or criticize one another.

Perhaps it goes without saying that a new theoretical framework for testing should generalize this previous work on the subject. One would also hope that an unambiguous measure of relative power of methods of testing would suggest improved methods. One might also expect that a formal approach would show that some of the intuitive conceptions about testing are wrong.

It is the purpose of this paper to describe a new theoretical framework for testing that makes progress toward these goals. Section II develops the framework, including definitions that relate verification and testing and the relative powers of methods of testing. Section III shows how previous work, both theoretical and applied, fits into this framework. Important theoretical works are shown to be special cases of the new

Manuscript received August 31, 1982; revised April 15, 1983.
The author is with the Department of Computer and Information Science, Ohio State University, Columbus, OH 43210.

0098-5589/83/1100-0686$01.00

framework (Sections A.1 through A.3), and the framework is shown to satisfy some previously articulated needs (Section A.4). Many of the common conceptions about the power of various practical testing methods are confirmed, but one, the implication of mutation testing's "competent programmer hypothesis," is shown to be false (Sections B.1 through B.3). One of the conceptions confirmed by the framework is the importance of finding greater use for specifications of programs in the generation of test data. The last part of Section III takes up this subject, looking briefly at hardware testing, an improvement on mutation testing that uses formal specifications, and a few methods for deriving test data directly from specifications.

II. Theoretical Framework

Anticipating very briefly the topic of Section III, we observe that there are three points of interest in the development of testing theory, the works of Goodenough and Gerhart, Howden, and Geller. While the major papers by these investigators appear chronologically in that order, they do not seem at first to form a logical progression. Goodenough and Gerhart propose, in effect, that methods of testing should be accepted or rejected based on whether or not they are capable of establishing program correctness. Howden quickly shows that their proposal is not a feasible one, because no computable methods of testing can establish correctness. Rather than generalizing their work, however, he abandons their framework and creates his own without exploring the relation between the two. He does, however, intimate that testing methods may be compared in terms of the degree to which they are (imperfectly) capable of program verification. Geller takes a different approach, resurrecting Goodenough and Gerhart's goal by attempting to use the results of tests to simplify correctness proofs. Again, there is no attempt to relate the proliferating theoretical frameworks.

Other problems also exist with these attempts at theory. Among them is that Geller's work, while theoretically important, is informal and difficult to generalize. Also, as we will see, Goodenough and Gerhart's mathematics is troublesome.

The purpose of the work described here, therefore, is to synthesize these three points of view. The first section defines the scope of the problem and provides a great deal of the vocabulary needed later. The second section introduces and justifies a conception of the power of testing methods that is similar to Howden's. The third section exercises some of the formal properties of the framework and is an aside with respect to the remainder of the paper.

A. Scope and Definitions

To motivate this theoretical discussion of testing let us begin with the following idealization of the programming process. First, a customer somehow specifies what is needed. Second, a programmer takes the specification and attempts to write a program to meet it. Third, the programmer or another person takes both the program and the specification, and, based on these, selects a test to be performed on the program. Finally, the test is performed and the results are compared with the specification. Failure of the test indicates that the program is wrong and needs to be rewritten. Success of the test is taken to mean that the program is ready for use.

In practice, a specification is never initially complete enough to allow a programmer to walk away with it and write a program in isolation. In fact, programmers expect to return frequently to the customer to clarify ambiguities. The fact that the specification is composed in parallel with the program is not important, however, because the end result is the same, namely that a specification is chosen and a program is written that purports to satisfy it.

Similarly, testing often goes on in parallel with programming. This also presents no conflict with the idealization, because the testing of components of the final program can be looked upon as the testing of individual programs with respect to their own separate specifications.

One important simplification of the real world does exist in this idealization, however. A typical test could comprise several runs of the program on various data. The assumption that the testing can be completed implies that each test run of the program can be completed in a reasonable amount of time. We cannot argue that there is no test in the world that does not terminate. What we must assume is that no tester will deliberately set himself an infinitely long list of tests, and that part of every specification is (perhaps implicitly) a performance criterion that will allow the tester to decide whether or not a given test has run too long and has therefore failed.

The ability to complete the testing also implies that the success or failure of a test can be decided. We will accept this, observing that (at least in the obvious case of a test being a run of the program on a particular input) it should be easier to establish the correctness of a test than of the program as a whole.

It is our purpose to concentrate on the selection of tests and on the interpretation of the results of tests. For this reason we will not worry about where programs and specifications come from. We will assume that we are simply given one of each and that the specification is correct in the sense that there is no other arbiter of the correctness of the program. The program is correct if and only if it satisfies the specification.

It is evident, then, that the theory of testing must be able to deal with the relationships between three sets of abstract objects: programs, specifications, and tests. The set of all programs will be denoted by $\mathcal{P}$, the set of all specifications by $\mathcal{S}$, and the set of all tests by $\mathcal{T}$. p and q will denote programs, elements of $\mathcal{P}$, r and s will denote specifications, elements of $\mathcal{S}$, and t and u will denote tests, elements of $\mathcal{T}$. Uppercase letters will denote subsets of $\mathcal{P}$, $\mathcal{S}$, and $\mathcal{T}$, so, for example, $p \in P \subseteq \mathcal{P}$.

The correctness of a program p with respect to a specification s will be denoted by p corr s. Clearly, corr $\subseteq \mathcal{P} \times \mathcal{S}$. We will also have use for another predicate, ok $\subseteq \mathcal{T} \times \mathcal{P} \times \mathcal{S}$. ok$(t, p, s)$ will be written as either p ok(t) s or p ok$_t$ s and will mean that test t performed on program p is judged successful by specification s.

In order for such a collection of sets and predicates to be a legitimate model of testing we need to know that if a program is correct with respect to a given specification, then no tests of that program will ever fail with respect to the specification.

In other words, we need to know that p corr $s \Rightarrow p$ ok$_t$ s. Thus, we have the following.

Definition II.A-1: A *testing system* is a collection $<\mathcal{P}, \mathcal{S}, \mathcal{T}$, corr, ok$>$ where $\mathcal{P}$, $\mathcal{S}$, and $\mathcal{T}$ are arbitrary sets, corr $\subseteq \mathcal{P} \times \mathcal{S}$, ok $\subseteq \mathcal{T} \times \mathcal{P} \times \mathcal{S}$, and $\forall p\ \forall s\ \forall t(p \text{ corr } s \Rightarrow p \text{ ok}_t\ s)$. ■

As is usual in the context of programming language semantics, the term "function" will refer to partial functions unless otherwise qualified.

Example II.A-1: A simple but abstract example will illustrate the idea of a testing system.

Let $\mathcal{P}_1$, $\mathcal{S}_1$, $\mathcal{T}_1$ all be the integers. Let corr$_1$ be equality and ok$_1(t)$ be equivalence modulo t. If p corr$_1$ s we know by definition that $p = s$. From this it certainly follows that $p \equiv s \bmod t$ and by definition p ok$_1(t)$ s. Hence this is a testing system. In this example, equivalence modulo t is serving as an imperfect test for equality. ■

Example II.A-2: For a more realistic example, pick up arbitrary set D, and let $\mathcal{P}_2$ be the set of flowchart programs that define functions on D. Let $\mathcal{S}_2$ be the set of predicate calculus formulas that define binary relations on D, and let $\mathcal{T}_2$ be the subsets of D. For a program $p \in \mathcal{P}_2$ and a specification $s \in \mathcal{S}_2$, let int(p) and int(s) denote the actual function and relation defined by them (their interpretations). Now let

$$p \text{ corr}_2\ s \Leftrightarrow \text{int}(p) \subseteq \text{int}(s)$$

$$p \text{ ok}_2(t)\ s \Leftrightarrow \text{int}(p)|_t \subseteq \text{int}(s)$$

where $f|_t$ is a function f restricted to the set t.

This is a testing system, because if p corr$_2$ s we know that int$(p) \subseteq$ int(s). Also, every restriction of int(p) is a subset of int(p). As a result, every restriction of int(p) is a subset of int(s). Hence, p corr$_2$ $s \Rightarrow p$ ok$_2(t)$ s for every t.

Three implications of these definitions are worth noting. First, the kind of correctness defined here is partial correctness. Under the usual interpretation of programs, int(p) is defined on a particular datum d if and only if p halts on d. p corr$_2$ s says, therefore, that when p halts, its input-output pair must be in s. It does not say that p must halt for any input.

Second, a specification s might be multiply defined for some $d \in D$. In other words, we might find $(d, d') \in s$ and $(d, d'') \in s$ where $d' \neq d''$. In this case, s is a "loose" specification in the sense that two programs that produce different outputs on d (namely d' and d'') can both satisfy s.

A third complementary possibility is for s to be undefined for some $d \in D$. Mathematically, we are saying that we might find that there is no d' such that $(d, d') \in s$. In this case, any program p that produces an output on d fails to satisfy s, because $(d, p(d)) \notin s$. The only programs that can satisfy this kind of specification are those that produce no output whatsoever on d, in other words programs that do not halt on input d. Of course, it is doubtful that one would ever write such a specification deliberately, but the possibility should be borne in mind. ■

Example II.A-3: Let $\mathcal{P}_3$ be the functions over D, $\mathcal{S}_3$ be the binary relations over D, and $\mathcal{T}_3$ be the subsets of D. Let

$$p \text{ corr}_3\ s \Leftrightarrow p \subseteq s \wedge \text{dom}(p) = \text{dom}(s), \text{ and}$$

$$p \text{ ok}_3(t)\ s \Leftrightarrow p|_t \subseteq s \wedge \text{dom}(p|_t) = \text{dom}(s|_t)$$

where dom(f) is the domain of f. If p corr$_3$ s, then dom(p) = dom(s), so for any t we have dom$(p|_t)$ = dom$(s|_t)$. Because we also know that $p|_t \subseteq s$ from the previous example, we conclude that p ok$_3(t)$ s.

This testing system is an abstraction of the previous one in the sense that programs have been identified with the functions they define, and specifications have been identified with the relations they define. It differs also from the previous one in that this testing system corresponds to total correctness of programs. If the specification is defined for a particular $d \in D$, then so must be the program. In other words, the program must terminate on d. The only way a program can fail to terminate for a particular d and still meet its specification is for the specification to be undefined on d. ■

Example II.A-4: Let $\mathcal{P}_4$ and $\mathcal{S}_4$ both be the set of linear functions over the integers. Let $\mathcal{T}_4$ be the set of all subsets of the integers. Let

$$p \text{ corr}_4\ s \Leftrightarrow p = s, \text{ and}$$

$$p \text{ ok}_4(t)\ s \Leftrightarrow \forall n(n \in t \Rightarrow p(n) = s(n)).$$

This is easily seen to be a special case of Example II.A-3 in which (among other things) the programs and specifications are both sets of total functions. ■

It is possible to construct complex testing systems from simpler ones in a variety of ways. Two such ways are important throughout this work, and are given in the following two theorems.

Definition II.A-2: Given a testing system $<\mathcal{P}, \mathcal{S}, \mathcal{T}$, corr, ok$>$, we will call the new system $<\mathcal{P}, \mathcal{S}, \mathcal{T}'$, corr, ok$'>$ a *set construction*, where $\mathcal{T}'$ is the set of all subsets of $\mathcal{T}$, and where

$$p \text{ ok}'_T\ s \Leftrightarrow \forall t(t \in T \Rightarrow p \text{ ok}_t\ s).$$

Recall that T denotes a subset of $\mathcal{T}$, and therefore denotes an element of $\mathcal{T}'$. ■

Theorem II.A-1: $<\mathcal{P}, \mathcal{S}, \mathcal{T}'$, corr, ok$'>$, a set construction on a testing system $<\mathcal{P}, \mathcal{S}, \mathcal{T}$, corr, ok$>$, is itself a testing system.

Proof: ok$'$ certainly has the correct domains, so all that needs to be shown is that p corr $s \Rightarrow p$ ok$'_T$ s. Suppose p corr s. By assumption, the original system is a testing system, so we know that for every t, p ok$_t$ s. Certainly, therefore, if we pick a T we know that for every $t \in T$, p ok$_t$ s. Hence, p ok$'_T$ s. ■

The interpretation of the set construction is the situation in which test consists of a number of trials of some sort, and success of the test as a whole depends on success of all the trials. This is the rule in testing practice, where the tester must run the program on a variety of data, and where failure of any one run is enough to invalidate the program. The underlying, simple testing system in this case is the one in which tests are single runs of the program.

Definition II.A-3: Given a testing system $<\mathcal{P}, \mathcal{S}, \mathcal{T}$, corr, ok$>$, we will call the new system $<\mathcal{P}, \mathcal{S}, \mathcal{T}'$, corr, ok$'>$ a *choice construction*, where $\mathcal{T}'$ is the set of subsets of $\mathcal{T}$ excluding the empty set, and

$$p \text{ ok}'_T\ s \Leftrightarrow \exists t(t \in T \wedge p \text{ ok}_t\ s).$$ ■

Theorem II.A-2: $<\mathcal{P}, \mathcal{S}, \mathcal{T}'$, corr, ok$'>$, a choice construc-

tion on a testing system $<\mathcal{P}, \mathcal{S}, \mathcal{T}, \text{corr}, \text{ok}>$, is itself a testing system.

Proof: As before, all that needs to be shown is that p corr $s \Rightarrow p$ $\text{ok}'_{\mathcal{T}}$ s. Suppose p corr s. Then, for every t, p ok_t s. Now, if we pick a $T \neq \phi$, we know that there is a t in it for which p ok_t s. Thus, we can say

$$\forall T(T \neq \phi \Rightarrow \exists t(t \in T \wedge p \text{ ok}_t s)), \text{ and}$$

$$\forall T(T \neq \phi \Rightarrow p \text{ ok}'_T s).$$ ■

The empty set must be excluded from $\mathcal{T}'$ in the choice construction, because if it were not, we would have

$$p \text{ corr } s \Rightarrow p \text{ ok}'_\phi s$$

$$\Rightarrow \exists t(t \in \phi \wedge p \text{ ok}_t s),$$

which is clearly false, violating the definition of a testing system.

The choice construction is one that also occurs regularly in practice. It models the situation in which a tester is given a number of alternative ways of testing the program, all of which are assumed to be equivalent. The tester must choose one of the alternatives, possibly at random. Since we have no way of knowing in advance which alternative will be chosen, we must assume the worst, that if the program can appear correct under any one of the alternatives, then it must be considered correct under the whole collection of alternatives. In statement testing, which will be considered in detail later, the tester is instructed to run the program a number of times in such a way that all the statements of the program will be executed at least once. For a particular statement, it does not matter which of the many inputs is chosen that causes it to be executed. It is implicit in the definition of statement testing that all of the choices are equivalent, so if the data differ somehow in their ability to detect errors, we must accept the weakest as the measure of the group.

Example II.A-5: It is possible to perform the choice construction on top of the set construction. It should be clear that Example II.A-1 already can be interpreted as a set construction. Performing a choice construction on it produces the testing system $<\mathcal{P}_1, \mathcal{S}_1, \mathcal{T}'_1, \text{corr}_1, \text{ok}'_1>$, where $\mathcal{T}'_1$ includes all the sets of subsets of D except ϕ, the set of no subsets of D. Note that $\{\phi\}$, the set containing the empty subset of D is included. ok'_1 is defined easily by

$$p \text{ ok}'_1(T) s \Leftrightarrow \exists t(t \in T \wedge p \text{ ok}_1(t) s)$$

$$\Leftrightarrow \exists t(t \in T \wedge \text{int}(p)|_t \subseteq \text{int}(s)).$$ ■

Now, with some idea of what testing systems are like, we are ready to consider the process of choosing tests. In the idealization of the programming process, the tester chooses a test based on the program and specification he is given.

From here on, we will work implicitly within the testing system $<\mathcal{P}, \mathcal{S}, \mathcal{T}, \text{corr}, \text{ok}>$, unless, of course, we explicitly mention another.

Definition II.A-4: A *test method* is a function

$$M: \mathcal{P} \times \mathcal{S} \to \mathcal{T}.$$

In other words, M takes a program and a specification and generates a test. ■

Definition II.A-5: p ok_M s will abbreviate p $\text{ok}_{M(p,s)}$ s. In other words, p ok_M s is true if and only if p ok_t s is true, where $t = M(p, s)$, the test generated by M for p and s. ■

Example II.A-6: Two possible test methods in the testing system of Example II.A-4 are

$$\text{ZERO}(p, s) = \{0\}, \text{ and}$$

$$\text{ZEROONE}(p, s) = \{0, 1\}.$$

These test methods will be used in later examples to illustrate the difference between test methods that are capable of verification and those that are not. These are both constant test methods, that is, the test chosen does not depend on the choice of p or s. p ok_{ZERO} s holds whenever $p(0) = s(0)$. p $\text{ok}_{\text{ZEROONE}}$ s holds whenever $p(0) = s(0)$ and $p(1) = s(1)$. Note that in this system $\text{ok}_{\text{ZEROONE}} = \text{corr}$, because a linear function is determined by its values at two points.

Example II.A-7: If we restrict the testing system of Example II.A-3 so that $\mathcal{P}$, $\mathcal{S}$, and $\mathcal{T}$ are the positive integers, a possible test method is max, the integer maximum function. p $\text{ok}_{\max}$ s holds whenever p is equivalent to s modulo $\max(p, s)$. Note that in this example also, $\text{ok}_{\max} = \text{corr}$, because the modulus is always chosen so that equivalence implies equality. Unlike the previous example, however, there is no single test that will always establish correctness. The success of max as a test method depends on the fact that it varies with p and s. ■

B. Power of Test Methods

Of immediate concern to a tester is what test method to use, or what test methods are better than others. An obvious and simple definition of what it means for M to be at least as good as N is that whenever N finds an error, so does M. More specifically, if a program tests incorrectly under N, it will also test incorrectly under M, with respect to the same specification. There is no particular reason why we should insist that M should use the same tests as N; we only need to know that *some* error is exposed by M whenever any error is exposed by N. This notion of power of test methods is formalized in the following definition. The rest of this section is devoted primarily to justifying the definition.

Definition II.B-1: Given two test methods M and N, we say that M has *power greater than or equal to* N (denoted $M \geqslant N$)[1] if

$$\forall p \ \forall s(p \text{ ok}_M s \Rightarrow p \text{ ok}_N s).$$ ■

While the simplicity of this definition is appealing, there are other competing notions of the power of test methods that deserve consideration. They all turn out to be equivalent to this definition, giving us additional confidence that this is the right definition. They are also important in terms of earlier work in program testing theory, and their development provides an opportunity to develop the new formalism more fully.

[1] It follows directly from the form of its definition that the power relation is a preorder and that the use of the "$\geqslant$" sign is warranted. A preorder is a reflexive and transitive binary relation; that is, $L \geqslant M \wedge M \geqslant N \Rightarrow L \geqslant N$. Note that $M \geqslant N \wedge N \geqslant M \not\Rightarrow M = N$.

We begin by formalizing the idea of reliability. Intuitively, a test method M works reliably for p and s if we can rely on the results of the test that it suggests, $t = M(p, s)$. If p is correct with respect to s, then any test will say so, hence our only interest is in the results of tests when the program is incorrect. In this case, when p is incorrect with respect to s, the results of t must demonstrate this fact, otherwise M clearly cannot be relied upon for the particular program and specification in question. What we have said, then, is that for M to be reliable for p and s, we must know that $\sim(p \text{ corr } s) \Rightarrow \sim(p \text{ ok}_M\ s)$, or in the contrapositive, $p \text{ ok}_M\ s \Rightarrow p \text{ corr } s$. Thus, we have the following.

Definition II.B-2:

$$p \text{ rel}_M\ s \Leftrightarrow (p \text{ ok}_M\ s \Rightarrow p \text{ corr } s)$$

and generalizing to sets of programs and sets of specifications,

$$P \text{ rel}_M\ s \Leftrightarrow \forall p(p \in P \Rightarrow p \text{ rel}_M\ s)$$

$$p \text{ rel}_M\ S \Leftrightarrow \forall s(s \in S \Rightarrow p \text{ rel}_M\ s)$$

and

$$P \text{ rel}_M\ S \Leftrightarrow \forall p\ \forall s(p \in P \wedge s \in S \Rightarrow p \text{ rel}_M\ s).$$

A test method M is *reliable* for a set of programs P and a set of specifications S if we can infer the correctness of any program in P with respect to any specification in S by the results of the tests given by M. ■

Example II.B-1: In any testing system and for any test method M, any set of programs P, and any set of specifications S,

$$P \text{ rel}_M\ \phi$$

$$\phi \text{ rel}_M\ S$$

and

$$\mathcal{P} \text{ rel}_M\ \mathcal{S} \Leftrightarrow \text{ok}_M = \text{corr}.$$

We can think of a test method M for which $\text{ok}_M = \text{corr}$ as one that is capable of verification. ■

Example II.B-2: Returning to the testing system of linear functions (Examples II.A-4 and II.A-6), let C_m be the set of funtions with slope m. We can observe that

$$C_m \text{ rel}_{\text{ZERO}} C_m, \text{ but}$$

$$\sim(C_m \text{ rel}_{\text{ZERO}} C_n) \quad \text{for} \quad m \neq n.$$

The former is true, because any two linear functions with the same slope that agree at zero must be identical. On the other hand, among functions of two different slopes, two can be found that intersect at zero. Since they take the same value at zero, testing both at zero gives the same result, when, in fact, they are different functions.

In the same testing system, we can also see that

$$\sim(\mathcal{P} \text{ rel}_{\text{ZERO}}\ \mathcal{S}), \text{ but}$$

$$\mathcal{P} \text{ rel}_{\text{ZEROONE}}\ \mathcal{S}.$$ ■

The relability relation is a characterization of a test method that is entirely independent of any algorithms embodied in the method, and of the actual tests selected by the method. Any two test methods that have exactly the same reliability relation are functionally identical. There is no reason to choose one over the other except, perhaps, for speed. For this reason, the relation seems to be an ideal basis for comparing the reliability or power of test methods. The most powerful test method imaginable would be one that performs reliably under all conditions, i.e., one that performs verification. Such a test method, as we have seen in Example II.B-1, also has the largest imaginable reliably tested sets, $\mathcal{P}$ and $\mathcal{S}$. This suggests that a measure of the power of a test method should be the size of the sets in its reliability relation.

Definition II.B-3:

$$\text{ms}_M(P) = \cup\{S \mid P \text{ rel}_M\ S\}$$

$$\text{mp}_M(S) = \cup\{P \mid P \text{ rel}_M\ S\}.$$

The function ms_M takes sets of programs to sets of specifications, denoting the *maximum reliably tested set* of specifications for M and P. mp_M is an analogous function from sets of specifications to sets of programs. ■

The proof that $\text{ms}_M(P)$ and $\text{mp}_M(S)$ are reliably tested sets will be postponed. Note, however, that if we know $\text{ms}_M(P)$, we know all the sets of specifications reliably tested with respect to P. By Theorem II.B-1, these are simply all the subsets of $\text{ms}_M(P)$. Then, knowledge of ms_M and mp_M captures all the information about the sizes of the reliably tested sets for M.

Thus, a reasonable alternative to our original definition of M being at least as powerful as N seems to be

$$\forall P(\text{ms}_M(P) \supseteq \text{ms}_N(P)).$$

With a bit more formalism we can show that this and several other similar formulations are equivalent to $M \geqslant N$.

Here and elsewhere we will be dealing with an arbitrary test method M in an arbitrary testing system. Just as we omit reference to the testing system, we will omit the M subscripts on relations and functions when no ambiguity results. Also, here and in subsequent theorems, proofs of the dual versions of the theorems will be omitted. We will only prove the first assertion of each theorem.

Going back to the nature of ms and mp, we have the following.

Theorem II.B-1:

$$\forall i(P \text{ rel } S_i) \Leftrightarrow P \text{ rel } \cup\{S_i\}, \text{ and}$$

$$\forall i(P_i \text{ rel } S) \Leftrightarrow \cup\{P_i\} \text{ rel } S.$$

Proof:

$$\forall i(P \text{ rel } S_i) \Leftrightarrow$$

$$\forall i\ \forall s(s \in S_i \Rightarrow P \text{ rel } s) \Leftrightarrow$$

$$\forall s(s \in \cup\{S_i\} \Rightarrow P \text{ rel } s) \Leftrightarrow$$

$$P \text{ rel } \cup\{S_i\}.$$ ■

Theorem II.B-2:

$$P \text{ rel}_M\ \text{ms}_M(P)$$

and $\text{ms}_M(P)$ is the unique largest such set of specifications.

$$\text{mp}_M(S) \text{ rel}_M\ S$$

and $\text{mp}_M(S)$ is the unique largest such set of programs.

Proof: The fact that $P \text{ rel}_M \text{ ms}_M(P)$ follows from Theorem II.B-1. The fact that it is largest follows from its construction; if $P \text{ rel}_M S$, then $S \subseteq \cup\{S \mid P \text{ rel}_M S\}$, hence $S \subseteq \text{ms}_M(P)$. ■

An alternative characterization of these maximum reliably tested sets is probably easier to comprehend.

Theorem II.B-3:

$$\text{ms}(P) = \{s \mid P \text{ rel } s\}$$

$$\text{mp}(S) = \{p \mid p \text{ rel } S\}.$$

Proof:

$$\text{ms}(P) =$$

$$\cup\{S \mid P \text{ rel } S\} \supseteq$$

$$\cup\{\{s\} \mid P \text{ rel } \{s\}\} =$$

$$\{s \mid P \text{ rel } s\}.$$

At the same time, however,

$$P \text{ rel ms}(P) \Rightarrow$$

$$\forall s(s \in \text{ms}(P) \Rightarrow P \text{ rel } s) \Rightarrow$$

$$\text{ms}(P) \subseteq \{s \mid P \text{ rel } s\}.$$ ■

Example II.B-3: For every testing system and every test method M, the following are true:

$$\text{ms}_M(\phi) = \S$$

$$\text{mp}_M(\phi) = \mathcal{P}.$$ ■

rel, ms, and mp are surprisingly interesting and complex just from a mathematical point of view. This topic will be taken up at some length in the next section, but a taste is given by the following theorem, one which is necessary for the more practical discussion here.

Theorem II.B-4:

$$\text{ms}(\cup\{P_i\}) = \cap\{\text{ms}(P_i)\}$$

$$\text{mp}(\cup\{S_i\}) = \cap\{\text{mp}(S_i)\}.$$

Proof:

$$\cap\{\text{ms}(P_i)\} =$$

$$\{s \mid \forall i(s \in \text{ms}(P_i))\} =$$

by Theorem (II.B-3)

$$\{s \mid \forall i(s \in \{r \mid P_i \text{ rel } r\})\} =$$

$$\{s \mid \forall i(P_i \text{ rel } s)\} =$$

(by Theorem II.B-1)

$$\{s \mid \cup\{P_i\} \text{ rel } s\} =$$

(by Theorem II.B-3)

$$\text{ms}(\cup\{P_i\}).$$ ■

An immediate corollary of this theorem is that $P \supseteq Q \Rightarrow \text{ms}(P) \subseteq \text{ms}(Q)$. This is because $\text{ms}(P) = \text{ms}(P \cup Q) = \text{ms}(P) \cap \text{ms}(Q) \subseteq \text{ms}(Q)$.

If we interchange the unions and intersections in the theorem, we get assertions that are not true. Ths unexpected result is taken up in the next section.

What we have conjectured is that it is the size of the ms and mp sets that define the power of test methods. More specifically, we would like to say that M being more powerful than N means that $\text{ms}_M(P)$ is bigger than $\text{ms}_N(P)$ for all P, and the dual, that $\text{mp}_M(S)$ is bigger than $\text{mp}_N(S)$ for all S. This seems reasonable, because we are insisting that for M to be more powerful than N, M must be reliable for all the programs and specifications that N is, and more, in all contexts.

It is not yet clear that power in this sense is well defined, and there are still other competing definitions of power. The following theorem deals with both these problems at the same time by showing the equivalence of several possible definitions of "M has power greater than or equal to N." The first two assertions are the dual forms of the conjecture made above. The third and fourth assertions are that power should be measured by the sizes of the maximum reliably tested sets relative to singleton sets only. These, as we will see in the next section, derive from assumptions made by Howden [12]. The last assertion, of course, is that M is at least as powerful as N in the original sense of Definition II.B-1.

Theorem II.B-5: The following are equivalent:

(1) $\forall P(\text{ms}_M(P) \supseteq \text{ms}_N(P))$

(2) $\forall S(\text{mp}_M(S) \supseteq \text{mp}_N(S))$

(3) $\forall p(\text{ms}_M(\{p\}) \supseteq \text{ms}_N(\{p\}))$

(4) $\forall s(\text{mp}_M(\{s\}) \supseteq \text{mp}_N(\{s\}))$

(5) $M \geqslant N$.

Proof: To see that (1) ⇒ (3), note that (3) is just the restriction of (1) to singleton sets of programs.

To show (3) ⇒ (1), pick any P. Then

$$\text{ms}_M(P) =$$

(by Theorem II.B-4)

$$\cap\{\text{ms}_M(\{p\}) \mid p \in P\} \supseteq$$

$$\cap\{\text{ms}_N(\{p\}) \mid p \in P\} =$$

$$\text{ms}_N(P)$$

To show (3) ⇔ (5),

$$\forall p(\text{ms}_M(\{p\}) \supseteq \text{ms}_N(\{p\})) \Leftrightarrow$$

(by Theorem II.B-3)

$$\forall p(\{s \mid p \text{ rel}_M s\} \supseteq \{s \mid p \text{ rel}_N s\}) \Leftrightarrow$$

$$\forall p \; \forall s(p \text{ rel}_N s \Rightarrow p \text{ rel}_M s) \Leftrightarrow$$

$$\forall p \; \forall s((p \text{ ok}_N s \Rightarrow p \text{ corr } s) \Rightarrow (p \text{ ok}_M s \Rightarrow p \text{ corr } s)).$$

Now, consider just the formula under the quantifiers. Because of the properties of the underlying testing system, if p corr s, then the whole formula is true. If $\sim(p \text{ corr } s)$, then $\sim(p \text{ ok}_N s) \Rightarrow \sim(p \text{ ok}_M s)$. Either case is equivalent to $p \text{ ok}_M s \Rightarrow p \text{ ok}_N s$.

The proofs that (2) ⇔ (4) ⇔ (5) are analogous. ■

It is the force of the equivalences proved above that allows us to make Definition II.B-1 with confidence.

Now we give three theorems that relate the abstract concept of power to particular properties of test methods in set and choice constructions.

Theorem II.B-6: Given a testing system $<\mathcal{P}, \mathcal{S}, \mathcal{T}'$, corr, ok'$>$ which is a set construction from $<\mathcal{P}, \mathcal{S}, \mathcal{T}$, corr, ok$>$, and given two test methods M and N in the set construction, then

$$\forall p\ \forall s(M(p,s) \supseteq N(p,s)) \Rightarrow M \geqslant N.$$

Proof: Pick p and s arbitrarily. Then

$$p\ \mathrm{ok}'_M\ s \Leftrightarrow$$

$$\forall t(t \in M(p,s) \Rightarrow p\ \mathrm{ok}_t\ s) \Rightarrow$$

$$\forall t(t \in N(p,s) \Rightarrow p\ \mathrm{ok}_t\ s) \Leftrightarrow$$

$$p\ \mathrm{ok}'_N\ s. \quad \blacksquare$$

This theorem is no surprise, saying, roughly, the more data the better. This is an intuitive idea that is confined nicely by the theory.

Example II.B-4: ZEROONE $\geqslant$ ZERO. This follows immediately from the above theorem and is consistent with our other observations about these test methods in Section I. $\blacksquare$

Theorem II.B-7: Given a testing system $<\mathcal{P}, \mathcal{S}, \mathcal{T}'$, corr, ok'$>$ which is a choice construction from $<\mathcal{P}, \mathcal{S}, \mathcal{T}$, corr, ok$>$, and given two test methods M, N: $\mathcal{P} \times \mathcal{S} \to \mathcal{T}'$ in the choice construction, then

$$\forall p\ \forall s(M(p,s) \subseteq N(p,s)) \Rightarrow M \geqslant N.$$

Proof: Pick p and s arbitrarily. Then

$$p\ \mathrm{ok}'_M\ s \Leftrightarrow$$

$$\exists t(t \in M(p,s) \wedge p\ \mathrm{ok}_t\ s) \Rightarrow$$

$$\exists t(t \in N(p,s) \wedge p\ \mathrm{ok}_t\ s) \Leftrightarrow$$

$$p\ \mathrm{ok}'_N\ s. \quad \blacksquare$$

This theorem may, at first, seem anomalous, because the test set containment seems to be going the wrong way. The theorem states that we do not lose, and perhaps gain power by reducing the size of the set of choices. This is reasonable, however, in the light of the interpretation of a set of choices. There is no way to know which choice will be taken, so we must assume the worst. Deleting choices, therefore, cannot hurt and may make things better by removing the worst choices.

Theorem II.B-8: Given a testing system $<\mathcal{P}, \mathcal{S}, \mathcal{T}''$, corr, ok''$>$ which is a choice construction from $<\mathcal{P}, \mathcal{S}, \mathcal{T}'$, corr, ok'$>$, which, in turn, is a set construction from $<\mathcal{P}, \mathcal{S}, \mathcal{T}$, corr, ok$>$, and given two test methods M, N: $\mathcal{P} \times \mathcal{S} \to \mathcal{T}''$, then

$$\forall p\ \forall s(\forall T(T \in M(p,s) \Rightarrow \exists U(U \in N(p,s) \wedge U \subseteq T))) \Rightarrow M \geqslant N$$

Proof: Pick p and s arbitrarily. Then

$$p\ \mathrm{ok}''_M\ s \Leftrightarrow$$

$$\exists T(T \in M(p,s) \wedge p\ \mathrm{ok}'_T\ s) \Leftrightarrow$$

$$\exists T(T \in M(p,s) \wedge \forall t(t \in T \Rightarrow p\ \mathrm{ok}_t\ s)).$$

Now, by assumption, we can find a $U \in N(p,s)$ such that $U \subseteq T$, so

$$\exists U(U \in N(p,s) \wedge \forall u(u \in U \Rightarrow p\ \mathrm{ok}_u\ s)) \Leftrightarrow$$

$$\exists U(U \in N(p,s) \wedge p\ \mathrm{ok}'_U\ s) \Leftrightarrow$$

$$p\ \mathrm{ok}''_N\ s. \quad \blacksquare$$

This theorem shows features of both the two preceding theorems. In the choice construction on top of the set construction, the output of a test method is a set of sets of elementary tests. For a high order test like this to succeed, at least one of the sets of tests in it, say T, must succeed. In order for T to succeed, all of the elementary tests in T must succeed. Reflecting this alternation of quantifiers, M is more powerful than N if the output of M contains fewer choices than the output of N, and if the individual choices of M contain more tests than the choices of N.

The construction described in Theorem II.B-8 will be used often enough in the next section to deserve a name and fixed notation.

Definition II.B-4: A *set-choice construction* testing system $<\mathcal{P}, \mathcal{S}, \mathcal{T}''$, corr, ok''$>$ is a choice construction on $<\mathcal{P}, \mathcal{S}, \mathcal{T}'$, corr, ok'$>$ which, in turn, is a set construction on $<\mathcal{P}, \mathcal{S}, \mathcal{T}$, corr, ok$>$. $\blacksquare$

C. Results of Theoretical Interest

This section lists a number of interesting properties of the relations between $\mathcal{P}$ and $\mathcal{S}$ induced by a test method M. It ends with another equivalent formulation of the power of test methods.

As was mentioned earlier, it is tempting to interchange the unions and intersections of Theorem II.B-4. What we get is containment in one direction, but not in the other.

Theorem II.C-1:

$$\mathrm{ms}(\cap\{P_i\}) \supseteq \cup\{\mathrm{ms}(P_i)\}$$

$$\mathrm{mp}(\cap\{S_i\}) \supseteq \cup\{\mathrm{mp}(S_i)\}.$$

Proof:

$$s \in \mathrm{ms}(P_i) \Leftrightarrow$$

$$P_i\ \mathrm{rel}\ s \Leftrightarrow$$

$$\forall p(p \in P_i \Rightarrow p\ \mathrm{rel}\ s) \Rightarrow$$

$$\forall p(p \in \cap\{P_i\} \Rightarrow p\ \mathrm{rel}\ s) \Leftrightarrow$$

$$\cap\{P_i\}\ \mathrm{rel}\ s \Leftrightarrow$$

$$s \in \mathrm{ms}(\cap\{P_i\}). \quad \blacksquare$$

The theorem should be strictly construed, because examples can be found in which the containment is proper. Example II.C-1, an extended example of many of the ideas of this section, will show that the containment can be proper even for the intersection of a descending chain.

Definition II.C-1:

$$\mathrm{xp}_M = \mathrm{mp}_M \circ \mathrm{ms}_M$$

$$\mathrm{xs}_M = \mathrm{ms}_M \circ \mathrm{mp}_M.$$

We call $xp(P)$ the *expansion of the set of programs P* and we obtain its value by first finding the maximum set of specifications related to P, $ms(P)$, and then finding the maximum set of programs related to this, $mp(ms(P))$. $xs(S)$ is defined analogously to be the *expansion of the set of specifications S.* ■

The next two theorems will justify the interpretations of these functions as expansions.

Theorem II.C-2:

$xp(P) \supseteq P$

$xs(S) \supseteq S.$

Proof:

$xp(P) =$

$mp(ms(P)) =$

$\cup\{Q \mid Q \text{ rel } ms(P)\}.$

We know that

$P \text{ rel } ms(P) \Rightarrow$

$P \subseteq \cup\{Q \mid Q \text{ rel } ms(P)\}.$

Thus

$P \subseteq xp(P).$ ■

Theorem II.C-3:

$xp \circ mp = mp$

$xs \circ ms = ms.$

Proof: By Theorem II.C-2, we know

$xp(mp(S)) \supseteq mp(S).$

On the other hand, a corollary to Theorem II.B-4 tells us that

$S \subseteq R \Rightarrow mp(S) \supseteq mp(R).$

Thus,

$S \subseteq xs(S) \Rightarrow$

$mp(S) \supseteq mp(xs(S)) \Rightarrow$

$mp(S) \supseteq mp(ms(mp(S))) \Rightarrow$

$mp(S) \supseteq xp(mp(S)).$ ■

Theorem II.C-4:

$xp \circ xp = xp$

$xs \circ xs = xs$

$ms \circ xp = ms$

$mp \circ xs = mp.$

Proof: These follow easily from Theorem II.C-3. For instance,

$xp \circ xp =$

$xp \circ mp \circ ms =$

$mp \circ ms =$

xp

and

$ms \circ xp =$

$ms \circ mp \circ ms =$

$xs \circ ms =$

$ms.$ ■

Definition II.C-2:

$\mathcal{P}^x_M = \{xp_M(P) \mid P \subseteq \mathcal{P}\}$

$\mathcal{S}^x_M = \{xs_M(S) \mid S \subseteq \mathcal{S}\}.$

These are the *sets of all expansions* of sets of programs and of specifications, respectively. ■

Theorem II.C-5:

$\mathcal{P}^x = \{mp(S) \mid S \subseteq \mathcal{S}\}$

$\mathcal{S}^x = \{ms(P) \mid P \subseteq \mathcal{P}\}.$

Proof: For any S,

$mp(S) = xp(mp(S)).$

Hence,

$mp(S) \in \mathcal{P}^x$, and

$\mathcal{P}^x \supseteq \{mp(S)\}.$

For any P,

$xp(P) \in \mathcal{P}^x$, and

$xp(P) = mp(ms(P)) \in \{mp(S)\}.$

Hence,

$\mathcal{P}^x \subseteq \{mp(S)\}.$ ■

Theorem II.C-6: Restricted to $\mathcal{P}^x$ and $\mathcal{S}^x$, mp and ms are total, one-to-one, and onto, $ms = mp^{-1}$, and xp and xs are both the identity function.

Proof: ms and mp are total and onto by Theorem II.C-5.

To show ms is one-to-one, pick $P, Q \in \mathcal{P}^x$, and suppose $ms(P) = ms(Q)$. There must be some P' such that $xp(P') = P$, and some Q' such that $xp(Q') = Q$, so we know that

$ms(xp(P')) = ms(xp(Q')) \Rightarrow$

$mp(ms(xp(P'))) = mp(ms(xp(Q'))) \Rightarrow$

$xp(xp(P')) = xp(xp(Q')) \Rightarrow$

$xp(P') = xp(Q') \Rightarrow$

$P = Q.$

To show that $ms = mp^{-1}$, pick $P \in \mathcal{P}^x$, and let $P = xp(P')$. Then,

$mp(ms(P)) =$

$xp(xp(P')) =$

$xp(P') =$

$P.$

Since $ms = mp^{-1}$, $xs = ms \circ mp = mp \circ ms = xp =$ the identity function. ■

It may be tempting, at first glance, go guess that rel is also one-to-one and onto restricted to $\mathcal{P}^x$ and $\mathcal{S}^x$. Note, however, that if P rel S, P rel R for any $R \subseteq S$. Since, in general, there will be comparable sets in $\mathcal{P}^x$ and $\mathcal{S}^x$, rel is not functional.

Theorem II.C-7: xs and xp are monotonic and $<\mathcal{P}^x, \subseteq>$ and $<\mathcal{S}^x, \subseteq>$ are complete lattices.

Proof:

$P \subseteq Q \Rightarrow$

$\text{ms}(P) \supseteq \text{ms}(Q) \Rightarrow$

$\text{mp}(\text{ms}(P)) \subseteq \text{mp}(\text{ms}(Q)) \Rightarrow$

$\text{xp}(P) \subseteq \text{xp}(Q)$.

Thus, xp is monotonic, and since its domain is a complete lattice (the subsets of $\mathcal{P}$ ordered by set containment) its range, $\mathcal{P}^x$, is also a complete lattice. ∎

It is tempting to guess that xp and xs are continuous, but this is not true, because of the inequality of Theorem II.C-1.

Example II.C-1: This example is based on a testing system $<\mathcal{P}, \mathcal{S}, \mathcal{T}, \text{corr}, \text{ok}>$ in which $\mathcal{P}$ and $\mathcal{S}$ are the linear functions over the real numbers, and tests are sets of real numbers. corr is equality of functions, and ok_t is interpreted as in a choice construction, namely, $p\ \text{ok}_t\ s \Leftrightarrow \exists x(x \in t \wedge p(x) = s(x))$. Let M be the constant method that always returns the full set of reals. It is interpreted as suggesting one test on any one of the real numbers:

$\text{ms}(P) =$

$\{s \mid P \text{ rel } s\} =$

$\{s \mid \forall p(p \in P \Rightarrow p \text{ rel } s\} =$

$\{s \mid \forall p(p \in P \Rightarrow (p \text{ ok } s \Rightarrow \text{p corr } s))\} =$

$\{s \mid \forall p(p \in P \Rightarrow (\exists x(p(x) = s(x)) \Rightarrow p = s))\} =$

$\{s \mid \forall p(p \in P \Rightarrow \forall x(p(x) \neq s(x)) \vee p = s)\}$.

Having $\forall x(p(x) \neq s(x)) \vee p = s$ hold is equivalent to saying that p and s have the same slope, or that they are parallel. We will write this as $p \parallel s$. Thus,

$\text{ms}(P) = \{s \mid \forall p(p \in P \Rightarrow p \parallel s)\}$.

Analogously,

$\text{mp}(S) = \{p \mid \forall s(s \in S \Rightarrow s \parallel p)\}$.

Note that all the functions in ms(P) must be parallel to each other and to all the functions in P. With this we can characterize the values of ms as follows. First, as with any test method,

$\text{ms}(\phi) = \mathcal{S}$.

If P is not empty and all the elements of P are parallel

$\text{ms}(P) = \{s \mid s \parallel P\}$

otherwise

$\text{ms}(P) = \phi$.

Finally,

$\text{ms}(\mathcal{P}) = \phi$.

Since the testing system is completely symmetric, mp takes exactly the same values as ms. Thus,

$\text{xp}(\phi) = \phi$.

If P is not empty and all the elements of P are parallel

$\text{xp}(P) = \{q \mid q \parallel P\}$

otherwise

$\text{xp}(P) = \mathcal{P}$.

Finally,

$\text{xp}(\mathcal{P}) = \mathcal{P}$.

If C_m denotes the set of linear functions of slope m, we have

$\mathcal{P}^x = \mathcal{S}^x = \{\phi\} \cup \{C_m \mid m \text{ is a real number}\} \cup \{\mathcal{P}\}$.

This example illustrates the inequality of Theorem II.C-1, because if $m \neq n$, then

$\text{ms}(C_m \cap C_n) = \text{ms}(\phi) = \mathcal{S}$

but

$\text{ms}(C_m) \cup \text{ms}(C_n) = C_m \cup C_n \subset \mathcal{S}$.

We can even construct a decreasing chain $P_1 \supset P_2 \supset \cdots$ for which the containment is proper. Let $f_1, f_2, \cdots$ be a series of distinct linear functions of slope 1. Let

$P_i = C_0 \cup \{f_j \mid j \geq i\}$.

Then

$\text{ms}(\cap\{P_i\}) = \text{ms}(C_0) = C_0$

but for all i

$\text{ms}(P_i) = \phi$;

hence

$\cup\{\text{ms}(P_i)\} = \phi$

and

$\text{ms}(\cap\{P_i\}) \supset \cup\{\text{ms}(P_i)\}$. ∎

$\mathcal{P}^x$ and $\mathcal{S}^x$ and the relation between them form a kind of skeleton of the original testing system, in the sense that the original system can be reconstructed from it. It is not surprising, therefore, that the power relation between test methods can be expressed entirely in terms of the expanded sets of programs and specifications. The exact expression, however, is complicated by the fact that the two test methods each induce their own distinct sets of expanded sets. One suitable formulation follows.

Theorem II.C-8:

$M \geq N \Leftrightarrow \forall P(P \in \mathcal{P}^x_N \Rightarrow \text{ms}_M(P) \supseteq \text{ms}_N(P))$.

Proof: If we assume $M \geq N$, we have

$\forall p(\text{ms}_M(P) \supseteq \text{ms}_N(P)) \Rightarrow$

$\forall p(P \in \mathcal{P}^x_N \Rightarrow \text{ms}_M(P) \supseteq \text{ms}_N(P))$.

To go the other direction, pick an arbitrary P. Then

$ms_N(P) =$

$ms_N(xp_N(P)) \subseteq$

(by the hypothesis)

$ms_M(xp_N(P)) \subseteq$

(because $xp_N(P) \supseteq P$)

$ms_M(P)$. ■

III. Reinterpretation of Previous Work

Prior work relating to the theory of testing has been inconsistent in its definitions, notation, and choice of problems. The theoretical framework of Section II developed from an attempt to synthesize the divergent work already present in the literature, and this section shows how that work can be put on a common foundation.

The topics considered are put into three categories, general theory (Section III-A), testing based solely on programs (Section III-B), and testing taking specifications into consideration (Section III-C). In the first section, the works of Goodenough and Gerhart, Howden, and Geller are shown to be special cases of the framework of Section II. The framework is shown to satisfy Weyuker and Ostrand's criticism of Goodenough and Gerhart's original mathematics, and it is compared with a similar mathematical structure developed by Kennaway and Hoare for dealing with the semantics of nondeterministic programs.

The second part of this section moves to more practical matters. It shows that most of the usual suppositions about the power of the various path testing methods and of mutation analysis are correct. It also shows that mutation analysis's "competent programmer hypothesis" is not the key to reliability that it is believed to be.

Finally, the third section takes up the topic of test data selected on the basis of specifications. Mentioned are a specification dependent variation on mutation analysis, that solves the problem just mentioned at the expense of an implemented formal specification. Also mentioned in the fact that hardware testing also fits into the theoretical framework, and that it is typically specification dependent and program independent. The section is closed by a discussion of a few long range projects to bring formal specifications into routine use in program testing.

A. Theoretical Work

1) Goodenough and Gerhart: Goodenough and Gerhart [9] were the first to grapple with the theoretical problems of program testing. Their mathematics is problematic and has received some amount of criticism. Nevertheless, their ideas are very wide ranging and have been seminal.

To their credit, they counter the argument that testing cannot show the absence of errors, and therefore is worthless. Their argument, simply stated, is that a careful study of a test method can reveal the class of errors detectable by the method. If, when applied, the method fails to detect any errors, one can conclude that all errors in the detectable class are absent.

Having thus made testing a legitimate topic of theory, they proceed through a large number of ideas about types of errors and the choice and evaluation of test data, ideas that have been echoed frequently since then.

Two specific topics of theirs form the basis of this paper. The first, a mathematical framework for the evaluation of test data, is reflected as a special case of the framework of Section II. The frameworks are superficially quite different, but the two investigations were motivated by the same problems. The new framework, in fact, originated in an attempt to correct some of the flaws in Goodenough and Gerhart's presentations. The second topic expanded upon in this paper is the importance of consulting specifications at the time of test selection. Goodenough and Gerhart argue persuasively, but informally, of its importance, and illustrate the process with an example involving a decision table specification. In this paper, the informal argument is made formal (in Section III-B). Their example is systematized and generalized well beyond decision tables elsewhere by Gourlay [8].

In their theoretical discussion, Goodenough and Gerhart assume that a program is a total function, and a specification is a binary relation with the same domain and range as the program. If we let the domain of programs be D and the range be D', then we can assume that underlying their work is a testing system in which

$$\mathcal{P} = \{p \mid p : D \to D'\}$$

$$\mathcal{S} = \{s \mid s \subseteq D \times D'\}$$

and

$$p \text{ corr } s \Leftrightarrow p \subseteq s.$$

It will develop that their testing system is a set-choice construction (see Definition II.B-4) so we will use notation consistent with that definition.

At the simplest level, a program p is tested by running it on a single point $d \in D$, and the result, $p(d)$ is checked against some specification s by seeing if $(d, p(d)) \in s$. Thus, we have

$$\mathcal{T} = D$$

$$p \text{ ok}_t\ s \Leftrightarrow (t, p(t)) \in s.$$

This is a testing system, because

$$p \text{ corr } s \Rightarrow$$

$$p \subseteq s \Rightarrow$$

$$\forall t((t, p(t)) \in s) \Rightarrow$$

$$\forall t(p \text{ ok}_t\ s).$$

The specific goal of their theory is to be able to choose and evaluate sets of test data, so they immediately introduce an abbreviation for p ok_t s that omits reference to both p and s.

Definition III.A.1-1:

$$\text{OK}(t) \Leftrightarrow p \text{ ok}_t\ s.$$ ■

OK is implicitly parameterized by p and s, and so is really no different than ok, the primitive of the testing system. Conceptually what they have done, however, is fixed p and s once

and for all, for the entire paper. The fixed p and s are inherited by all the subsequent definitions, and so all of Goodenough and Gerhart's results are about testing a given single program against a given single specification. Their notation inability, more apparent than real, to deal with varying programs or specifications is interpreted by Weyuker and Ostrand [18] as a failure of the theoretical system, but perhaps it is fairer to regard it as a limitation of the scope of their investigation.

Goodenough and Gerhart use the word "test" to refer to a set of data. A test is "successful" if the output of the program is correct for every input in the test. In the terminology of the previous section, they have moved to the set construction $<\mathcal{P}, \mathcal{S}, \mathcal{T}'$, corr, ok$'>$. They define SUCCESSFUL to be the exact analog of OK in the basic testing system.

Definition III.A.1-2:

$$\text{SUCCESSFUL}(T) \Leftrightarrow p\ \text{ok}'_T\ s. \quad \blacksquare$$

Recall that the definition of the set construction gives us

$$\text{SUCCESSFUL}(T) \Leftrightarrow$$
$$\forall t(t \in T \Rightarrow p\ \text{ok}_t\ s) \Leftrightarrow$$
$$\forall t(t \in T \Rightarrow \text{OK}(t)).$$

The latter is the original form of the definition of SUCCESSFUL.

Goodenough and Gerhart's next step is to move to the set-choice construction $<\mathcal{P}, \mathcal{S}, \mathcal{T}''$, corr, ok$''>$. They introduce an abstract version of what they call a "test data selection criterion," simply a test method M in the set-choice system:

$$M\text{: } \mathcal{P} \times \mathcal{S} \rightarrow \mathcal{T}''.$$

(We will not be specific about the nature of a "criterion" until Definition III.A.1-6.) They introduce a predicate COMPLETE to distinguish between sets of data that are and are not in the image of M for an implicitly given p and s.

Definition III.A.1-3:

$$\text{COMPLETE}(T, M) \Leftrightarrow T \in M(p, s).$$

A set T that satisfies COMPLETE(T, M) is said to be a *complete* test set for M. ■

Their idea of testing, in its entirety, is to select a "complete" test set arbitrarily (assuming that any complete test set is as good as any other), and to run the program on each element of the test set, checking that the output is correct in each case. A set-choice construction is clearly the correct model for this.

With this groundwork laid, we can make the famous definitions of reliability and validity.

The property "reliable," not to be confused with the rel relation of Section II, means "consistent" to Goodenough and Gerhart. For a test method to be reliable, all of the "complete" test data sets must yield the same result. All of the data sets must be successful, or all must be unsuccessful.

Definition III.A.1-4:

$$\text{RELIABLE}(M) \Leftrightarrow$$
$$\forall T\ \forall U(T \in M(p, s) \wedge U \in M(p, s) \Rightarrow$$
$$(p\ \text{ok}'_T\ s \Leftrightarrow p\ \text{ok}'_U\ s)). \quad \blacksquare$$

Note that, as in the earlier definition, RELIABLE is implicitly parameterized by p and s.

This is Goodenough and Gerhart's original definition transliterated into the notation of Section II. It is easy to see that the following also apply:

$$\text{RELIABLE}(M) \Leftrightarrow$$
$$\forall T(T \in M(p, s) \Rightarrow p\ \text{ok}'_T\ s) \vee \sim(p\ \text{ok}'_T\ s) \Leftrightarrow$$
$$\forall T(T \in M(p, s) \Rightarrow p\ \text{ok}'_T\ s) \vee$$
$$\forall T(T \in M(p, s) \Rightarrow \sim(p\ \text{ok}'_T\ s)).$$

The idea behind the definition of RELIABLE is that if it holds, there is no reason to choose any one of the test data sets in $M(p, s)$ over any other They are all guaranteed to deliver exactly the same information. In Section II, we have taken a different attitude. Rather than insisting on consistency of this sort, we simply declare that the information delivered by the method is only as good as the worst possible case. Aside from being mathematically simpler, this approach has the additional advantage that it does not automatically exclude useful test methods from consideration simply because they are inconsistent.

Continuing Goodenough and Gerhart's train of thought nevertheless, we find that they also consider whether or not a test method is ever able to produce a useful result, even if it is not consistent. A test method that can is called "valid."

Definition III.A.1-5:

$$\text{VALID}(M) \Leftrightarrow$$
$$\forall t(\sim(p\ \text{ok}_t\ s) \Leftrightarrow \exists T(T \in M(p, s) \wedge \sim(p\ \text{ok}'_T\ s))). \quad \blacksquare$$

Again, this is a transliteration of Goodenough and Gerhart's original definition. Alternative formulations are:

$$\text{VALID}(M) \Leftrightarrow$$
$$\sim(p\ \text{corr}\ s) \Rightarrow \exists T(T \in M(p, s) \wedge \sim(p\ \text{ok}'_T\ s)) \Leftrightarrow$$
$$\forall T(T \in M(p, s) \Rightarrow p\ \text{ok}'_T\ s) \Rightarrow p\ \text{corr}\ s.$$

In other words, if the program is incorrect, one of the choices of test data sets will show it. Equivalently, if the program tests correctly against all the choices given by M, then the program is correct.

If the choices presented by a test method M are reliable (they all show errors or they all do not) and valid (at least one choice shows an error if one is present), then certainly an error, if present, will be shown by any randomly chosen test data set from M.

Theorem III.A.1-1:

$$\text{VALID}(M) \wedge \text{RELIABLE}(M) \Leftrightarrow p\ \text{rel}_M\ s.$$

Proof:

$$\text{VALID}(M) \wedge \text{RELIABLE}(M) \Leftrightarrow$$
$$(\forall T(T \in M(p, s) \Rightarrow p\ \text{ok}'_T\ s) \Rightarrow p\ \text{corr}\ s) \wedge$$
$$(\forall T(T \in M(p, s) \Rightarrow p\ \text{ok}'_T\ s) \vee \sim(p\ \text{ok}''_M\ s)) \Leftrightarrow$$
$$(\sim\forall T(T \in M(p, s) \Rightarrow p\ \text{ok}'_T\ s) \vee p\ \text{corr}\ s) \wedge$$
$$(\forall T(T \in M(p, s) \Rightarrow p\ \text{ok}'_T\ s) \wedge \sim(p\ \text{ok}''_M\ s)) \Leftrightarrow$$

(distributing and cancelling)

$$p\ \text{corr}\ s \vee \sim(p\ \text{ok}''_M\ s) \Leftrightarrow$$
$$p\ \text{rel}_M\ s. \quad \blacksquare$$

A corollary to this theorem is Goodenough and Gerhart's original "fundamental theorem":

$$\text{VALID}(M) \wedge \text{RELIABLE}(M) \wedge p\ \text{ok}''_M\ s \Rightarrow p\ \text{corr}\ s.$$

Thus, if a test method M can be shown to be both valid and reliable for p and s, and if p tests correctly with respect to s (using M), then p is correct.

Their conclusion is that theorists should endeavor to prove various test methods valid and reliable for all programs and specifications, thus promoting them to verification methods. This approach is marred, first by the observation that no computable test methods exist that are also valid and reliable (see Section III-B.1). The all-or-nothing evaluation of test methods certainly must give way to approach which respects degrees of ability to do verification. This was the primary motivation behind the work of Section II.

A second problem calls into the question the elegance of refining the problem of test method evaluation into two subproblems, those of validity and reliability. It turns out that the two properties are not independent, and that disproving one proves the other. The following formal statement of this fact originated with Weyuker and Ostrand [18].

Theorem III.A.1-2:

$$\text{VALID}(M) \vee \text{RELIABLE}(M).$$

Proof:

$$\text{VALID}(M) \vee \text{RELIABLE}(M) \Leftrightarrow$$

$$(\forall T(T \in M(p,s) \Rightarrow p\ \text{ok}'_T\ s) \Rightarrow p\ \text{corr}\ s) \vee$$

$$(\forall T(T \in M(p,s) \Rightarrow p\ \text{ok}'_T\ s) \vee \sim(p\ \text{ok}''_M\ s)) \Leftrightarrow$$

$$\sim\forall T(T \in M(p,s) \Rightarrow p\ \text{ok}'_T\ s) \vee p\ \text{corr}\ s \vee$$

$$\forall T(T \in M(p,s) \Rightarrow p\ \text{ok}'_T\ s) \vee \sim(p\ \text{ok}''_M\ s) \Leftrightarrow$$

true. ■

Were the two properties independent, one might reasonably look for approximations to perfect test methods by looking for those that are valid but not reliable, or vice versa. Because at least one of the properties must always hold, however, there really is no middle ground, and the only interesting case is when VALID and RELIABLE both hold. In practice, then, there is only one property, VALID and RELIABLE conjoined. It is probably because of this problem that validity and reliability in Goodenough and Gerhart's sense have not been investigated further, despite the frequent citations of their work.

In the last section of their paper, Goodenough and Gerhart investigate test data selection criteria, narrowly construed, as those which specify test data sets indirectly by supplying a set of unary predicates on the input set D. A test set is "complete" if the data in the set together satisfy all the supplied predicates. The situation they are trying to model is that of a decision table specification of a program. The predicates supplied by the criterion are the conditions of the decision table. They argue that a program cannot have been "completely" tested unless all the decision table conditions have been satisfied by at least one of the data used during the testing. We will argue in Section IV that, intuitively, this is not a complete test of the program. Nevertheless, the consequence of this definition is interesting, so we will retain it for the moment.

Definition III.A.1-6: A *test data selection criterion* C is a function from programs and specifications into sets of unary predicates on the domain of programs:

$$C: \mathcal{P} \times \mathcal{S} \rightarrow \mathcal{T}''.$$

We will denote a predicate by c for "condition," rather than the equivalent symbol T, to distinguish the different roles. $t \in c$ will be written $c(t)$ for the same reason.

The test method M corresponding to a test data selection criterion C is defined by

$$M(p,s) = \{T \mid \forall c(c \in C(p,s) \Rightarrow \exists t(t \in T \wedge c(t)))\}.$$ ■

This definition is actually somewhat simpler than Goodenough and Gerhart's original one, but it captures the spirit of theirs and suffices for the following discussion. Their original definition of the test method corresponding to a criterion specifically excludes data satisfying none of the conditions. Their exclusion provides a distinction between the problems of proving reliability and validity, a technicality that we will not maintain.

Goodenough and Gerhart ask what distribution of good and bad test data corresponds to RELIABLE(M) in such a system. The answer they give turns out to be incorrect, and it can be corrected using their definitions, but a more appropriate question to answer at this point is the nearly identical one of what distribution of good and bad test data corresponds to p rel s. To do this we need one more definition.

Definition III.A.1-7: Given a test data selection criterion c, and a set of conditions $B \subseteq C(p,s)$, let

$$E(B) = \{t \mid \forall c((c \in B \Rightarrow c(t)) \wedge (c \in C(p,s) - B \Rightarrow \sim c(t)))\}.$$

$E(B)$ is the set of all inputs that satisfy all the predicates in B and no others. ■

The following theorem characterizes the reliability relation in terms of the distribution of test data in the sets given by a criterion. With minor changes, it can also suffice as a correction to Goodenough and Gerhart's analogous theorem.

Theorem III.A.1-3:

$$p\ \text{rel}_M\ s \Leftrightarrow$$

$$(\sim(p\ \text{corr}\ s) \Rightarrow$$

$$\exists c(c \in C(p,s) \wedge \forall B(c \in B \wedge B \subseteq C(p,s) \Rightarrow$$

$$\forall t(t \in E(B) \Rightarrow \sim(p\ \text{ok}_t\ s))))).$$

Proof: Assume the right-hand side is true. This can happen trivially if p corr s, in which case p rel_M s. If $\sim(p\ \text{corr}\ s)$, then the assumption forces

$$\exists c(c \in C(p,s) \wedge \forall B(c \in B \wedge B \subseteq C(p,s) \Rightarrow$$

$$\forall t(t \in E(B) \Rightarrow \sim(p\ \text{ok}_t\ s))))$$

to hold. This asserts the existence of a particular element of $C(p, s)$, say c_0, that has the given properties. Now pick any $T \in M(p,s)$. We know from Definition III.A.1-6 that

$$\forall c(c \in C(p,s) \Rightarrow \exists t(t \in T \wedge c(t))).$$

Therefore, we know that $\exists t(t \in T \wedge c_0(t))$, and we can pick a t_0 such that

$$t_0 \in T \wedge c_0(t_0).$$

Now, t_0 may satisfy predicates other than c_0, so let

$$B_0 = \{c \mid c(t_0) \wedge c \in C(p, s)\}.$$

Certainly, $B_0 \subseteq C(p, s)$, $c_0 \in B_0$, and $t_0 \in E(B_0)$. Thus, by the original assumption that the right-hand side of the theorem is true, we conclude that $\sim(p\ \mathrm{ok}(t_0)\ s)$. Since $t_0 \in T$, we also have $\sim(p\ \mathrm{ok}'_T\ s)$, and since T was arbitrarily chosen, we have $\sim(p\ \mathrm{ok}''_M\ s)$. Thus $\sim(p\ \mathrm{corr}\ s) \Rightarrow \sim(p\ \mathrm{ok}''_M\ s)$, or $p\ \mathrm{rel}_M\ s$, and the left direction ($\Leftarrow$) of the theorem is proved.

Assume now that the right-hand side of the theorem if false. This means that $\sim(p\ \mathrm{corr}\ s)$ and

$$\forall c(c \in C(p, s) \Rightarrow \exists B(c \in B \wedge B \subseteq C(p, s) \wedge \exists t(t \in E(B) \wedge p\ \mathrm{ok}_t\ s))).$$

This says that for every c we can pick a set of predicates B_c and a piece of data t_c, with the properties $B_c \subseteq C(p, s)$, $c \in B_c$, $t_c \in E(B_c)$, and $p\ \mathrm{ok}(t_c)\ s$. The definition of E tells us that $c(t_c)$ for every c. For this reason, if we let

$$T = \{t_c \mid c \in C(p, s)\}$$

we know that $T \in M(p, s)$. Since we also know that $p\ \mathrm{ok}(t_c)\ s$ for every c, it follows that $p\ \mathrm{ok}'_T\ s$ and that $p\ \mathrm{ok}''_M\ s$. Finally, $p\ \mathrm{ok}''_M\ s$ and $\sim(p\ \mathrm{corr}\ s)$ mean that $\sim(p\ \mathrm{rel}_M\ s)$, and the right-hand direction ($\Rightarrow$) of the theorem is proved. ■

What this theorem says is that a necessary and sufficient condition for reliability is the existence of a c_0 such that every B containing it has $E(B)$ consisting solely of failed tests (t's such that $\sim(p\ \mathrm{ok}_t\ s)$). If this were true, we would define a new test data selection criterion that supplied only the single condition c_0. The test method corresponding to this criterion would reliably test p and s, because $E(\{c_0\})$ under the new criterion would be the union of all the $E(B)$ from the original criterion where B contains c_0. The new method would also be much simpler than the original one, because it would never prescribe more than one piece of test data. Anticipating the next section, however, we cannot hope to algorithmically discover such a c_0 without finding a solution to the program equivalence problem. Thus, what in principle is a powerful test simplification theorem turns out to be impractical.

2) Howden: Two rather distinct goals are evident in the work of Howden, both following directly in the footsteps of Goodenough and Gerhart. First, he seeks ways of describing and evaluating the effectiveness of test methods, and second, he seeks ever better methods of selecting test data, with the emphasis on consulting the specification in the selection process.

In "Reliability of the Path Analysis Testing Strategy" [12], a paper that typifies the first goal, he introduces the term "reliable" in the sense of Section II. His underlying testing system is a set construction in which $\mathcal{P}$ is the set of programs in some unspecified programming language, and $\mathcal{S}$ is the set of computable functions. A test T is reliable for p and s if $p\ \mathrm{ok}'_T\ s \Rightarrow p\ \mathrm{corr}\ s$. The definition of the rel_M relation of Section II is a conscious generalization of this idea, from individual tests to test methods, and from individual programs and specifications to sets of programs and specifications. As we saw in the previous section, reliability in Howden's sense and ours is the conjunction of reliability and validity in Goodenough and Gerhart's sense. Howden does not state this explicitly, but he does admit his debt to Goodenough and Gerhart for the term "reliable."

Howden discusses Goodenough and Gerhart's implicit goal, which is to find a test method that satisfies both VALID and RELIABLE for all programs and specifications. In the terminology of Section II, the problem is to find an M such that $\mathcal{P}\ \mathrm{rel}_M\ \mathcal{S}$, a universally reliable test method. Such an M exists in principle, because for any p and s such that $\sim(p\ \mathrm{corr}\ s)$, we can let $M(p, s)$ take a value t where $\sim(p\ \mathrm{ok}_t\ s)$, and if $p\ \mathrm{corr}\ s$, we can let $M(p, s)$ take any value whatsoever. The property that every failure of correctness can be exposed by some test is not inherent in the definition of a testing system. It is, however, an appropriate one for practical purposes, because an undetectable error may as well be disregarded.

Although universally reliable test methods exist in a mathematical sense, they cannot be constructed in testing systems of reasonable complexity. A natural testing system is one in which $\mathcal{P}$ and $\mathcal{S}$ are both the set of all programs in some general purpose programming language. Correctness is program equivalence. The standard assumption of testing theory (and practice) is that $p\ \mathrm{ok}_t\ s$ is decidable. Given this, a universally reliable M must not be computable, because an algorithm to decide $p\ \mathrm{ok}_M\ s$ for every p and s would solve the program equivalence problem.

Howden's answer to the futility of finding a universally reliable test method is to try to judge the extent to which admittedly imperfect test methods are reliable. This he proposes to do by discovering the classes of programs for which a particular test method is reliable. This is precisely the problem of finding P for which $P\ \mathrm{rel}_M\ s$ for some fixed s. Implicit in his discussion is the assumption that the bigger P can be the stronger M is. This is made precise and shown to be justified in Theorem II.B-5.

The remainder of the paper deals with path testing specifically, which will be discussed in a later section.

In another paper, "Algebraic Program Testing" [11], Howden pursues the goal of generating test data from a knowledge of the program's specification. Here, he generalizes his concept of reliability almost to the fully general definition of Section II. Working in a set-choice testing system in which $\mathcal{P} = \mathcal{S}$ = the numeric functions, he describes a number of restricted classes of functions for which reliable finite test methods exist. In this work he generalizes reliability one step further, to be able to talk about F, a set of functions, and M, such that $F\ \mathrm{rel}_M\ F$. An example of the spirit, but not the complexity, of his results is that $F\ \mathrm{rel}_M\ F$, where F is the set of polynomials and

$$M(p, s) = \{T \mid T \text{ contains } 1 + \max(\deg(p), \deg(s)) \text{ points}\}.$$

In other words, if we can ascertain that both the program and specification are polynomials of degree less than or equal to n, then they can be reliably tested by comparing them on any set of $n + 1$ points.

3) Geller: A natural next step from Howden's work is Geller's work on proving programs correct with the aid of test data [7]. As Howden shows, testing cannot be expected to do the whole job for all programs and specifications. Geller suggests, however, that testing and proving may complement each other, and specifically, that the results of some carefully chosen tests might simplify a program's correctness proof. Lamport

[15] argues that the amount of work saved is insignificant, but both Geller and Lamport come to their conclusions on the basis of a single example, so the jury is still out.

The principle behind Geller's ideas is this. Given the knowledge that P rel_M S for some test method M, we can prove a program p correct with respect to a specification s in two steps. First, prove that $p \in P$ and $s \in S$, and second, by actually running the program, show that p ok_M s.

The knowledge that P rel_M S is part of what Geller calls the "generalization assertion." While Geller proves all his generalization assertions on the spot, he clearly intends that a collection of assertions of the form P rel_M S be available to practitioners. These could be gleaned from the sort of work done by Howden on reliably testable sets of functions. The rest of the generalization assertion is $p \in P$ and $s \in S$. Because of the presence of p and s in these assertions, these must be proved anew with each application of the method. Conceptually, these proofs demonstrate that a certain property holds for the given program and specification. Geller's own examples include such properties as linearity and the absence of loops.

The knowledge that p ok_M s is called the "test data assertion," and the desired consequence, that p corr s, is called the "synthesized assertion." The reason that the latter is not simply called "correctness" is that Geller proves his programs section by section, using the synthesized assertions of the sections as ordinary assertions in a Floyd-style correctness proof of the whole program.

Geller's method, although stated quite informally, is significant, because it makes the final step in generalizing Howden's reliability to classes of specifications drawn from a universe $\mathcal{S}$, distinct from $\mathcal{P}$. Geller's specifications are formulas in predicate calculus, so he has stepped beyond Howden's theorems of the form F rel_M F.

4) Weyuker and Ostrand: Weyuker and Ostrand [18] have made the most thorough criticism of the work of Goodenough and Gerhart. The testing theory of Section II was not designed to answer Weyuker and Ostrand's criticisms explicitly. Rather, it was intended to unify the ideas of Goodenough and Gerhart, Howden, and Geller. The important features of the theoretical framework were already in place by the time Weyuker and Ostrand's paper appeared, so it is interesting to see it fares well under Weyuker and Ostrand's criticism.

Of the four failings specifically identified by Weyuker and Ostrand, the most evident, perhaps, is that the definitions of validity and reliability are relative to a single program and a single specification. As we have seen, a careful study of Goodenough and Gerhart's work shows that its real failing along these lines is that it deals with all programs and all specifications at once. The theory of Section II follows Howden in providing a way to talk about important classes of programs and specifications, thus answering Weyuker and Ostrand's criticism directly.

A second criticism is really a symptom of the first, that validity and reliability are not preserved throughout the debugging process. Weyuker and Ostrand contend that the validity and reliability of the test method in use must be reestablished every time the program under development is changed. In the theory of Section II, a test method is characterized by its rel relation, which is independent of the given specification and program. In principle, at least, the applicability of a test method is captured once and for all in its rel relation and need not be computed as the program changes during debugging. In practice the rel relation may not be fully known for most test methods, so knowledge of it may have to be extended as debugging proceeds. It is a possibility, however, that the changing program will remain in a known reliably tested set of programs, mitigating the problem of incomplete knowledge of rel.

Weyuker and Ostrand's third criticism of Goodenough and Gerhart's work is the lack of independence of validity and reliability. This has been dealt with in Section II (and by Howden) by conjoining the two properties and dealing strictly with the new composite property. To the extent that the separate properties have intuitive appeal, this approach is inadequate. Mathematically, however, it works effectively.

Weyuker and Ostrand's fourth criticism is one that ties directly to their own proposal for improving Goodenough and Gerhart's work. They feel that validity and reliability should be generalized to apply to subsets of the programs' input domain. This is a tangent to the reasoning of this paper and other testing research, but it would appear that Goodenough and Gerhart's definitions applied to a program and specification restricted to some subdomain would suffer from the same faults as in the full domain. Weyuker and Ostrand's pursuit of this idea is brief and does not deal with this question.

Thus the theoretical framework of Section II solves three of the four problems raised by Weyuker and Ostrand.

5) Kennaway and Hoare: The work by Kennaway and Hoare [14] on the semantics of nondeterministic computer programs does not deal with the testing of programs explicitly, but it is intriguing because it describes a mathematical structure that is almost identical to the one in Section II. The notation they develop is consistent within itself, but it overlaps the notation of Section II in ways that would make its use here very confusing. What we will do, therefore, is use our notation, but reinterpret it in their way. The differences between the developments of the two theories raise some interesting questions about the testing theory.

The primitive objects in Kennaway and Hoare's discussion are the "deterministic machines" $\mathcal{P}$, and the "tests" $\mathcal{S}$. Each machine $p \in \mathcal{P}$ may or may not "pass" a test $s \in \mathcal{S}$, and the relation rel $\subseteq \mathcal{P} \times \mathcal{S}$ tells which machines pass which tests. For Kennaway and Hoare the relation rel is arbitrary and there are no analogs to $\mathcal{T}$ and M. In the testing theory, on the other hand, rel is defined in terms of the more primitive ok and corr. This raises the first question about the theory of testing, does the derived nature of rel allow any relation whatsoever between $\mathcal{P}$ and $\mathcal{S}$? The answer, one can quickly see, is yes, because p rel_M s is defined to be p ok_M $s \Rightarrow p$ corr s. If we choose ok so that p ok_t s is true for all p, s, and t, then p ok_M s always holds and p rel_M s is equivalent to p corr s. We also need to know that p corr $s \Rightarrow p$ ok_t s, but since the right-hand side is always true, corr and therefore rel_M may be any arbitrary relation.

Kennaway and Hoare then proceed to define "nondeterministic machines" to be sets of deterministic machines and "specifications" to be sets of tests. A nondeterministic machine P "satisfies" a specification S (written P rel S) if

$$\forall p \; \forall s (p \in P \wedge s \in S \Rightarrow p \text{ rel } s).$$

Switching again to the testing theory, we see that this is identical to the definition of reliability for sets of programs and sets of specifications. The question raised here is whether or not there is an interpretation for sets of programs and sets of specifications other than simply as sets. This is a somewhat philosophical question that cannot be answered absolutely, but it is reasonable to say no. The reason comes from one of the goals of the testing theory, going back to Howden. In order to know when we can effectively use testing, we need to know on which sets of programs our method is reliable. It does not seem reasonable to interpret sets of programs in any other light. The idea could bear some fruit in an indirect way, however. A set of specifications can be regarded as a specification as well, but a specification of a different sort. A program could be said to satisfy one of these "abstract" specifications if it satisfies one of the "concrete" specifications within it. One could analogously define abstractions of programs as sets of programs, but both of these ventures into power sets must be done in addition to the construction of the power sets for the definition of reliability. What practical consequences such definitions might have is an open question.

Kennaway and Hoare next define the functions ms, mp, xs, and xp in exactly the same way as they are defined in the testing theory. mp(S), for instance, is the nondeterministic machine composed of all deterministic machines that pass all of the tests in S. They state many properties of these functions, including making the definitions of $\mathcal{P}^x$ and $\mathcal{S}^x$. They go beyond the presentation in Section II, however, when they conclude that ms and mp are monotonic and that $\mathcal{P}^x$ and $\mathcal{S}^x$ are isomorphic. This is accomplished by turning the lattice of nondeterministic machines upside down (subsets of $\mathcal{P}$ ordered by superset), using the argument that the more deterministic machines in a nondeterministic machine, the less predictable it is, and the lower it should be in an ordering of nondeterministic machines. In the previous paragraph we argued that it does not seem reasonable to interpret a set of programs as anything else, so we certainly cannot insist that sets of programs be ordered differently from sets of specifications. Were we to play the formal game, however, the monotonicity and isomorphism would apply in the testing theory as well.

At this point, Kennaway and Hoare's theory diverges rather substantially from the testing theory. They undertake the usual program of denotational semantics by trying to find continuous operators for constructing complex nondeterministic machines from simpler ones. Two assumptions they make along the way are nevertheless of some interest.

In order to know that xp and xs are continuous functions, they make an assumption analogous to the following: for every sequence $P_0 \supseteq P_1 \supseteq P_2 \supseteq \cdots$ of sets in $\mathcal{P}^x$ assume that

$$\cap\{P_i\} \text{ rel } s \Rightarrow \exists j(P_j \text{ rel } s).$$

What bearing such an assumption might have on the computability or finiteness of the underlying test method is an open question.

A final observation is that Kennaway and Hoare find it appropriate to assume that there are no "miraculous" machines, no machines that pass all tests. This is equivalent to assuming that $\text{mp}(\S) = \phi$, something not at all appropriate for the testing theory. In a testing system reflecting partial correctness, for instance, a program that never halts for any input will vacuously satisfy all specifications, and there is nothing paradoxical or miraculous about it.

B. Specification Independent Testing

1) Limitation: We have seen that from the theoretical point of view, a test method is a function that depends on both programs and specifications. In order to be able to describe an algorithm for a test method in general requires that both programs and specifications be formal objects suitable for use as data in the computation that yields a test. Ever since the advent of compilers, programs have been suitable formal objects, but specifications are still rarely formal. For this reason, the most discussed and best understood test methods do not depend on specifications.

The restriction of test methods to depend on programs alone is a technical convenience only, so it comes as no surprise that the power of this class of test methods is also restricted. The usual argument for this is a hand wave called the "missing path problem": if the given program fails to perform an action required by the specification, then it may be that no amount of analysis of the program can produce test data that reveal the action's absence. Only by consulting the specification as well as the program can this be done. The loopholes in this reasoning are many, including the lack of definition of the term "action," how one is specified and how one can be missing. Nevertheless, the idea is clear and the missing path problem has driven all research to date on testing based on specifications.

One of the encouraging things about the theoretical framework described in Section II is that it provides a precise theoretical version of the missing path problem.

A test method that depends only on programs is independent of specifications in the mathematical sense.

Definition III.B.1-1: A test method M is *specification independent* if

$$\forall p\ \forall s\ \forall r(M(p, s) = M(p, r)).$$

We will abbreviate $M(p, s)$ by $M(p)$ when M is specification independent. ■

The fact that a test method is specification independent does not in and of itself limit the power of the method. A constant test method is certainly specification independent and we have already seen a testing system with a constant, universally reliable test method (Example II.A-6). If, on the other hand, we make a reasonable assumption about the generality of the underlying testing system, we find that specification independence does imply a limit on power. The necessary assumption is that for every program and test, we can always find a specification that will fool the test. In other words, we can always find a specification against which the program tests correctly, but with respect to which the program is incorrect. Among actual programs in any general purpose language, and among specifications robust enough to specify all programs, this is clearly true.

Theorem III.B.1-1: For a specification independent test method M in a testing system that has the property that

$$\forall p\ \forall t\ \exists s(p \text{ ok}_t\ s \wedge \sim(p \text{ corr } s))$$

$\forall p\ \forall s(p \in P \wedge s \in S \Rightarrow p \text{ rel } s)$.

we have

$\text{mp}_M(\delta) = \phi$.

Proof:

$\text{mp}_M(\delta) =$

$\{p \mid p \text{ rel}_M\ \delta\} =$

$\{p \mid \forall s(p \text{ rel}_M\ s)\} =$

$\{p \mid \forall s(p \text{ ok}_M\ s \Rightarrow p \text{ corr } s)\} =$

$\{p \mid \forall s(\sim(p \text{ ok}_M\ s) \vee p \text{ corr } s)\} =$

$\{p \mid \sim\exists s(p \text{ ok}_M\ s \wedge \sim(p \text{ corr } s))\}$

Now, suppose this set is not empty, and pick a program p_0 from it. Because M is specification independent, we can let $t_0 = M(p_0)$, knowing that $\forall s(t_0 = M(p_0, s))$. Thus, we can see that

$\sim\exists s(p_0 \text{ ok}_{t_0}\ s \wedge \sim(p_0 \text{ corr } s))$.

Following from this we have

$\exists p\ \exists t \sim\exists s(p \text{ ok}_t\ s \wedge \sim(p \text{ corr } s)) \Leftrightarrow$

$\sim\forall p\ \forall t\ \exists s(p \text{ ok}_t\ s \wedge \sim(p \text{ corr } s))$

a direct contradiction to the initial assumption about the testing system. The conclusion, therefore, is that $\text{mp}_M(\delta)$ is empty. ■

This is not to say that specification independent testing is worthless. $\text{mp}_M(S)$ for some S smaller than δ may well be nonempty. It does suggest, however, that specification dependent testing can be a worthwhile goal.

2) Path Testing: Specification independent testing first came into practical use in the form of "statement testing." Statement testing is based on the observation that an error in a program statement cannot possibly be detected unless the statement is executed. A goal of testing, therefore, should be to run the program on a diverse enough collection of data so that every statement of the program is executed at least once. This was never thought to be a fully reliable test method, but it was seen as a minimally acceptable amount of testing.

It was quickly recognized that many errors go undetected under a statement testing regime. A particular kind of error missed is akin to the missing path problem: there is nothing in statement testing to force the program to follow an empty branch. If the program contains an empty branch that should have (according to some unwritten specification) done something specific, statement testing will not systematically detect it. This led to the introduction of what we will call "branch testing," which in the literature is often called "path testing." In branch testing, the program is not minimally tested until every branch (or arc) in the program's flow graph has been traversed at least once. This corrects the immediate problem by fixing attention on the branches rather than the statements (or nodes) of the flow graph. It is also intuitively clear that branch testing should be more powerful than statement testing, because any set of data that satisfies the branch testing rule will also satisfy the statement testing rule.

Just like statement testing, branch testing fails to detect certain kinds of errors, and proposals have been made to improve on branch testing. It is beyond the scope of this section to itemize these. What we will do is illustrate the use of the theoretical framework of Section II to formalize the relationship between statement and branch testing. In the process, we will find that statement testing and branch testing are just the first two of an infinite collection of specification independent test methods which we will call the "path testing" methods.

Huang [13] and Howden [12] deal extensively with the topic of branch testing, so we will move quickly and try not to repeat large amounts of their discussion. We will let the set of programs $\mathcal{P}$ be the set of finite digraphs with distinguished start and stop nodes, and annotated with a set of programming language constructs (which we will not define). Since we are dealing with specification independent testing, the set of specifications δ will remain totally undefined.

Definition III.B.2-1: We denote a path through a program flow graph by w, the sequence of nodes (statements) visited by the path. The length of a path is just the length of the sequence. $\text{paths}_n(p)$ denotes the set of all paths in p of length less than or equal to n. ■

Note that $\text{paths}_1(p)$ is the set of all nodes of p. $\text{paths}_2(p)$ is the set of all nodes together with the set of all arcs (or branches) of p.

Definition III.B.2-2: $\text{dom}(p, w)$ is the set of all inputs to p that cause it to execute along a path of which w is a subpath. ■

Statement testing provides a rule under which certain sets of test data are considered acceptable and others are not. Among the acceptable sets of test data, statement testing makes no distinction, and within a set all the tests must be passed. Again we have a set-choice construction (see Definition II.B-4) based on $\mathcal{T}$, the input domain of programs. The formal definition of statement testing is now easy.

Definition III.B.2-3:

$\text{STATEMENT}(p) =$

$\{T \mid \forall w(w \in \text{paths}_1(p) \wedge \text{dom}(p, w) \neq \phi \Rightarrow \text{dom}(p, w) \cap T \neq \phi)\}$.

$\text{STATEMENT}(p)$ consists of all sets of inputs that include data in the domain of every reachable statement in the program. ■

The definition of branch testing is analogous, and we see that statement testing and branch testing are just two special cases of the following more general definition.

Definition III.B.2-4:

$\text{PATH}_n(p) =$

$\{T \mid \forall w(w \in \text{paths}_n(p) \wedge \text{dom}(p, w) \neq \phi \Rightarrow \text{dom}(p, w) \cap T \neq \phi)\}$

$\text{BRANCH}(p) = \text{PATH}_2(p)$. ■

Note that $\text{STATEMENT}(p) = \text{PATH}_1(p)$.

The PATH_n testing methods for $n \geq 3$ have not been described in the literature, but at the same time they are not unwanted or frivolous. Specifically, Holthouse and Hatch [10] observe that branch testing does not force a while loop to be tested for the case in which it does not loop. In their experience, a significant number of additional errors could be detected, were

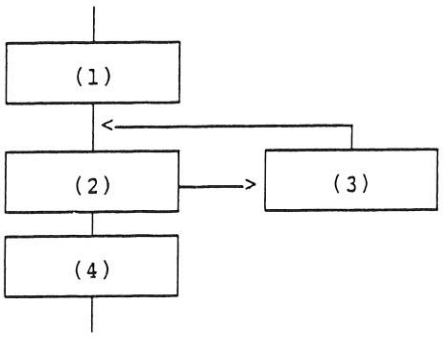

Fig. 1. Program requiring PATH_3.

branch testing augmented to force the execution of such special cases. PATH_3 does exactly what they want, because it forces the execution of statement triples, distinguishing (see Fig. 1) the path $<1, 2, 4>$ from $<3, 2, 4>$.

With the definition of PATH_n, the assertion that BRANCH > STATEMENT becomes a corollary of the following theorem.

Theorem III.B.2-1: For every $n \geqslant 0$,

$$\text{PATH}_{n+1}(p) > \text{PATH}_n(p).$$

Proof: Pick a $T \in \text{PATH}_{n+1}(p)$. Then

$$\forall w(w \in \text{paths}_{n+1}(p) \wedge \text{dom}(w, p) = \phi \Rightarrow \text{dom}(w, p) \cap T \neq \phi).$$

Since $\text{paths}_n(p) \subseteq \text{paths}_{n+1}(p)$ we have

$$\forall w(w \in \text{paths}_n(p) \wedge \text{dom}(w, p) \neq \phi \Rightarrow \text{dom}(w, p) \cap T \neq \phi)$$

or

$$T \in \text{PATH}_n(p).$$

Thus, $\text{PATH}_{n+1}(p) \subseteq \text{PATH}_n(p)$, and by Theorem II.B-7,

$$\text{PATH}_{n+1} \geqslant \text{PATH}_n.$$

A proof that the power relation is strict requires a proof that

$$\sim(\text{PATH}_n \geqslant \text{PATH}_{n+1}) \Leftrightarrow$$

$$\sim(\forall p\ \forall s(p\ \text{ok}''_{\text{PATH}(n)}\ s \Rightarrow p\ \text{ok}''_{\text{PATH}(n+1)}\ s)) \Leftrightarrow$$

$$\exists p\ \exists s(p\ \text{ok}''_{\text{PATH}(n)}\ s \wedge \sim(p\ \text{ok}''_{\text{PATH}(n+1)}\ s)).$$

In other words, we need to find an example of p and s that test correctly under PATH_n but incorrectly under PATH_{n+1}. More specifically, recalling the set-choice construction underlying the various PATH_n, we need to find a program p that runs correctly on at least one of the sets in $\text{PATH}_n(p)$, but on none of the sets in $\text{PATH}_{n+1}(p)$. To be rigorous we would need to precisely define the set of specifications, the statements and conditions of the programming language, and the semantics of both. Instead, suffice it to say that a program can easily be constructed in which the conditional execution of one statement has the side effect of causing a detectable failure in another statement n steps further along in the execution of the program. In Fig. 2, the execution of s_1 will cause statement s_{n+1} to attempt to divide by zero. In PATH_{n+1} of this program every T will have to contain something in $\text{dom}(s_1, \cdots, s_{n+1})$. It thus will detect the error. In PATH_n, however, the requirement is only that $T \in \text{PATH}_n(p)$ contains an element of each

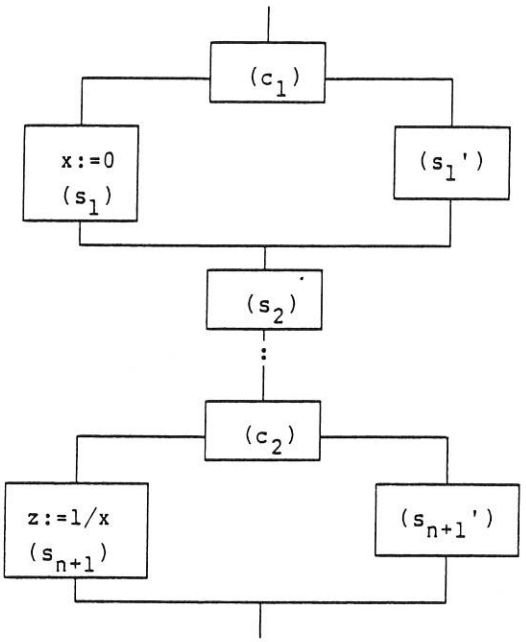

Fig. 2. Program requiring PATH_{n+1}.

$\text{dom}(w)$ where w is of length n. Both $\text{dom}(s_1, s_2, \cdots, c_2)$ and $\text{dom}(s_2, \cdots, c_2, s_{n+1})$ contain data that bypass the deadly combination. Thus, there are elements of $\text{PATH}_n(p)$ that do not detect the error. ■

Some people involved in path testing might argue that our definition of PATH_n is too simplistic, because the sets of data that interest a path tester are really just the minimal sets that exercise all paths. Our definitions could be altered to reflect this difference, with exactly the same consequence. The proof of the preceding theorem would become a bit more complicated and would require an application of Theorem II.B-8 rather than Theorem II.B-7, because $\text{PATH}_{n+1}(p)$ would no longer be comparable, setwise, to $\text{PATH}_n(p)$.

3) Mutation Testing: Budd *et al.* [1] have described another specification independent approach to testing that they call "mutation analysis." Superficially quite different from path testing, mutation analysis turns out to fit nicely into the same framework. This will be discussed briefly and will be followed by some observations about the "competent programmer hypothesis." Contrary to the claims of the proponents of mutation analysis, we will see that the hypothesis is not enough to ensure the reliability of mutation analysis.

Like path testing, mutation analysis provides a criterion for judging the adequacy of a set of inputs for testing a given program. The program in question is modified or "mutated" in a variety of ways. The resulting battery of mutant programs is run on the candidate test data set, and the results are compared with the output of the original program on the same data. The candidate test data set is judged adequate for testing the original program if every mutant is distinguishable from the original program. Of course, just as some paths are infeasible in path testing, some mutants are equivalent to the original program. Detecting both infeasible paths and equivalent mutants can be difficult, but like all other investigators, we assume they can be recognized and omitted from consideration.

From this description it is clear that mutation analysis can

be based on the same set-choice construction as path testing. The exact power of mutation analysis, however, depends entirely on the way the mutants are generated. In their paper, Budd *et al.* describe the generation of mutants in terms of the repeated application of a small number of "mutant operators."

One mutant operator, for instance, replaces any statement with a special statement that causes a distinctive failure whenever it is executed. When this mutant operator is applied in all possible ways to obtain one mutant for every statement in the original program. Data to distinguish all these mutants from the original program must collectively execute all statements of the original program. Another pair of mutant operators replace logical expressions by the constants true or false. Again, data to distinguish all possible mutants obtained from these operators must collectively force the original program to follow all its branches.

What we have just shown, as did Budd, is that mutation analysis taken as a test method is at least as powerful as PATH_2 or branch testing. That it is strictly more powerful is an easy matter of finding examples in analogy to the previous theorem. The relation of mutation analysis to higher path testing methods is less clear, but it would appear that there is no way to use the mutant operators informally described by Budd to force the execution of arbitrary paths of length three or more. Thus, we have the following.

Theorem III.B.3-1: Mutation analysis is strictly more powerful than PATH_2 and is incomparable to PATH_n for any higher n.

Proof: Given informally above. ∎

The proponents of mutation analysis are not so much interested in the relation between mutation analysis and path testing as they are in the probability of success of their method. It is clear that the program written in response to a given specification is not chosen at random. In the vast majority of cases (we are ignoring the case of a malevolent programmer deliberately writing a "Trojan Horse") the program is the result of a conscientious effort to meet the specification, and, as a result is "almost correct." The competent programmer hypothesis is an attempt at formalizing this idea with the end of showing that mutation analysis is almost always reliable.

The competent programmer hypothesis as applied to mutation testing is the following. Given a program p written in response to specification s, it is very likely that either p corr s or q corr s for some q, a mutant of p. Rather than arguing, for any particular set of mutant operators, whether or not the competent programmer hypothesis holds, we will assume that it does hold for an abstract set of mutants and see what it implies.

The first step is to pin down what is meant by the "probability of reliability" of a test method.

To this end, let us hypothesize a probability distribution over the sample space consisting of all the subsets of $\mathcal{P} \times \mathcal{S}$. $\text{pr}(X)$ for some event $X \subseteq \mathcal{P} \times \mathcal{S}$ will denote the probability that one of the $(p, s) \in X$ will arise in our idealization of the programming process (given at the beginning of Section II). Of course, one could never hope to establish the actual values of this distribution, because they vary over time with the experience of individual programmers, the state-of-the-art of programming, and the growing expectations of consumers of computer technology.

Despite this, we can make general inferences about the distribution. For instance, the fact that we need to do testing or verification at all tells us that the likelihood of a newly written program being correct is unacceptably small. Perhaps

$$\text{pr}\{(p, s) \mid p \text{ corr } s \cong 0\}.$$

Recalling that the rel relation was chosen to identify those programs and specifications for which we could rely on the results of testing, we can readily define the probability of reliability of a test method as the probability of a program-specification pair arising that is reliably tested.

Definition III.B.3-1: For a test method M in a testing system in which a probability distribution pr is defined, the *probability of reliability* of M is given by

$$\text{pr}(M) = \text{pr}\{(p, s) \mid p \text{ rel}_M s\}.$$ ∎

An alternative definition is to use the conditional probability that p corr s given that p ok_M s. It can be shown that test methods are ranked in exactly the same way under both definitions. Qualitative results being the same, we will use the simpler definition.

We note that this new characterization of the power of a test method is consistent with the old one.

Theorem III.B.3-2: Given two test methods M and N in a testing system for which a probability distribution pr is defined,

$$M \geqslant N \Rightarrow \text{pr}(M) \geqslant \text{pr}(N).$$

Proof:

$$M \geqslant N \Rightarrow$$

$$\forall p(\text{ms}_M(\{p\}) \supseteq \text{ms}_N(\{p\})) \Rightarrow$$

$$\forall p(\{s \mid p \text{ rel}_M s\} \supseteq \{s \mid p \text{ rel}_N s\}) \Rightarrow$$

$$\{(p, s) \mid p \text{ rel}_M s\} \supseteq \{(p, s) \mid p \text{ rel}_N s\} \Rightarrow$$

$$\text{pr}\{(p, s) \mid p \text{ rel}_M s\} \geqslant \text{pr}\{(p, s) \mid p \text{ rel}_N s\} \Rightarrow$$

$$\text{pr}(M) \geqslant \text{pr}(N).$$ ∎

We now need to establish some vocabulary for discussing mutation analysis formally.

Definition III.B.3-2: For every program p we denote the *set of mutant programs* derived from it by mutants(p). We assume that $p \in \text{mutants}(p)$ as the result of an identity mutant operator. ∎

As we noted earlier, mutation analysis is defined in a set-choice testing system in which $\mathcal{T}$ is simply the set of input data.

We will need to be able to compare programs with each other in much the same way we compare programs with specifications using corr and ok.

Definition III.B.3-3: In the mutation testing system, $p(t)$ denotes the output of program p run on test (datum) t. Additionally, if $\mathcal{T}'$ is the powerset of $\mathcal{T}$ and $\mathcal{T}''$ is the powerset of $\mathcal{T}'$, and if t, t', and t'' are elements of $\mathcal{T}$, $\mathcal{T}'$, and $\mathcal{T}''$, respec-

tively, we define three kinds of program equivalence as follows:

$$p \equiv q \Leftrightarrow \forall t(p(t) = q(t))$$

$$p \equiv_{t'} q \Leftrightarrow \forall t(t \in t' \Rightarrow p(t) = q(t))$$

$$p \equiv_{t''} q \Leftrightarrow \exists t'(t' \in t'' \wedge p \equiv_{t'} q).$$

These are the analogs of corr, ok′, and ok″, but relating programs to one another rather than to specifications. ■

As has been the case implicitly throughout this section, we assume that correctness is based entirely on the function computed by a program, or

$$p \equiv q \Rightarrow (p \text{ corr } s \Leftrightarrow q \text{ corr } s).$$

We are now in a position to define the mutation test method and to formalize the competent programmer hypothesis.

Definition III.B.3-4:

$$\text{MUTATION}(p) =$$

$$\{T \mid \forall q(q \in \text{mutants}(p) \Rightarrow (p \equiv_T q \Rightarrow p \equiv q))\}.$$

In other words, mutation testing data distinguishes every nonequivalent mutant from p. ■

Definition III.B.3-5: The *competent programmer hypothesis* is

$$\text{pr}\{(p, s) \mid \exists q(q \in \text{mutants}(p) \wedge q \text{ corr } s)\} \cong 1.$$

In other words, the event of one of a program's mutants being correct is almost certain. ■

Contrast with the assertion earlier that the event of a program itself being correct is very unlikely.

The claim of Budd *et al.* can now be formalized as "the competent programmer hypothesis implies pr(MUTATION) $\cong$ 1." Unfortunately, this is not true. The following example shows that there are testing systems in which the implication fails utterly. In the example we see a situation in which the competent programmer hypothesis has a probability of one, and yet mutation testing has a probability of reliability equal to that of no testing at all.

Example III.B.3-1: Consider a testing system where

$$\mathcal{P} = \{p_1, p_2, p_3\}$$

$$\mathcal{S} = \{s_{12}, s_{23}\}$$

$$\mathcal{T} = \{t_1, t_2\}.$$

The programs are functions from $\mathcal{T}$ to the integers, with the values given in Table I. They are all distinguishable on all inputs, hence $\equiv_t$ for each t is the same as equality.

The specifications are relations whose values are given in the same table.

Taking correctness to be partial correctness in the sense we have used frequently before, we can derive Table II.

No matter how we choose the function mutants(p), we can see that for every program p,

$$\{t_2\} \in \text{MUTATION}(p).$$

This is because of the distinguishability of programs:

$$\forall p\ \forall q(p \equiv_{\{t_2\}} q \Leftrightarrow p = q) \Rightarrow$$

$$\forall p\ \forall q(q \in \text{mutants}(p) \Rightarrow (p \equiv_{\{t_2\}} q \Rightarrow p \equiv q)).$$

TABLE I
$\mathcal{P}$ AND $\mathcal{S}$ FOR EXAMPLE III.B.3-1

t	$p_1(t)$	$p_2(t)$	$p_3(t)$	$s_{12}(t)$	$s_{23}(t)$
t_1	1	2	3	1,2	2,3
t_2	4	5	6	4,5,6	4,5,6

TABLE II
corr AND ok FOR EXAMPLE III.B.3-1

	corr		ok(t_1)		ok(t_2)	
	s_{12}	s_{23}	s_{12}	s_{23}	s_{12}	s_{23}
p_1	T	F	T	F	T	T
p_2	T	T	T	T	T	T
p_3	F	T	F	T	T	T

The consequence of this is that $\text{rel}_{\text{MUTATION}}$ = corr, because

$$p \text{ rel}_{\text{MUTATION}}\ s \Leftrightarrow$$

$$p \text{ ok}''_{\text{MUTATION}(p)}\ s \Rightarrow p \text{ corr } s \Leftrightarrow$$

$$\exists T(T \in \text{MUTATION}(p) \Rightarrow p \text{ ok}'_T\ s) \Rightarrow p \text{ corr } s \Leftrightarrow$$

$$p \text{ ok}'_{\{t_2\}}\ s \Rightarrow p \text{ corr } s \Leftrightarrow$$

$$\text{true} \Rightarrow p \text{ corr } s \Leftrightarrow$$

$$p \text{ corr } s.$$

Thus,

$$\text{pr}(\text{MUTATION}) =$$

$$\text{pr}\{(p, s) \mid p \text{ rel}_{\text{MUTATION}}\ s\} =$$

$$\text{pr}\{(p, s) \mid p \text{ corr } s\}.$$

The probability of reliability of mutation testing in this context is independent of the mutation operators [or, equivalently, the choice of mutants(p)] and is fixed at the theoretical minimum value for any test method, the probability of correctness without testing.

The choice of mutants(p) does, on the other hand, influence the degree to which the competent programmer hypothesis holds. If we choose mutants as given in Table III, then for every choice of p and s, p has a correct mutant:

$$\text{pr}\{(p, s) \mid \exists q(q \in \text{mutants}(p) \Rightarrow q \text{ corr } s)\} =$$

$$\text{pr}\{(p, s) \mid p \in \mathcal{P} \wedge s \in \mathcal{S}\} =$$

$$1.$$

Thus, we can conclude that the competent programmer hypothesis can hold with the highest degree of certainty, but at the same time the probability of reliability of mutation testing can be the minimum possible for any test method. ■

Despite the fact that the assumption of mutation testing can fail in the general case, it is possible to find restricted situations in which it does work. One way of doing this is to impose a restriction on the testing system.

Definition III.B.3-6: The *unique specification property* of a testing system in which $\mathcal{T} = D$, the domain of programs, is

$$p \text{ ok}_t\ s \wedge q \text{ ok}_t\ s \Rightarrow p(t) = q(t).$$ ■

TABLE III
MUTANTS FOR EXAMPLE III.B.3-1

p	mutants(p)
p_1	$\{p_1, p_2\}$
p_2	$\{p_2, p_3\}$
p_3	$\{p_3, p_1\}$

A careful reading of Budd and Angluin [3] reveals that the implicit testing system within which they study mutation analysis has the unique specification property. Their specifications are, in fact, functions, and by correctness Budd and Angluin appear to mean total correctness.

Theorem III.B.3-3: pr(MUTATION) $\cong 1$, provided that the function mutants satisfies the competent programmer hypothesis, and that the underlying testing system has the unique specification property.

Proof: Our goal is to show that the event of reliability under MUTATION contains the event of a program having a correct mutant. This is the same as showing that if we pick a pair (p, s) such that $\exists q(q \in \text{mutants}(p) \wedge q \text{ corr } s)$, $p \text{ rel}_{\text{MUTATION}} s$ also holds.

We can divide the problem into two cases, p corr s and $\sim(p \text{ corr } s)$. The first case is easy:

$$p \text{ corr } s \Rightarrow$$

$$\sim(p \text{ ok}''_{\text{MUTATION}} s) \vee p \text{ corr } s \Rightarrow$$

$$p \text{ rel}_{\text{MUTATION}} s.$$

Therefore, assume that $\sim(p \text{ corr } s)$. By our original choice of p and s, we can find another program, say q_0, such that

$$q_0 \in \text{mutants}(p) \wedge q_0 \text{ corr } s.$$

We also know that $p \not\equiv q_0$ because q_0 is correct and p is not. By the definition of MUTATION we then conclude that

$$\forall T(T \in \text{MUTATION}(p) \Rightarrow p \not\equiv_T q_0)$$

and

$$\forall T(T \in \text{MUTATION}(p) \Rightarrow \exists t(t \in T \wedge p(t) \neq q_0(t))).$$

Now, by the unique specification property and the knowledge that q_0 corr s, we can assert

$$\forall T(T \in \text{MUTATION}(p) \Rightarrow \exists t(t \in T \wedge \sim(p \text{ ok}_t s))).$$

Without this property of the testing system, it would be perfectly possible for p to differ from q_0 at some t and yet for both p and q_0 to test correctly. p is incorrect by assumption and therefore will test incorrectly for some element of $\mathcal{T}$. What we are saying is that mutation analysis may blindly choose an irrelevant distinction between p and q_0. The unique specification property asserts that there are no irrelevant distinctions between programs.

Quickly finishing the proof, we note that the above assertion is the definition of

$$\sim(p \text{ ok}''_{\text{MUTATION}} s)$$

from which, because $\sim(p \text{ corr } s)$, we conclude that

$$p \text{ rel}_{\text{MUTATION}} s.$$

Now that we have reliability in both cases,

$$\text{pr}\{(p, s) \mid \exists q(q \in \text{mutants}(p) \wedge q \text{ corr } s)\} \leqslant \text{pr}\{(p, s) \mid p \text{ rel}_{\text{MUTATION}} s\}$$

and

$$\text{pr}(\text{MUTATION}) \cong 1. \quad \blacksquare$$

The unique specification property is significant. The competent programmer hypothesis is a property of the test method. Should it be called into question, the proponents of mutation testing simply need to modify the method to satisfy the criticism. The unique specification property, on the other hand, is a property of the testing system, and applies to all test methods built in the system, whether they are mutation methods or not. It cannot be acquired by any amount of tuning of the mutation testing method.

Two simple examples will show that, while it is not outrageous, it is quite restrictive. First, most programs must deal with erroneous data of one sort or another. Specification uniqueness insists that a specification provide for every jot and tittle of every error message, lest an erroneous program be mistaken for a correct one because it differs in the punctuation of an error message as well as in some significant way. Second, subroutines are often written to fit into contexts in which only a portion of the potential input domain will ever appear. Specification uniqueness would force the specifier of such a subroutine into the burden of providing behavior for all of the impossible data so that mutation testing will not be fooled.

Thus, we see from a formal analysis of the assumptions behind mutation testing that the test method must be used with caution or its claimed reliability, even given the competent programmer hypothesis, will fail to apply.

C. Specification Dependent Testing

Work on generating test data from an analysis of specifications has been frustrated by the lack of formal specifications and of a uniform language for writing them. We have already observed that Goodenough and Gerhart give a difficult to generalize example of using a decision table specification. Howden restricts his specifications to simple mathematical functions. Geller allows full verification-style specifications, but his examples leave unclear where the boundary between proving and testing should be. His actual examples of test data assertions involve simple mathematical properties like Howden's.

Despite this difficulty, specification dependent testing is practiced, usually in the absence of any kind of formal specification at all. In industry it is known as "black box" testing, where a testing team is given a completed piece of software together with its documentation and is asked to exercise it. The description "black box" is intended to suggest the idea that the testing group has no knowledge of the internals of the software. Their goal is to try to find out if the program does what it is supposed to do without paying attention to how it does it. To the extent that this ideal is realized, black box testers are actually practicing program independent testing. This has limitations in exact analogy to specification independent testing, but it is apparent that practitioners correctly

view it as complementary to the informal path testing that undoubtedly took place throughout the process of writing the program. The need of practitioners for specification dependent testing should be an added spur to research in general methods.

In this section, we will see that mutation testing can be improved by the introduction of specification dependence, and that hardware testing is inherently specification dependent and program independent. In addition, some very general methods of using specifications in testing are outlined.

1) Mutation Testing Revisited: Recently, in the process of implementing a mutation analysis tool, Budd [2] has made a significant change to the principles of mutation analysis. Seemingly trivial, the change actually results in a test method that is incomparable to mutation testing as it was originally formulated. Unlike its predecessor, the new method is specification dependent and, significantly, is reliable under the competent programmer hypothesis. Unfortunately, the new method seems in ways to be more difficult to apply than the old one.

The new approach to mutation analysis produces mutants in the usual way, but assumes that the program and its mutants are imbedded in a driver that monitors the execution of the program on test data, and that reports whether or not the program meets its specification on the tests. Instead of comparing the raw outputs of a program and its mutants, we now compare the outputs of the monitor. Otherwise, the new and old methods are the same. In other words, the new mutation analysis is exactly the same as the old one except that it uses a new notion of program equivalence, equivalence with respect to a specification. This would appear to be precisely what is needed to avoid the problem described in Section III.B.3, and we confirm this when we proceed formally, below.

It is important to notice that the new version of mutation analysis uses a formal, implemented specification in the process of judging the adequacy of a candidate test data set. This is a profound change from the old scheme in which no formal specification was used. The creation of the formal specification was used. The creation of the formal specification adds significantly to the intellectual difficulty of mutation testing, and this intangible cost must be considered in judging the value of the new method. On the other hand, we have seen that the incorporation of specifications into the process of test data selection is a natural and important next step in the development of testing. It is likely that research in this vein (some of which is outlined at the end of this paper) will contribute some method to the process of specification writing. Certainly the shift in mutation testing represents another vote for the routine use of formal specifications.

Using the context of the last section, we can formalize the new method as follows.

Definition III.C.1-1:

$$\text{NMUTATION}(p,s) = \{T \mid \forall q (q \in \text{mutants}(p) \Rightarrow (p \equiv_{s,T} q \Rightarrow p \equiv_s q))\}$$

where

$$p \equiv_s q \Leftrightarrow (p \text{ corr } s \Leftrightarrow q \text{ corr } s)$$

$$p \equiv_{s,T} q \Leftrightarrow (p \text{ ok}'_T s \Leftrightarrow q \text{ ok}'_T s)$$

and where the other notation is the same as in Section III.B.3. ■

Note the analogy between this definition and Definition III.B.3-4. $\equiv_s$ is substituted for $\equiv$ and $\equiv_{s,T}$ is substituted for $\equiv_T$.

Theorem III.C.1-1: pr(NMUTATION) $\cong$ 1, provided that the function mutants satisfies the competent programmer hypothesis.

Proof: As in Theorem III.B.3-3, we want to show that the event of reliability under NMUTATION contains the event of a program having a correct mutant. Thus, we pick a pair (p, s) such that $\exists q (q \in \text{mutants}(p) \wedge q \text{ corr } s)$. p corr s trivially implies p $\text{rel}_{\text{NMUTATION}}$ s, so we assume $\sim(p \text{ corr } s)$.

Now, we pick q_0 such that $q_0 \in \text{mutants}(p) \wedge q_0$ corr s. It follows that

$$p \not\equiv_s q_0.$$

If we pick $T_0 \in \text{NMUTATION}(p, s)$, we can see that

$$p \not\equiv_{s,T_0} q_0.$$

Since q_0 corr s, it must be that

$$\sim(p \text{ ok}'_{T_0} s)$$

and because of the arbitrary choices of q_0 and T_0

$$\sim(p \text{ ok}''_{\text{NMUTATION}} s)$$

$$p \text{ rel}_{\text{NMUTATION}} s$$

and finally, as in the other proof,

$$\text{pr}(\text{NMUTATION}) \cong 1. \quad \blacksquare$$

It is tempting to guess at this point that NMUTATION $\geqslant$ MUTATION. It is true that NMUTATION cannot be fooled by irrelevant distinctions between a program and its mutants. On the other hand, if a program and one of its mutants are both incorrect, they are equivalent under the specification, and the new method will permit any test data to be used. Under the same conditions, the old method may actually expose the error. It appears, therefore, that in the absence of the competent programmer hypothesis the two methods are actually incomparable. This can be made formal as follows.

Theorem III.C.1-2:

$$\text{MUTATION} \not\geqslant \text{NMUTATION}$$

and

$$\text{NMUTATION} \not\geqslant \text{MUTATION}.$$

Proof: Considering the testing system represented in Table IV, we see that both $\sim(p_1 \text{ corr } s_1)$ and $\sim(p_2 \text{ corr } s_2)$. By reasoning similar to that in Example III.B.3-1, we can see that p_1 $\text{ok}_{\text{MUTATION}}$ s_1 and $\sim(p_1$ $\text{ok}_{\text{NMUTATION}}$ $s_1)$, while at the same time $\sim(p_2$ $\text{ok}_{\text{MUTATION}}$ $s_2)$ and p_2 $\text{ok}_{\text{NMUTATION}}$ s_2. Thus each method detects an erroneous program that the other misses. ■

Before we can conclude our discussion of the new mutation analysis, we must mention a feature of the method that at first appears paradoxical. Consider the situation of a tester who is given program p, and specification s, and is asked if test data set T is adequate under the new mutation method. It is likely that some mutant, say q, is indistinguishable from p under s and T (i.e., $p \equiv_{s,T} q$). The tester's challenge is now to reject T in favor of a larger set that does distinguish p and q, or to prove p and q fully equivalent under s ($p \equiv_s q$), thus admitting that

TABLE IV
$\mathcal{P}$ AND $\mathcal{S}$ FOR THEOREM III.C.1-2

	$P_1(t)$	$P_2(t)$	$P_3(t)$	$P_4(t)$	$S_1(t)$	$S_2(t)$
t_1	1	5	2	5	1	5
t_2	3	7	4	8	3,4	3

no test can distinguish them. One way to go about proving $p \equiv_s q$ is to prove the truth or falsity of p corr s and then prove that q corr s has the same value. But, if we know the truth value of p corr s, there is no need to go on evaluating test data, because the sole purpose of the analysis is to help find that information.

In fact, trying to prove p corr s and then q corr s is not the practical way to prove $p \equiv_s q$. In practice, q differs from p in one syntactically very small way; they differ by a mutation to one statement. It is within reason, and Budd confirms that it is true in practice, that one can prove that the mutation does not change the correctness of the program. Because of its need for a formal specification, this is probably more difficult than the pure program equivalence proof required by the old form of mutation testing. That it is easier than a program correctness proof can only be established by experience.

2) Hardware Testing: The testing of digital electronic hardware, of course, is a subject that has filled volumes. Our purpose here is simply to point out some of the similarities and differences between hardware and software testing, and to make the observation that hardware testing has traditionally been specification dependent.

In order to see the difference between hardware and software testing, we need to recognize that there are really two different applications of testing in the production of hardware and software. The first is the confirmation that the prototype proposed by the engineer or programmer does what the ultimate user wants to do. The second is the confirmation that each replica of the final prototype is identical (within limits) to the original. In software, the technology for duplicating programs is so well established that there seems to be no need, except in extraordinary cases, to question the accuracy of a copy. Thus, it is the first testing application, the testing of prototypes, that dominates the theory and practice of software testing. In hardware, on the other hand, the physical replicas can deteriorate with time, and manufacturing itself is not always highly reliable. It is here, in the second application of testing, that the subject of hardware testing concentrates. Why the testing of hardware prototypes or designs takes a lower precedence is debatable. It may be that mass produced hardware components have tended to be less complex than software components, making the correctness of hardware designs a matter of intuition. It may also be that the testing of designs is an inherently more difficult problem, requiring, as it does, specifications of a different sort than the component being designed. It is clear, however, that the development of VLSI eventually will make the testing of designs as important in hardware as in software.

Thus, hardware and software testing contrast formally in that a traditional hardware testing system will have the set of specifications equal to the set of "programs." In software testing, as we have seen, it is atypical to use a testing system in which $\mathcal{P}$ and $\mathcal{S}$ are equal.

TABLE V
MUTANTS FOR THEOREM III.C.1-2

p	mutants(p)
p_1	$\{p_1, p_3\}$
p_2	$\{p_2, p_4\}$
p_3	$\{p_3\}$
p_4	$\{p_4\}$

The problem faced by a hardware tester can be made a bit more specific. Using the manufacture of integrated circuit chips as an example, the tester is repeatedly given two chips, one of which is constant and correct by definition (the prototype), and one is unknown (it has just been manufactured or returned from the field). It makes sense in this situation for the tester to study the prototype, the manufacturing process, and the aging process, trying to anticipate the kind of faults that are likely to occur in the mass produced chips. He or she then tries to assemble a collection of tests that is guaranteed to expose any of the anticipated faults.

In the case of faults introduced during manufacture, what we have is the fault model approach to hardware testing [16], which is similar in some respects to mutation testing of software. Our analysis of the manufacturing process gives us a function *fault* that maps a prototype chip into the set of chips that are likely to come out of the manufacturing process.

For example, a frequently mentioned fault model is the "stuck-on" model in which fault(p) is the set of chips that differ from p only in that one gate is replaced by a wire carrying a constant (zero or one) value.

Since the manufacturing process will probably be constant over a wide variety of chip designs, one would like to find a general method of deriving an adequate set of tests from any given prototype chip. This amounts to finding a test method M such that (recalling that $\mathcal{P} = \mathcal{S}$):

$$\forall p(\text{fault}(p)\ \text{rel}_M\ p).$$

It is immediately evident that the power of M depends on the size of fault(p) and that in general we will not obtain full reliability unless fault(p) = $\mathcal{P}$.

The spirit of the fault model, however, is that it identifies the *likely* faulty chips. Thus, as in mutation analysis, we have an assumption about the probability distribution of program-specification pairs, or in this case, chip-chip pairs. The assumption is, formally,

$$\text{pr}\{(q, p) \mid q \in \text{fault}(p)\} \cong 1.$$

It follows trivially that

$$\text{pr}(M) \cong 1.$$

In other words, the probability of reliability of M depends on the probability that the manufactured chips contain the modeled faults.

All this is no great surprise. It merely serves to illustrate the essential similarity between hardware and software testing. Two fundamental questions are raised, however. First, we notice that the kind of hardware testing we have described is specification or prototype dependent and program indepen-

dent. We have learned earlier that the greatest reliability is obtained with test methods that are dependent in both programs and specifications. Is it possible to improve hardware testing by tailoring the tests to some of the properties of the "program," the manufactured chip under test? Second, how can the testing of hardware designs be done when the complexity of VLSI forces this issue? Will the similarity to program testing mean that software techniques will apply, or will an entirely new approach be appropriate?

3) Further Research into Specification Dependent Testing: A number of recent investigations have been made into specification based testing. Because of the lack of a single, widely used specification language, each investigator includes his own unique variety of specifications. Because the testing system varies from one investigator to the next, there is very little that can be said about the relative powers of the methods. This section consists, therefore, of informal descriptions and observations about the methods.

Weyuker and Ostrand make a bit of progress in the direction of testing based on general specifications. They illustrate how a simple predicate calculus specification can be decomposed into a number of disjoint input conditions and corresponding restricted output specifications. The conditions on the input produce a partition of the input set that is probably different from the one that is derived from branch testing. They suggest finding a mutual refinement of the two partitions, and selecting test data from all of the classes in the refinement. In the context of the preceding two sections, it should be clear without proof that such a method would be strictly more powerful than selecting data from one of the two original partitions. The problem with their approach is, like Goodenough and Gerhart's, that it is given in the form of examples and it is not clear how to generalize it, especially to formulas with quantifiers.

Another relatively early attempt is the work of Chow [5]. In a manner reminiscent of Howden's, he restricts the set of specifications to the set of finite state machines. In effect, programs to Chow are also finite state machines, but a fairer way of putting it is that he accepts any usual program paired with an abstraction that leaves only finite state behavior visible. He shows that universally reliable test methods can be constructed in this kind of testing system, and he shows that fewer test data are necessary than might first be expected. Since Chow is dealing strictly with universally reliable test methods, his interest is mainly in the complexity of the testing process, something that is secondary to our concerns. Finite state machine specifications are clearly applicable to some software, such as network protocols and lexical scanners, but due to the inherent limitation of the computational model they cannot form the basis for a general method.

The "domain testing" of White and Cohen [19], as implemented, is one of the enhanced versions of path testing alluded to above. It is mentioned here, because in the abstract it is an interesting specification dependent testing method. Similar to Chow's, their method abstracts all computation out of the program, leaving only the program's branching structure to assign input data to equivalence classes. Specifications, were they made explicit, would define the equivalence classes that a satisfactory program would have to exhibit. Test data are chosen to show whether or not the boundaries between the classes are in the specified places. The problem with the method, of course, is that the detection of "domain errors" is only a small part of the problem of showing program correctness. Also, the method is applicable only to programs whose input set has a metric to provide a definition of boundary. Perhaps domain testing can be generalized by a careful study of what constitute boundary values of programming problems.

Two recent papers move closer to the goal of test data generation from general specifications. Richardson and Clarke [17] and Cartwright [4] both effectively follow Weyuker and Ostrand in deriving a partition of the input set from a specification. Each of the two suggest new specification languages which are high level applicative programming languages. Richardson and Clarke explicitly do a path analysis of the specification to obtain a partition which they then use to refine a path testing partition. Cartwright is less specific, but suggests using a symbolic evaluation of conditions in the specification, partitioning the input set at branch points in the evaluation. The briefly described process probably amounts to a kind of path analysis of the specification.

Both of these approaches are vulnerable to the criticism applied to all operational semantics, that specifying a program with another program is redundant, and that it should be possible to write a compiler for the specification language, thus eliminating the need to verify or test the program. A minor criticism is that both approaches require users of the method to learn a new language.

Cartwright argues that a new specification language is required, because the language of verification, predicate calculus, has failed to permit practical verification. The tedium and detail that make verification difficult, however, are generally present only in the proofs of programs, and not in their predicate calculus specifications. Predicate calculus provides ample opportunity for capturing the abstractions needed to specify large programs, and test data derived from a predicate calculus specification will carry with it none of the tedium associated with proofs of correctness. In addition, predicate calculus is as close as we can get to a *lingua franca* for specifications.

Gourlay [8] approaches this goal of using predicate calculus specifications by recognizing that disjuncts from such a specification are themselves partial specifications of a program and often represent simple computations that need to be performed. By striking an analogy between disjoints of a specification and paths of a program, one can treat quantifiers as if they were loops in a program being path tested. In this way a very general and effective test method can be obtained. A use of the same principle of recognizing paths in specifications occurs in the work of Gannon *et al.* [6], where an abstract data structure is tested against its algebraic specification by applying a path testing criterion simultaneously to the specification and the implementation.

IV. Conclusion

We have seen a formal framework for testing that unifies and extends previous work, that is mathematically interesting, and that has born fruit of consequences to practical testing. The

attempt to prove the commonly accepted relation between statement testing and branch testing led to a hierarchy of improvements to path testing filling an expressed need in practice. A formal analysis of the assumptions underlying mutation testing led to a clarification of the assumptions and a restriction on when mutation testing can be used reliably. The theory's confirmation of the need for testing based on specifications reemphasizes the importance of research in this area.

This list certainly does not exhaust the possibilities of the theory. Perhaps the most evident direction in which to proceed is to continue research into specification dependent methods with the goal of putting them into routine use. This will require work not only in methods of test data selection, but in specification languages themselves. Another topic is to pursue the relation between software and hardware testing with the hope of cross fertilization between the fields. Another is the experimental investigation of the probability distributions (for hardware and software) that give rise to probabilities of reliability with the hope of explaining the observed reliability of relatively simple test methods.

A number of theoretical applications also deserve further attention. Careful study of the expanded sets of programs and specifications for the path testing methods and for specification testing would be of interest. Knowing these sets probably would not add to our knowledge of the relative powers of these methods, but the examples might contribute to answers to some of the theoretical questions left unanswered in Section II. One particular question along this line is the significance of Kennaway and Hoare's continuity assumption in terms of the finiteness or computability of test methods. Perhaps the mathematical characterization of actual test methods can be significantly tighter than the one given in Section II.

The list of further questions can be made arbitrarily long. It appears as if one contribution of the theoretical framework is the ability to pose questions about testing unambiguously, and to have some idea of the form answers may take. In any case, the theory of testing is clearly not the sterile subject it might appear to be.

Acknowledgment

The author wishes to thank all the people who helped in one way or another with the research presented in this paper. Special thanks go to B. Rounds whose encouragement and example were essential to its completion.

References

[1] T. A. Budd *et al.*, "Mutation analysis," Dep. Comput. Sci., Yale Univ., Res. Rep. 155, 1979.
[2] T. A. Budd, personal communications, 1982.
[3] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," Dep. Comput. Sci., Univ. Arizona, Res. Rep. TR 80-19, 1980.
[4] R. Cartwright, "Formal program testing," in *Proc. 8th Ann. ACM Symp. Principles of Programming Languages*, 1981.
[5] T. S. Chow, "Testing software design modeled by finite-state mahines," *IEEE Trans. Software Eng.*, vol. SE-4, May 1978.
[6] J. Gannon, P. McMullin, and R. Hamlet, "Data-abstraction implementation, specification, and testing," *ACM Trans. Programming Languages Syst.*, vol. 3, July 1981.
[7] M. Geller, "Test data as an aid in proving program correctness," *Commun. Ass. Comput. Mach.*, vol. 21, May 1978.
[8] J. S. Gourlay, "Theory of testing computer programs," Ph.D. dissertation, Univ. Michigan, Univ. Microfilms, 1981.
[9] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Software Eng.*, vol. SE-1, June 1975.
[10] M. A. Holthouse and M. J. Hatch, "Experience with automated testing analysis," *IEEE Computer*, vol. 12, Aug. 1979.
[11] W. E. Howden, "Algebraic program testing," *Acta Informatica*, vol. 10, Jan. 1978.
[12] —, "Reliability of the path analysis testing strategy," *IEEE Trans. Software Eng.*, vol. SE-2, Sept. 1976.
[13] J. C. Huang, "An approach to program testing," *Comput. Surveys*, vol. 7, Sept. 1975.
[14] J. R. Kennaway and C.A.R. Hoare, "A theory of nondeterminism," in *Automata, Languages and Programming (Lecture Notes Comput. Sci.)*, vol. 85, J. W. deBakker and J. van Leeuwen, Eds. New York: Springer-Verlag, 1980.
[15] L. Lamport, "On the proof of correctness of a calendar program," *Commun. Ass. Comput. Mach.*, vol. 22, Oct. 1979.
[16] K. W. Lai, "Functional testing of digital systems," Ph.D. dissertation, Carnegie-Mellon Univ., 1982.
[17] D. J. Richardson and L. A. Clarke, "A partition analysis method to increase program reliability," in *Proc. 5th IEEE Int. Conf. Software Eng.*, 1981.
[18] E. J. Weyuker and T. J. Ostrand, "Theories of program testing and the application of revealing subdomains," *IEEE Trans. Software Eng.*, vol. SE-6, 1980.
[19] L. J. White and E. I. Cohen, "A domain strategy for computer program testing," *IEEE Trans. Software Eng.*, vol. SE-6, May 1980.

John S. Gourlay (M'81) received the B.S. degree in 1971 and the Ph.D. degree in 1981, both from the University of Michigan, Ann Arbor.

From 1971 to 1976 he worked as a System Programmer at the General Motors Proving Ground, Milford, MI. He worked for the University of Michigan in various teaching and research capacities during the period 1976–1981. Since then he has been an Assistant Professor in the Department of Computer and Information Science, Ohio State University, Columbus. His research interests continue to be in specification and testing of software.

Dr. Gourlay is a member of the Association for Computing Machinery and Sigma Xi.