Software Unit Test Coverage and Adequacy

HONG ZHU

Nanjing University

PATRICK A. V. HALL AND JOHN H. R. MAY

The Open University, Milton Keynes, UK

Objective measurement of test quality is one of the key issues in software testing. It has been a major research focus for the last two decades. Many test criteria have been proposed and studied for this purpose. Various kinds of rationales have been presented in support of one criterion or another. We survey the research work in this area. The notion of adequacy criteria is examined together with its role in software dynamic testing. A review of criteria classification is followed by a summary of the methods for comparison and assessment of criteria.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms: Measurement, Performance, Reliability, Verification

Additional Key Words and Phrases: Comparing testing effectiveness, fault-detection, software unit test, test adequacy criteria, test coverage, testing methods

1. INTRODUCTION

In 1972, Dijkstra claimed that "program testing can be used to show the presence of bugs, but never their absence" to persuade us that a testing approach is not acceptable [Dijkstra 1972]. However, the last two decades have seen rapid growth of research in software testing as well as intensive practice and experiments. It has been developed into a validation and verification technique indispensable to software engineering discipline. Then, where are we today? What can we claim about software testing?

In the mid-'70s, in an examination of the capability of testing for demonstrating the absence of errors in a program, Goodenough and Gerhart [1975, 1977] made an early breakthrough in research on software testing by pointing out that the central question of software testing is "what is a test criterion?", that is, the criterion that defines what constitutes an adequate test. Since then, test criteria have been a major research focus. A great number of such criteria have been proposed and investigated. Considerable research effort has attempted to provide support for the use of one criterion or another. How should we understand these different criteria? What are the future directions for the subject?

In contrast to the constant attention given to test adequacy criteria by aca-

Authors' addresses: H. Zhu, Institute of Computer Software, Nanjing University, Nanjing, 210093, P.R. of China; email: ⟨hzhu@netra.nju.edu.cn⟩; P.A.V. Hall and J.H.R. May, Department of Computing, The Open University, Walton Hall, Milton Keynes, MK76AA, UK.

demics, the software industry has been slow to accept test adequacy measurement. Few software development standards require or even recommend the use of test adequacy criteria [Wichmann 1993; Wichmann and Cox 1992]. Are test adequacy criteria worth the cost for practical use?

Addressing these questions, we survey research on software test criteria in the past two decades and attempt to put it into a uniform framework.

1.1 The Notion of Test Adequacy

Let us start with some examples. Here we seek to illustrate the basic notions underlying adequacy criteria. Precise definitions will be given later.

—*Statement coverage.* In software testing practice, testers are often required to generate test cases to execute every statement in the program at least once. A *test case* is an input on which the program under test is executed during testing. A *test set* is a set of test cases for testing a program. The requirement of executing all the statements in the program under test is an adequacy criterion. A test set that satisfies this requirement is considered to be adequate according to the statement coverage criterion. Sometimes the percentage of executed statements is calculated to indicate how adequately the testing has been performed. The percentage of the statements exercised by testing is a measurement of the adequacy.

—*Branch coverage.* Similarly, the branch coverage criterion requires that all control transfers in the program under test are exercised during testing. The percentage of the control transfers executed during testing is a measurement of test adequacy.

—*Path coverage.* The path coverage criterion requires that all the execution paths from the program's entry to its exit are executed during testing.

—*Mutation adequacy.* Software testing is often aimed at detecting faults in software. A way to measure how well this objective has been achieved is to plant some artificial faults into the program and check if they are detected by the test. A program with a planted fault is called a *mutant* of the original program. If a mutant and the original program produce different outputs on at least one test case, the fault is detected. In this case, we say that the mutant is *dead* or *killed* by the test set. Otherwise, the mutant is still *alive*. The percentage of dead mutants compared to the mutants that are not equivalent to the original program is an adequacy measurement, called the *mutation score* or *mutation adequacy* [Budd et al. 1978; DeMillo et al. 1978; Hamlet 1977].

From Goodenough and Gerhart's [1975, 1977] point of view, a software test adequacy criterion is a predicate that defines "what properties of a program must be exercised to constitute a 'thorough' test, i.e., one whose successful execution implies no errors in a tested program." To guarantee the correctness of adequately tested programs, they proposed *reliability* and *validity* requirements of test criteria. Reliability requires that a test criterion always produce consistent test results; that is, if the program tested successfully on one test set that satisfies the criterion, then the program also tested successfully on all test sets that satisfies the criterion. Validity requires that the test always produce a meaningful result; that is, for every error in a program, there exists a test set that satisfies the criterion and is capable of revealing the error. But it was soon recognized that there is no computable criterion that satisfies the two requirements, and hence they are not practically applicable [Howden 1976]. Moreover, these two requirements are not independent since a criterion is either reliable or valid for any given software [Weyuker and Ostrand 1980]. Since then, the focus of research seems to have shifted from seeking theoretically ideal criteria to the search for practically applicable approximations.

Currently, the software testing literature contains two different, but closely related, notions associated with the term test data adequacy criteria. First, an adequacy criterion is considered to be a stopping rule that determines whether sufficient testing has been done that it can be stopped. For instance, when using the statement coverage criterion, we can stop testing if all the statements of the program have been executed. Generally speaking, since software testing involves the program under test, the set of test cases, and the specification of the software, an adequacy criterion can be formalized as a function C that takes a program p, a specification s, and a test set t and gives a truth value *true* or *false*. Formally, let P be a set of programs, S be a set of specifications, D be the set of inputs of the programs in P, T be the class of test sets, that is, $T = 2^D$, where 2^X denotes the set of subsets of X.

Definition 1.1 (*Test Data Adequacy Criteria as Stopping Rules*). A test data adequacy criterion C is a function C: $P \times S \times T \rightarrow \{true, false\}$. $C(p, s, t) =$ *true* means that t is adequate for testing program p against specification s according to the criterion C, otherwise t is inadequate.

Second, test data adequacy criteria provide measurements of test quality when a degree of adequacy is associated with each test set so that it is not simply classified as good or bad. In practice, the percentage of code coverage is often used as an adequacy measurement. Thus, an adequacy criterion C can be formally defined to be a function C from a program p, a specification s, and a test set t to a real number $r = C(p, s, t)$, the degree of adequacy [Zhu and Hall 1992]. Formally:

Definition 1.2 (*Test Data Adequacy Criteria as Measurements*). A test data adequacy criterion is a function C, C: $P \times S \times T \rightarrow [0,1]$. $C(p, s, t) = r$ means that the adequacy of testing the program p by the test set t with respect to the specification s is of degree r according to the criterion C. The greater the real number r, the more adequate the testing.

These two notions of test data adequacy criteria are closely related to one another. A stopping rule is a special case of measurement on the continuum since the actual range of measurement results is the set {0,1}, where 0 means *false* and 1 means *true*. On the other hand, given an adequacy measurement M and a degree r of adequacy, one can always construct a stopping rule M_r such that a test set is adequate if and only if the adequacy degree is greater than or equal to r; that is, $M_r(p, s, t) = \text{true} \Leftrightarrow M(p, s, t) \geq r$. Since a stopping rule asserts a test set to be either adequate or inadequate, it is also called a *predicate rule* in the literature.

An adequacy criterion is an essential part of any testing method. It plays two fundamental roles. First, an adequacy criterion specifies a particular software testing requirement, and hence determines test cases to satisfy the requirement. It can be defined in one of the following forms.

(1) It can be an explicit specification for test case selection, such as a set of guidelines for the selection of test cases. Following such rules one can produce a set of test cases, although there may be some form of random selections. Such a rule is usually referred to as a *test case selection criterion*. Using a test case selection criterion, a testing method may be defined constructively in the form of an algorithm which generates a test set from the software under test and its own specification. This test set is then considered adequate. It should be noticed that for a given test case selection criterion, there may exist a number of test case generation algorithms. Such an algorithm may also involve random sampling among many adequate test sets.

(2) It can also be in the form of specifying how to decide whether a given test set is adequate or specifying how to measure the adequacy of a test set. A rule that determines whether a test set is adequate (or more generally, how adequate) is usually referred to as a *test data adequacy criterion*.

However, the fundamental concept underlying both test case selection criteria and test data adequacy criteria is the same, that is, the notion of test adequacy. In many cases they can be easily transformed from one form to another. Mathematically speaking, test case selection criteria are generators, that is, functions that produce a class of test sets from the program under test and the specification (see Definition 1.3). Any test set in this class is adequate, so that we can use any of them equally.[1] Test data adequacy criteria are acceptors that are functions from the program under test, the specification of the software and the test set to a characteristic number as defined in Definition 1.1. Generators and acceptors are mathematically equivalent in the sense of one-one correspondence. Hence, we use "test adequacy criteria" to denote both of them.

Definition 1.3 (*Test Data Adequacy Criteria as Generators* [Budd and Angluin 1982]). A test data adequacy criterion C is a function $C: P \times S \rightarrow 2^T$. A test set $t \in C(p, s)$ means that t satisfies C with respect to p and s, and it is said that t is adequate for (p, s) according to C.

The second role that an adequacy criterion plays is to determine the observations that should be made during the testing process. For example, statement coverage requires that the tester, or the testing system, observe whether each statement is executed during the process of software testing. If path coverage is used, then the observation of whether statements have been executed is insufficient; execution paths should be observed and recorded. However, if mutation score is used, it is unnecessary to observe whether a statement is executed during testing. Instead, the output of the original program and the output of the mutants need to be recorded and compared.

Although, given an adequacy criterion, different methods could be developed to generate test sets automatically or to select test cases systematically and efficiently, the main features of a testing method are largely determined by the adequacy criterion. For example, as we show later, the adequacy criterion is related to fault-detecting ability, the dependability of the program that passes a successful test and the number of test cases required. Unfortunately, the exact relationship between a particular adequacy criterion and the correctness or reliability of the software that passes the test remains unclear.

Due to the central role that adequacy criteria play in software testing, software testing methods are often compared in terms of the underlying adequacy criteria. Therefore, subsequently, we use the name of an adequacy criterion as a synonym of the corresponding testing method when there is no possibility of confusion.

1.2 The Uses of Test Adequacy Criteria

An important issue in the management of software testing is to "ensure that before any testing the objectives of that testing are known and agreed and that the objectives are set in terms that can be measured." Such objectives "should be quantified, reasonable and achievable" [Ould and Unwin 1986]. Almost all test adequacy criteria proposed in the literature explicitly specify particular requirements on software testing. They are objective rules applicable by project managers for this purpose.

For example, branch coverage is a

[1] Test data selection criteria as generators should not be confused with test case generation software tools, which may only generate one test set.

test requirement that all branches of the program should be exercised. The objective of testing is to satisfy this requirement. The degree to which this objective is achieved can be measured quantitatively by the percentage of branches exercised. The mutation adequacy criterion specifies the testing requirement that a test set should be able to rule out a particular set of software faults, that is, those represented by mutants. Mutation score is another kind of quantitative measurement of test quality.

Test data adequacy criteria are also very helpful tools for software testers. There are two levels of software testing processes. At the lower level, testing is a process where a program is tested by feeding more and more test cases to it. Here, a test adequacy criterion can be used as a stopping rule to decide when this process can stop. Once the measurement of test adequacy indicates that the test objectives have been achieved, then no further test case is needed. Otherwise, when the measurement of test adequacy shows that a test has not achieved the objectives, more tests must be made. In this case, the adequacy criterion also provides a guideline for the selection of the additional test cases. In this way, adequacy criteria help testers to manage the software testing process so that software quality is ensured by performing sufficient tests. At the same time, the cost of testing is controlled by avoiding redundant and unnecessary tests. This role of adequacy criteria has been considered by some computer scientists [Weyuker 1986] to be one of the most important.

At a higher level, the testing procedure can be considered as repeated cycles of testing, debugging, modifying program code, and then testing again. Ideally, this process should stop only when the software has met the required reliability requirements. Although test data adequacy criteria do not play the role of stopping rules at this level, they make an important contribution to the assessment of software dependability. Generally speaking, there are two basic aspects of software dependability assessment. One is the dependability estimation itself, such as a reliability figure. The other is the confidence in estimation, such as the confidence or the accuracy of the reliability estimate. The role of test adequacy here is a contributory factor in building confidence in the integrity estimate. Recent research has shown some positive results with respect to this role [Tsoukalas 1993].

Although it is common in current software testing practice that the test processes at both the higher and lower levels stop when money or time runs out, there is a tendency towards the use of systematic testing methods with the application of test adequacy criteria.

1.3 Categories of Test Data Adequacy Criteria

There are various ways to classify adequacy criteria. One of the most common is by the source of information used to specify testing requirements and in the measurement of test adequacy. Hence, an adequacy criterion can be:

—*specification-based*, which specifies the required testing in terms of identified features of the specification or the requirements of the software, so that a test set is adequate if all the identified features have been fully exercised. In software testing literature it is fairly common that no distinction is made between specification and requirements. This tradition is followed in this article also;

—*program-based*, which specifies testing requirements in terms of the program under test and decides if a test set is adequate according to whether the program has been thoroughly exercised.

It should not be forgotten that for both specification-based and program-based testing, the correctness of program outputs must be checked against the specification or the requirements. However, in both cases, the measurement of test adequacy does not depend on the results of this checking. Also, the definition of specification-based criteria given previously does not presume the existence of a formal specification.

It has been widely acknowledged that software testing should use information from both specification and program. Combining these two approaches, we have:

—*combined specification- and program-based criteria*, which use the ideas of both program-based and specification-based criteria.

There are also test adequacy criteria that specify testing requirements without employing any internal information from the specification or the program. For example, test adequacy can be measured according to the prospective usage of the software by considering whether the test cases cover the data that are most likely to be frequently used as input in the operation of the software. Although few criteria are explicitly proposed in such a way, selecting test cases according to the usage of the software is the idea underlying *random testing*, or *statistical testing*. In random testing, test cases are sampled at random according to a probability distribution over the input space. Such a distribution can be the one representing the operation of the software, and the random testing is called representative. It can also be any probability distribution, such as a uniform distribution, and the random testing is called nonrepresentative. Generally speaking, if a criterion employs only the "interface" information—the type and valid range for the software input—it can be called an interface-based criterion:

—*interface-based criteria*, which specify testing requirements only in terms of the type and range of software input without reference to any internal features of the specification or the program.

In the software testing literature, people often talk about *white-box testing* and *black-box testing*. Black-box testing treats the program under test as a "black box." No knowledge about the implementation is assumed. In white-box testing, the tester has access to the details of the program under test and performs the testing according to such details. Therefore, specification-based criteria and interface-based criteria belong to black-box testing. Program-based criteria and combined specification and program-based criteria belong to white-box testing.

Another classification of test adequacy criteria is by the underlying testing approach. There are three basic approaches to software testing:

(1) *structural testing:* specifies testing requirements in terms of the coverage of a particular set of elements in the structure of the program or the specification;
(2) *fault-based testing:* focuses on detecting faults (i.e., defects) in the software. An adequacy criterion of this approach is some measurement of the fault detecting ability of test sets.[2]
(3) *error-based testing:* requires test cases to check the program on certain error-prone points according to our knowledge about how programs typically depart from their specifications.

The source of information used in the adequacy measurement and the underlying approach to testing can be considered as two dimensions of the space of software test adequacy criteria. A software test adequacy criterion can be classified by these two aspects. The review of adequacy criteria is organized according to the structure of this space.

[2] We use the word *fault* to denote defects in software and the word *error* to denote defects in the outputs produced by a program. An execution that produces an error is called a *failure*.

1.4 Organization of the Article

The remainder of the article consists of two main parts. The first part surveys various types of test data adequacy criteria proposed in the literature. It includes three sections devoted to structural testing, fault-based testing, and error-based testing. Each section consists of several subsections covering the principles of the testing method and their application to program-based and specification-based test criteria. The second part is devoted to the rationale presented in the literature in support of the various criteria. It has two sections. Section 5 discusses the methods of comparing adequacy criteria and surveys the research results in the literature. Section 6 discusses the axiomatic study and assessment of adequacy criteria. Finally, Section 7 concludes the paper.

2. STRUCTURAL TESTING

This section is devoted to adequacy criteria for structural testing. It consists of two subsections, one for program-based criteria and the other for specification-based criteria.

2.1 Program-Based Structural Testing

There are two main groups of program-based structural test adequacy criteria: control-flow criteria and data-flow criteria. These two types of adequacy criteria are combined and extended to give dependence coverage criteria. Most adequacy criteria of these two groups are based on the flow-graph model of program structure. However, a few control-flow criteria define test requirements in terms of program text rather than using an abstract model of software structure.

2.1.1 *Control Flow Adequacy Criteria.* Before we formally define various control-flow-based adequacy criteria, we first give an introduction to the flow graph model of program structure.

A. *The flow graph model of program structure.* The control flow graph stems from compiler work and has long been used as a model of program structure. It is widely used in static analysis of software [Fenton et al. 1985; Kosaraju 1974; McCabe 1976; Paige 1975]. It has also been used to define and study program-based structural test adequacy criteria [White 1981]. In this section we give a brief introduction to the flow-graph model of program structure. Although we use graph-theory terminology in the following discussion, readers are required to have only a preliminary knowledge of graph theory. To help understand the terminology and to avoid confusion, a glossary is provided in the Appendix.

A flow graph is a directed graph that consists of a set N of nodes and a set $E \subseteq N \times N$ of directed edges between nodes. Each node represents a linear sequence of computations. Each edge representing transfer of control is an ordered pair $\langle n_1, n_2 \rangle$ of nodes, and is associated with a predicate that represents the condition of control transfer from node n_1 to node n_2. In a flow graph, there is a begin node and an end node where the computation starts and finishes, respectively. The begin node has no inward edges and the end node has no outward edges. Every node in a flow graph must be on a path from the begin node to the end node. Figure 1 is an example of flow graph.

Example 2.1 The following program computes the greatest common divisor of two natural numbers by Euclid's algorithm. Figure 1 is the corresponding flow graph.

```
Begin
   input (x, y);
   while (x > 0 and y > 0) do
         if (x > y)
          then x:= x − y
          else y:= y − x
         endif
   endwhile;
   output (x + y);
  end
```

It should be noted that in the literature there are a number of conventions of flow-graph models with subtle differ-

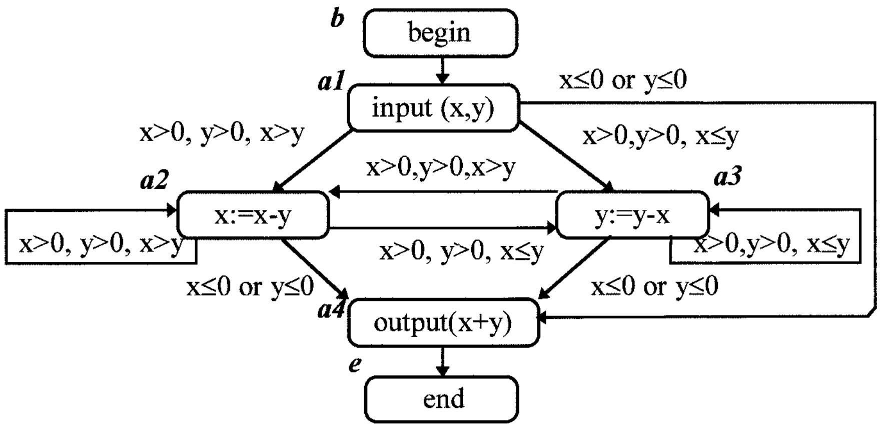

Figure 1. Flow graph for program in Example 2.1.

ences, such as whether a node is allowed to be associated with an empty sequence of statements, the number of outward edges allowed for a node, and the number of end nodes allowed in a flow graph, and the like. Although most adequacy criteria can be defined independently of such conventions, using different ones may result in different measures of test adequacy. Moreover, testing tools may be sensitive to such conventions. In this article no restrictions on the conventions are made.

For programs written in a procedural programming language, flow-graph models can be generated automatically. Figure 2 gives the correspondences between some structured statements and their flow-graph structures. Using these rules, a flow graph, shown in Figure 3, can be derived from the program given in Example 2.1. Generally, to construct a flow graph for a given program, the program code is decomposed into a set of disjoint blocks of linear sequences of statements. A block has the property that whenever the first statement of the block is executed, the other statements are executed in the given order. Furthermore, the first statement of the block is the only statement that may be executed directly after the execution of a statement in another block. Each block corresponds to a node in the flow graph. A control transfer from one block to another is represented by a directed edge between the nodes such that the condition of the control transfer is associated with it.

B. *Control-flow adequacy criteria.* Now, given a flow-graph model of a program and a set of test cases, how do we measure the adequacy of testing for the program on the test set? First of all, recall that the execution of the program on an input datum is modeled as a traverse in the flow graph. Every execution corresponds to a path in the flow graph from the begin node to the end node. Such a path is called a complete computation path, or simply a computation path or an execution path in software testing literature.

A very basic requirement of adequate testing is that all the statements in the program are covered by test executions. This is usually called statement coverage [Hetzel 1984]. But full statement coverage cannot always be achieved because of the possible existence of infeasible statements, that is, dead code. Whether a piece of code is dead code is undecidable [Weyuker 1979a; Weyuker 1979b; White 1981]. Because statements correspond to nodes in flow-graph models, this criterion can be defined in terms of flow graphs, as follows.

Definition 2.1 (*Statement Coverage Criterion*). A set *P* of execution paths

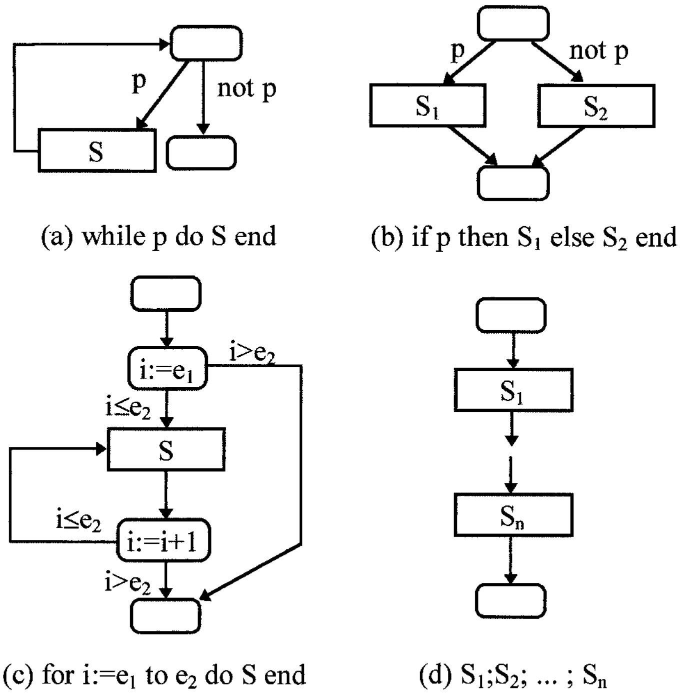

Figure 2. Example flow graphs for structured statements.

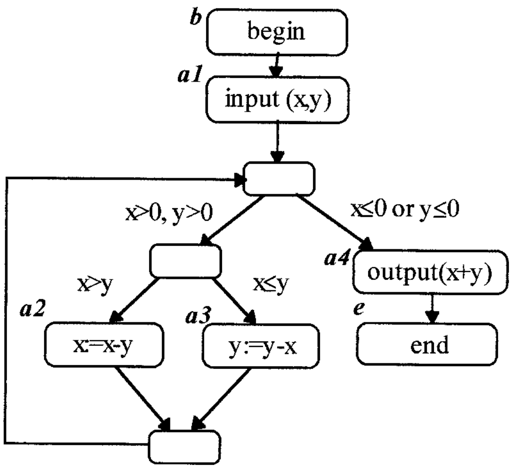

Figure 3. Flow graph for Example 2.1.

satisfies the *statement coverage criterion* if and only if for all nodes n in the flow graph, there is at least one path p in P such that node n is on the path p.

Notice that statement coverage is so weak that even some control transfers may be missed from an adequate test. Hence, we have a slightly stronger requirement of adequate test, called *branch coverage* [Hetzel 1984], that all control transfers must be checked. Since control transfers correspond to edges in flow graphs, the branch coverage criterion can be defined as the coverage of all edges in the flow graph.

Definition 2.2 (*Branch Coverage Criterion*). A set P of execution paths satisfies the *branch coverage criterion* if and only if for all edges e in the flow graph, there is at least one path p in P such that p contains the edge e.

Branch coverage is stronger than statement coverage because if all edges in a flow graph are covered, all nodes are necessarily covered. Therefore, a test set that satisfies the branch coverage criterion must also satisfy statement coverage. Such a relationship between adequacy criteria is called the *subsumes* relation. It is of interest in the comparison of software test adequacy criteria (see details in Section 5.1.3).

However, even if all branches are exercised, this does not mean that all combinations of control transfers are checked. The requirement of checking all combinations of branches is usually called path coverage or path testing, which can be defined as follows.

Definition 2.3 (*Path Coverage Criterion*). A set P of execution paths satisfies the *path coverage criterion* if and only if P contains all execution paths from the *begin* node to the *end* node in the flow graph.

Although the path coverage criterion still cannot guarantee the correctness of a tested program, it is too strong to be practically useful for most programs, because there can be an infinite number of different paths in a program with loops. In such a case, an infinite set of test data must be executed for adequate testing. This means that the testing cannot finish in a finite period of time. But, in practice, software testing must be fulfilled within a limited fixed period of time. Therefore, a test set must be finite. The requirement that an adequacy criterion can always be satisfied by a finite test set is called *finite applicability* [Zhu and Hall 1993] (see Section 6).

The statement coverage criterion and branch coverage criterion are not finitely applicable either, because they require testing to cover infeasible elements. For instance, statement coverage requires that all the statements in a program are executed. However, a program may have infeasible statements, that is, dead code, so that no input data can cause their execution. Therefore, in such cases, there is no adequate test set that can satisfy statement coverage. Similarly, branch coverage is not finitely applicable because a program may contain infeasible branches. However, for statement coverage and also branch coverage, we can define a finitely applicable version of the criterion by requiring testing only to cover the feasible elements. Most program-based adequacy criteria in the literature are not finitely applicable, but finitely applicable versions can often be obtained by redefinition in this way. Subsequently, such a version is called the *feasible* version of the adequacy criterion. It should be noted, first, that although we can often obtain finite applicability by using the feasible version, this may cause the *undecidability* problem; that is, we may not be able to decide whether a test set satisfies a given adequacy criterion. For example, whether a statement in a program is feasible is undecidable [Weyuker 1979a; Weyuker 1979b; White 1991]. Therefore, when a test set does not cover all the statements in a program, we may not be able to decide whether a statement not covered by the test data is dead code. Hence, we may not be able to decide if the test set satisfies the feasible version of statement coverage. Second, for some adequacy criteria, such as path coverage, we cannot obtain finite applicability by such a redefinition.

Recall that the rationale for path coverage is that there is no path that does not need to be checked by testing, while finite applicability forces us to select a finite subset of paths. Thus, research into flow-graph-based adequacy criteria has focused on the selection of the most important subsets of paths. Probably the most straightforward solution to the conflict is to select paths that contain no redundant information. Hence, two notions from graph theory can be used. First, a path that has no repeated occurrence of any edge is called a simple path in graph theory. Second, a path that has no repeated occurrences of any node is called an elementary path. Thus, it is possible to define simple path coverage and elementary path coverage criteria, which require that adequate test sets should cover all simple paths and elementary paths, respectively.

These two criteria are typical ones that select finite subsets of paths by specifying restrictions on the complexity of the individual paths. Another example of this type is the *length-n path coverage* criterion, which requires coverage of all subpaths of length less than or equal to n [Gourlay 1983]. A more complicated example of the type is Paige's *level-i path coverage* criterion [Paige 1978; Paige 1975]. Informally, the criterion starts with testing all elementary paths from the begin node to the end node. Then, if there is an elementary subpath or cycle that has not been exercised, the subpath is required to be checked at next level. This process is repeated until all nodes and edges are covered by testing. Obviously, a test set that satisfies the level-i path coverage criterion must also satisfy the elementary path coverage criterion, because elementary paths are level-0 paths.

A set of control-flow adequacy criteria that are concerned with testing loops is loop count criteria, which date back to the mid-1970s [Bently et al. 1993]. For any given natural number K, the loop count-K criterion requires that every loop in the program under test should be executed zero times, once, twice, and so on, up to K times [Howden 1975]. Another control-flow criterion concerned with testing loops is the cycle combination criterion, which requires that an adequate test set should cover all execution paths that do not contain a cycle more than once.

An alternative approach to defining control-flow adequacy criteria is to specify restrictions on the redundancy among the paths. McCabe's cyclomatic measurement is such an example [McCabe 1976; McCabe 1983; McCabe and Schulmeyer 1985]. It is based on the theorem of graph theory that for any flow graph there is a set of execution paths such that every execution path can be expressed as a linear combination of them. A set of paths is independent if none of them is a linear combination of the others. According to McCabe, a path should be tested if it is independent of the paths that have been tested. On the other hand, if a path is a linear combination of tested paths, it can be considered redundant. According to graph theory, the maximal size of a set of independent paths is unique for any given graph and is called the cyclomatic number, and can be easily calculated by the following formula.

$$v(G) = e - n + p,$$

where $v(G)$ denotes the cyclomatic number of the graph G, n is the number of vertices in G, e is the number of edges, and p is the number of strongly connected components.[3] The adequacy criterion is then defined as follows.

Definition 2.4 (*Cyclomatic-Number Criterion*). A set P of execution paths satisfies the *cyclomatic number criterion* if and only if P contains at least one set of v independent paths, where $v = e - n + p$ is the cyclomatic number of the flow graph.

McCabe also gave an algorithm to generate a set of independent paths from any given flow graph [McCabe 1976]. Paige [1978] has shown that the level-i path coverage criterion subsumes McCabe's cyclomatic number criterion.

The preceding control-flow test adequacy criteria are all defined in terms of flow-graph models of program structure except loop count criteria, which are defined in terms of program text. A number of other test adequacy criteria are based on the text of program. One of the most popular criteria in software testing practice is the so-called multiple condition coverage discussed in Myers' [1979] classic book which has proved popular in commercial software testing practice. The criterion focuses on the conditions of control transfers, such as the condition in an IF-statement or a WHILE-LOOP statement. A test set is said of satisfying the decision coverage criterion if for every condition there is at least one test case such that the condition has value true when evalu-

[3] A graph is strongly connected if, for any two nodes a and b, there exists a path from a to b and a path from b to a. Strongly connected components are maximal strongly connected subgraphs.

ated, and there is also at least one test case such that the condition has value false. In a high-level programming language, a condition can be a Boolean expression consisting of several atomic predicates combined by logic connectives like *and*, *or*, and *not*. A test set satisfies the condition coverage criterion if for every atomic predicate there is at least one test case such that the predicate has value true when evaluated, and there is also at least one test case such that the predicate has value false. Although the result value of an evaluation of a Boolean expression can only be one of two possibilities, *true* or *false*, the result may be due to different combinations of the truth values of the atomic predicates. The multiple condition coverage criterion requires that a test set should cover all possible combinations of the truth values of atomic predicates in every condition. This criterion is sometimes also called extended branch coverage in the literature. Formally:

Definition 2.5 (*Multiple Condition Coverage*). A test set T is said to be adequate according to the *multiple-condition-coverage criterion* if, for every condition C, which consists of atomic predicates $(p_1, p_2, \ldots, p_n)$, and all the possible combinations $(b_1, b_2, \ldots, b_n)$ of their truth values, there is at least one test case in T such that the value of p_i equals b_i, $i = 1, 2, \ldots, n$.

Woodward et al. [1980] proposed and studied a hierarchy of program-text-based test data adequacy criteria based on a class of program units called *linear code sequence and jump* (LCSAJ). These criteria are usually referred to as test effectiveness metrics in the literature. An LCSAJ consists of a body of code through which the flow of control may proceed sequentially and which is terminated by a jump in the control flow. The hierarchy TER_i, $i = 1, 2, \ldots, n, \ldots$ of criteria starts with statement coverage as the lowest level, followed by branch coverage as the next lowest level. They are denoted by TER_1 and TER_2, respectively, where TER represents *test effectiveness ratio*. The coverage of LCSAJ is the third level, which is defined as TER_3. The hierarchy is then extended to the coverage of program paths containing a number of LCSAJs.

Definition 2.6 (*TER*$_3$: *LCSAJ Coverage*)

$$\text{TER}_3 = \frac{\text{number of LCSAJs exercised at least once}}{\text{total number of LCSAJs.}}$$

Generally speaking, an advantage of text-based adequacy criteria is that test adequacy can be easily calculated from the part of program text executed during testing. However, their definitions may be sensitive to language details. For programs written in a structured programming language, the application of TER_n for n greater than or equal to 3 requires analysis and reformatting of the program structure. In such cases, the connection between program text and its test adequacy becomes less straightforward. In fact, it is observed in software testing practice that a small modification to the program may result in a considerably different set of linear sequence code and jumps.

It should be noted that none of the adequacy criteria discussed in this section are applicable due to the possible existence of infeasible elements in a program, such as infeasible statements, infeasible branches, infeasible combinations of conditions, and the like. These criteria, except path coverage, can be redefined to obtain finite applicability by only requiring the coverage of feasible elements.

2.1.2 *Data-Flow-Based Test Data Adequacy Criteria.* In the previous section, we have seen how control-flow information in the program under test is used to specify testing requirements. In this section, data-flow information is taken into account in the definition of testing requirements. We first introduce the way that data-flow information is added into the flow-graph models of program structures. Then, three basic groups of data-flow adequacy criteria are reviewed. Finally, their limitations and extensions are discussed.

A. Data-flow information in flow graph. Data-flow analysis of test adequacy is concerned with the coverage of flow-graph paths that are significant for the data flow in the program. Therefore, data-flow information is introduced into the flow-graph models of program structures.

Data-flow testing methods are based on the investigation of the ways in which values are associated with variables and how these associations can effect the execution of the program. This analysis focuses on the occurrences of variables within the program. Each variable occurrence is classified as either a definition occurrence or a use occurrence. A definition occurrence of a variable is where a value is bound to the variable. A use occurrence of a variable is where the value of the variable is referred. Each use occurrence is further classified as being a computational use or a predicate use. If the value of a variable is used to decide whether a predicate is true for selecting execution paths, the occurrence is called a predicate use. Otherwise, it is used to compute a value for defining other variables or as an output value. It is then called a computational use. For example, the assignment statement "$y := x_1 + x_2$" contains computational uses of x_1 and x_2 and a definition of y. The statement "if $x_1 < x_2$ then goto L endif" contains predicate uses of x_1 and x_2.

Since we are interested in tracing the flow of data between nodes, any definition that is used only within the node in which the definition occurs is of little importance. Therefore a distinction is made between local computational uses and global computational uses. A global computational use of a variable x is where no definition of x precedes the computational use within the node in which it occurs. That is, the value must have been bound to x in some node other than the one in which it is being used. Otherwise it is a local computational use.

Data-flow test adequacy analysis is concerned with subpaths from definitions to nodes where those definitions are used. A definition-clear path with respect to a variable x is a path where for all nodes in the path there is no definition occurrence of the variable x. A definition occurrence of a variable x at a node u reaches a computational use occurrence of the variable at node v if and only if there is a path p from u to v such that $p = (u, w_1, w_2, \ldots, w_n, v)$, and $(w_1, w_2, \ldots, w_n)$ is definition-clear with respect to x, and the occurrence of x at v is a global computational use. We say that the definition of x at u reaches the computational occurrence of x at v through the path p. Similarly, if there is a path $p = (u, w_1, w_2, \ldots, w_n, v)$ from u to v, and $(w_1, w_2, \ldots, w_n)$ is definition-clear with respect to x, and there is a predicate occurrence of x associated with the edge from w_n to v, we say that u reaches the predicate use of x on the edge (w_n, v) through the path p. If a path in one of the preceding definitions is feasible, that is, there is at least one input datum that can actually cause the execution of the path, we say that a definition feasibly reaches a use of the definition.

Three groups of data-flow adequacy criteria have been proposed in the literature, and are discussed in the following.

B. Simple definition-use association coverage—the Rapps-Weyuker-Frankl family. Rapps and Weyuker [1985] proposed a family of testing adequacy criteria based on data-flow information. Their criteria are concerned mainly with the simplest type of data-flow paths that start with a definition of a variable and end with a use of the same variable. Frankl and Weyuker [1988] later reexamined the data-flow adequacy criteria and found that the original definitions of the criteria did not satisfy the applicability condition. They redefined the criteria to be applicable. The following definitions come from the modified definitions.

The all-definitions criterion requires that an adequate test set should cover all definition occurrences in the sense that, for each definition occurrence, the testing paths should cover a path through which the definition reaches a use of the definition.

Definition 2.7 (*All Definitions Criterion*). A set P of execution paths satisfies the *all-definitions criterion* if and only if for all definition occurrences of a variable x such that there is a use of x which is feasibly reachable from the definition, there is at least one path p in P such that p includes a subpath through which the definition of x reaches some use occurrence of x.

Since one definition occurrence of a variable may reach more than one use occurrence, the all-uses criterion requires that all of the uses should be exercised by testing. Obviously, this requirement is stronger than the all-definition criterion.

Definition 2.8 (*All Uses Criterion*). A set P of execution paths satisfies the *all-uses criterion* if and only if for all definition occurrences of a variable x and all use occurrences of x that the definition feasibly reaches, there is at least one path p in P such that p includes a subpath through which that definition reaches the use.

The all-uses criterion was also proposed by Herman [1976], and called *reach-coverage* criterion. As discussed at the beginning of the section, use occurrences are classified into computational use occurrences and predicate use occurrences. Hence, emphasis can be put either on computational uses or on predicate uses. Rapps and Weyuker [1985] identified four adequacy criteria of different strengths and emphasis. The all-c-uses/some-p-uses criterion requires that all of the computational uses are exercised, but it also requires that at least one predicate use should be exercised when there is no computational use of the variable. In contrast, the all-p-uses/some-c-uses criterion puts emphasis on predicate uses by requiring that test sets should exercise all predicate uses and exercise at least one computational use when there is no predicate use. Two even weaker criteria were also defined. The all-predicate-uses criterion completely ignores the computational uses and requires that only predicate uses need to be tested. The all-computation-uses criterion only requires that computational uses should be tested and ignores the predicate uses.

Notice that, given a definition occurrence of a variable x and a use of the variable x that is reachable from that definition, there may exist many paths through which the definition reaches the use. A weakness of the preceding criteria is that they require only one of such paths to be exercised by testing. However, the applicability problem arises if all such paths are to be exercised because there may exist an infinite number of such paths in a flow graph. For example, consider the flow graph in Figure 1, the definition of y at node $a1$ reaches the use of y at node $a3$ through all the paths in the form:

$$(a1, a2) \wedge (a2, a2)^n \wedge (a2, a3), \quad n \geq 1,$$

where $\wedge$ is the concatenation of paths, p^n is the concatenation of p with itself for n times, which is inductively defined to be $p^1 = p$ and $p^k = p \wedge p^{k-1}$, for all $k > 1$. To obtain finite applicability, Frankl and Weyuker [1988] and Clarke et al. [1989] restricted the paths to be cycle-free or only the end node of the path to be the same as the start node.

Definition 2.9 (*All Definition-Use-Paths Criterion: Abbr. All DU-Paths Criterion*). A set P of execution paths satisfies the *all-du-paths* criterion if and only if for all definitions of a variable x and all paths q through which that definition reaches a use of x, there is at least one path p in P such that q is a subpath of p, and q is cycle-free or contains only simple cycles.

However, even with this restriction, it is still not applicable since such a path may be infeasible.

C. *Interactions between variables—the Ntafos required K-tuples criteria.* Ntafos [1984] also used data-flow information to analyze test data adequacy. He studied how the values of different variables interact, and defined a family of adequacy criteria called required k-tuples, where $k > 1$ is a natural number. These criteria require that a path set *cover* the chains of alternating definitions and uses, called definition-reference interactions (abbr. k–dr interactions) in Ntafos' terminology. Each definition in a k–dr interaction reaches the next use in the chain, which occurs at the same node as the next definition in the chain. Formally:

Definition 2.10 (*k–dr interaction*). For $k > 1$, a *k–dr interaction* is a sequence $K = [d_1(x_1), u_1(x_1), d_2(x_2), u_2(x_2), \ldots, d_k(x_k), u_k(x_k)]$ where

(i) $d_i(x_i)$, $1 \leq i < k$, is a definition occurrence of the variable x_i;
(ii) $u_i(x_i)$, $1 \leq i < k$, is a use occurrence of the variable x_i;
(iii) the use $u_i(x_i)$ and the definition $d_{i+1}(x_i)$ are associated with the same node n_{i+1};
(iv) for all i, $1 \leq i < k$, the ith definition $d_i(x_i)$ reaches the ith use $u_i(x_i)$.

Note that the variables $x_1, x_2, \ldots, x_k$ and the nodes $n_1, n_2, \ldots, n_k$ need not be distinct. This definition comes from Ntafos' [1988] later work. It is different from the original definition where the nodes are required to be distinct [Ntafos 1984]. The same modification was also made by Clark et al. [1989] in their formal analysis of data flow adequacy criteria.

An interaction path for a k–dr interaction is a path $p = (n_1) * p_1 * (n_2) * \ldots * (n_{k-1}) * p_{k-1} * (n_k)$ such that for all $i = 1, 2, \ldots, k - 1$, $d_i(x_i)$ reaches $u_i(x_i)$ through p_i. The required k-tuples criterion then requires that all k–dr interactions are tested.

Definition 2.11 (*Required k-Tuples Criteria*). A set P of execution paths satisfies the required k-tuples criterion, $k > 1$, if and only if for all j–dr interactions L, $1 < j \leq k$, there is at least one path p in P such that p includes a subpath which is an interaction path for L.

Example 2.1 Consider the flow graph in Figure 2. The following are 3–dr interaction paths.

$(a1, a3, a2, a4)$ for the 3–dr interaction $[d_1(x), u_1(x), d_2(y), u_2(y), d_3(x), u_3(x)]$; and

$(a1, a2, a3, a4)$ for the 3–dr interaction $[d_1(y), u_1(y), d_2(x), u_2(x), d_3(y), u_3(y)]$.

D. *Combinations of definitions—the Laski-Korel criteria.* Laski and Korel [1983] defined and studied another kind of testing path selection criteria based on data-flow analysis. They observed that a given node may contain uses of several different variables, where each use may be reached by several definitions occurring at different nodes. Such definitions constitute the context of the computation at the node. Therefore, they are concerned with exploring such contexts for each node by selecting paths along which the various combinations of definitions reach the node.

Definition 2.12 (*Ordered Context*). Let n be a node in the flow graph. Suppose that there are uses of the variables $x_1, x_2, \ldots, x_m$ at the node n.[4] Let

[4] This assumption comes from Clark et al. [1989]. The original definition given by Laski and Korel

$[n_1, n_2, \ldots, n_m]$ be a sequence of nodes such that for all $i = 1, 2, \ldots, m$, there is a definition of x_i on node n_i, and the definition of x_i reaches the node n with respect to x_i. A path $p = p_1 * (n_1) * p_2 * (n_2) * \ldots * p_m * (n_m) * p_{m+1} * (n)$ is called an *ordered context path* for the node n with respect to the sequence $[n_1, n_2, \ldots, n_m]$ if and only if for all $i = 2, 3, \ldots, m$, the subpath $p_i * (n_i) * p_{i+1} * \ldots * p_{m+1}$ is definition clear with respect to x_{i-1}. In this case, we say that the sequence $[n_1, n_2, \ldots, n_m]$ of nodes is an *ordered context* for n.

Example 2.2 Consider the flow graph in Figure 1. There are uses of the two variables x and y at node $a4$. The node sequences $[a1, a2]$, $[a1, a3]$, $[a2, a3]$, and $[a3, a2]$ are ordered contexts for node $a4$. The paths $(a1, a2, a4)$, $(a1, a3, a4)$, $(a2, a3, a4)$, and $(a3, a2, a4)$ are the ordered context paths for them, respectively.

The ordered context coverage requires that an adequate test set should cover all ordered contexts for every mode.

Definition 2.13 (*Ordered-Context Coverage Criterion*). A set P of execution paths satisfies the ordered-context coverage criterion if and only if for all nodes n and all ordered contexts c for n, there is at least one path p in P such that p contains a subpath which is an ordered context path for n with respect to c.

Given a node n, let $\{x_1, x_2, \ldots, x_m\}$ be a nonempty subset of the variables that are used at the node n, the nodes n_i, $i = 1, 2, \ldots, m$, have definition occurrences of the variables x_i, that reach the node n. If there is a permutation σ of the nodes which is an ordered context for n, then we say that the set $\{n_1, n_2, \ldots, n_m\}$ is a context for n, and an ordered context path for n with respect to σ is also called a definition context path for n with respect to the context $\{n_1, n_2, \ldots, n_m\}$. Ignoring the ordering between the nodes, a slightly weaker criterion, called the context-coverage criterion, requires that all contexts for all nodes are covered.

[1983] defines a context to be formed from all variables having a definition that reaches the node.

Definition 2.14 (*Context-Coverage Criterion*). A set P of execution paths satisfies the context coverage criterion if and only if for all nodes n and for all contexts for n, there is at least one path p in P such that p contains a subpath which is a definition context path for n with respect to the context.

E. *Data-flow testing for structured data and dynamic data.* The data flow testing methods discussed so far have a number of limitations. First, they make no distinction between atomic data such as integers and structured or aggregate data such as arrays and records. Modifications and references to an element of a structured datum are regarded as modifications and references to the whole datum. It was argued that treating structured data, such as arrays, as aggregate values may lead to two types of mistakes [Hamlet et al. 1993]. A commission mistake may happen when a definition-use path is identified but it is not present for any array elements. An omission mistake may happen when a path is missed because of a false intermediate assignment. Such mistakes occur frequently even in small programs [Hamlet et al. 1993]. Treating elements of structured data as independent data can correct the mistakes. Such an extension seems to add no complexity when the references to the elements of structured data are static, such as the fields of records. However, treating arrays element-by-element may introduce a potential infinity of definition-use paths to be tested. Moreover, theoretically speaking, whether two references to array elements are references to the same element is undecidable. Hamlet et al. [1993] proposed a partial solution to this problem by using symbolic execution and a symbolic equation solver to determine whether two occurrences of array elements can be the occurrences of the same element.

The second limitation of the data-flow testing methods discussed is that dynamic data were not taken into account. One of the difficulties in the data-flow analysis of dynamic data such as those referred to by pointers is that a pointer variable may actually refer to a number of data storage. On the other hand, a data storage may have a number of references to it, that is, the existence of alias. Therefore, for a given variable V, a node may contain a *definite definition* to the variable if a new value is definitely bound to the variable at the node. It has a possible definition at a node n if it is possible that a new value is bound to it at the node. Similarly, a path may be *definitely definition-clear* or *possibly definition-clear* with respect to a variable. Ostrand and Weyuker [1991] extended the definition-use association relation on the occurrences of variables to a hierarchy of relations. A definition-use association is *strong* if there is a definite definition of a variable and a definite use of the variable and every definition-clear path from the definition to the use is definitely definition-clear with respect to the variable. The association is *firm* if both the definition and the use are definite and there is at least one path from the definition to the use that it is definitely definition-clear. The association is *weak* if both the definition and the use are definite, but there is no path from the definition to the use which is definitely definition-clear. An association is *very weak* if the definition or the use or both of them are possible instead of definite.

F. *Interprocedural data-flow testing*. The data-flow testing methods discussed so far have also been restricted to testing the data dependence existing within a program unit, such as a procedure. As current trends in programming encourage a high degree of modularity, the number of procedure calls and returns executed in a module continues to grow. This mandates the efficient testing of the interaction between procedures. The basic idea of interprocedural data-flow testing is to test the data dependence across procedure interfaces. Harrold and Soffa [1990; 1991] identified two types of interprocedural data dependences in a program: direct data dependence and indirect data dependence. A *direct data dependence* is a definition-use association whose definition occurs in procedure P and use occurs in a directly called procedure Q of P. Such a dependence exists when (1) a definition of an actual parameter in one procedure reaches a use of the corresponding formal parameter at a call site (i.e., a procedure call); (2) a definition of a formal parameter in a called procedure reaches a use of the corresponding actual parameter at a return site (i.e., a procedure return); or (3) a definition of a global variable reaches a call or return site. An *indirect data dependence* is a definition-use association whose definition occurs in procedure P and use occurs in an indirectly called procedure Q of P. Conditions for indirect data dependence are similar to those for direct data dependence, except that multiple levels of procedure calls and returns are considered. Indirect data dependence can be determined by considering the possible uses of definitions along the calling sequences. When a formal parameter is passed as an actual parameter at a call site, an indirect data dependence may exist. Given this data dependence information, the data-flow test adequacy criteria can be easily extended for interprocedural data-flow testing. Harrold and Soffa [1990] proposed an algorithm for computing the interprocedural data dependences and developed a tool to support interprocedural data-flow testing.

Based on Harrold and Soffa's work, Ural and Yang [1988; 1993] extended the flow-graph model for accurate representation of interprocedural data-flow information. Pande et al. [1991] proposed a polynomial-time algorithm for determining interprocedural definition-use association including dynamic data of single-level pointers for C programs.

2.1.3 *Dependence Coverage Criterion—an Extension and Combination of Data-Flow and Control-Flow Testing.* An extension of data-flow testing methods was made by Podgurski and Clarke [1989; 1990] by generalizing control and data dependence. Informally, a statement s is *semantically dependent* on a statement s' if the function computed by s' affects the execution behavior of s. Podgurski and Clarke then proposed a necessary condition of semantic dependence called *weak syntactic dependence* as a generalization of data dependence. There is a weak syntactic dependence between two statements if there is a chain of data flow and a weak control dependence between the statements, where a statement u is weakly control-dependent on statement v if v has successors v' and v'' such that if the branch from v to v' is executed then u is necessarily executed within a fixed number of steps, whereas if the branch v to v'' is taken then u can be bypassed or its execution can be delayed indefinitely. Podgurski and Clarke also defined the notion of *strong syntactic dependence*: there is a strong syntactic dependence between two statements if there is a chain of data flow and a strong control dependence between the statements. Roughly speaking, a statement u is strongly control dependent on statement v if v has two successors v' and v'' such that the execution through the branch v to v' may result in the execution of u, but u may be bypassed when the branch from v to v'' is taken. Podgurski and Clarke proved that strong syntactic dependence is not a necessary condition of semantic dependence.

When the definition-use association relation is replaced with various dependence relations, various dependence-coverage criteria can be obtained as extensions to the data-flow test adequacy criteria. Such criteria make more use of semantic information contained in the program under test. Furthermore, these dependence relations can be efficiently calculated.

2.2 Specification-Based Structural Testing

There are two main roles a specification can play in software testing [Richardson et al. 1992]. The first is to provide the necessary information to check whether the output of the program is correct [Podgurski and Clarke 1989; 1990]. Checking the correctness of program outputs is known as the oracle problem. The second is to provide information to select test cases and to measure test adequacy. As the purpose of this article is to study test adequacy criteria, we focus on the second use of specifications.

Like programs, a specification has two facets, syntactic structure and semantics. Both of them can be used to select test cases and to measure test adequacy. This section is concerned with the syntactic structure of a specification.

A specification specifies the properties that the software must satisfy. Given a particular instance of the software's input and its corresponding output, to check whether the instance of the software behavior satisfies these properties we must evaluate the specification by substituting the instance of input and output into the input and output variables in the specification, respectively. Although this evaluation process may take various forms, depending on the type of the specification, the basic idea behind the approach is to consider a particular set of elements or components in the specification and to calculate the proportion of such elements or components involved in the evaluation.

There are two major approaches to formal software functional specifications, model-based specifications and property-oriented specifications such as axiomatic or algebraic specifications. The following discussion is based on these types of specifications.

2.2.1 *Coverage of Model-Based Formal Functional Specifications.* When a specification is model-based, such as those written in Z and VDM, it has two parts. The first describes the state space of the software, and the second part specifies the required operations on the space. The state space of the software system can be defined as a set of typed variables with a predicate to describe the invariant property of the state space. The operations are functions mapping from input data and the state before the operation to the output data and the state after the operation. Such operations can be specified by a set of predicates that give the precondition, that is, the condition on the input data and the state before the operation, and postconditions that specify the relationship between the input data, output data, and the states before and after the operation.

The evaluation of model-based formal functional specifications is fairly similar to the evaluation of a Boolean expression in an imperative programming language. When input and output variables in the expression are replaced with an instance of input data and program outputs, each atomic predicate must be either true or false. If the result of the evaluation of the whole specification is true, then the correctness of the software on that input is confirmed. Otherwise, a program error is found. However, the same truth value of a specification on two instances of input/output may be due to different combinations of the truth values of the atomic predicates. Therefore it is natural to require that an adequate test cover a certain subset of feasible combinations of the predicates. Here a feasible combination means that the combination can be satisfied; that is, there is an assignment of values to the input and output variables such that the atomic predicates take their corresponding values in the predicate combination. In the case where the specification contains nondeterminism, the program may be less nondeterministic than the specification. That is, some of the choices of output allowed by the specification may not be implemented by the program. This may not be considered a program error, but it may result in infeasible combinations.

A feasible combination of the atomic predicates in the preconditions is a description of the conditions that test cases should satisfy. It specifies a subdomain of the input space. It can be expressed in the same specification language. Such specifications of testing requirements are called test templates. Stocks and Carrington [1993] suggested the use of the formal functional specification language Z to express test templates because the schema structure of Z and its schema calculus can provide support to the derivation and refinement of test templates according to formal specifications and heuristic testing rules. Methods have also been proposed to derive such test templates from model-based specification languages. Amla and Ammann [1992] described a technique to extract information from formal specifications in Z and to derive test templates written in Z for partition testing. The key step in their method is to identify the categories of the test data for each parameter and environment variable of a functional unit under test. These categories categorize the input domain of one parameter or one environment variable according to the major characteristics of the input. According to Amla and Ammann, there are typically two distinct sources of categories in Z specifications: (a) characteristics enumerated in the preconditions and (b) characteristics of a parameter or environment variable by itself. For parameters, these characteristics are based on their type. For environment variables, these characteristics may also be based on the invariant for the state components. Each category is then further divided into a set of choices. A *choice* is a subset of data that can be assigned to the parameter or the environment variable. Each category can be broken into at least two choices: one for the valid inputs and the other for the invalid inputs. Finer partitions of valid inputs are derived according to the syntactic structure of the precondition predicate, the parameters, or the invariant predicate of the environment variables. For example, the predicate "$A \vee B$" is partitioned into three choices: (a) "$\neg (A \vee B)$" for the set of data which are invalid inputs; (b) "A" for the subset of valid inputs which satisfy condition A; (3) "B" for the subset of valid inputs which satisfy the condition B.

Based on Amla and Ammann's [1992] work, Ammann and Offutt [1994] recently considered how to test a functional unit effectively and efficiently by selecting test cases to cover various subsets of the combinations of the categories and choices when the functional unit has more than one parameter and environment variable. They proposed three coverage criteria. The *all-combinations* criterion requires that software is tested on all combinations of choices; that is, for each combination of the choices of the parameters and the environment variables, there is at least one test datum in the combination. Let $x_1, x_2, \ldots, x_n$ be the parameters and environment variables of the functional unit under test. Suppose that the choices for x_i are $A_{i,1}, A_{i,2}, \ldots, A_{i,k_i}$, $k_i > 0$, $i = 1, 2, \ldots, n$. Let $C = \{A_{1,u_1} \times A_{2,u_2} \times \ldots \times A_{n,u_n} \mid 1 \le u_i \le k_i \text{ and } 1 \le i \le n\}$. C is then the set of all combinations of choices. The all combination criterion can be formally defined as follows.

Definition 2.15 (*All-Combination Criterion*). A set of test data T satisfies the *all-combination* criterion if for all $c \in C$, there exists at least one $t \in T$ such that $t \in c$.

This criterion was considered to be inefficient, and the each-choice-used criterion was considered ineffective [Ammann and Offutt 1994]. The each-choice-used criterion requires that each choice is used in the testing; that is, for each choice of each parameter or environment variable, there is at least one test datum that belongs to the choice. Formally:

Definition 2.16 (*Each-Choice-Used Criterion*). A set of test data T satisfies the *each-choice-used* criterion if the subset $E = \{e \mid e \in C \text{ and } \exists t \in T.(t \in e)\}$ satisfies the condition:

$$\forall i.(1 \le i \le n \Rightarrow (E_i = \{A_{i,1}, A_{i,2}, \ldots, A_{i,k_i}\})),$$

where

$$E_i = \{e \mid \exists X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n.(X_1 \times \ldots \times X_{i-1} \times e \times X_{i+1} \times \ldots \times X_n \in E)\}.$$

Ammann and Offutt suggested the use of the base-choice-coverage criterion and described a technique to derive test templates that satisfy the criterion. The base-choice-coverage criterion is based on the notion of base choice, which is a combination of the choices of parameters and environment variables that represents the normal operation of the functional unit under test. Therefore, test cases of the base choice are useful to evaluate the function's behavior in normal operation mode. To satisfy the base-choice-coverage criterion, software needs to be tested on the subset of combinations of choices such that for each choice in a category, the choice is combined with the base choices for all other categories. Assume that $A_{1,1} \times A_{2,1} \times \ldots \times A_{n,1}$ is the base choice. The base-choice coverage criterion can be formally defined as follows.

Definition 2.17 (*Base-Choice-Coverage Criterion*). A set of test data T satisfies the base-choice-coverage criterion if the subset $E = \{e \mid e \in C \wedge \exists t \in T.(t \in e)\}$ satisfies the following condition:

$$E \supseteq \bigcup_{i=1}^{n} B_i,$$

where

$$B_i = \{A_{1,1} \times \ldots \times A_{i-1,1} \times A_{i,j} \times A_{i+1,1} \times \ldots \times A_{n,1} | j = 1, 2, \ldots, k_i\}.$$

There are a number of works on specification-based testing that focus on derivation of test cases from specifications, including Denney's [1991] work on test-case generation from Prolog-based specifications and many others [Hayes 1986; Kemmerer 1985; McMullin and Gannon 1983; Wild et al. 1992].

Model-based formal specification can also be in an executable form, such as a finite state machine or a state chart. Aspects of such models can be represented in the form of a directed graph. Therefore, the program-based adequacy criteria based on the flow-graph model can be adapted for specification-based testing [Fujiwara et al. 1991; Hall and Hierons 1991; Ural and Yang 1988; 1993].

2.2.2 *Coverage of Algebraic Formal Functional Specifications.* Property-oriented formal functional specifications specify software functions by a set of properties that the software should possess. In particular, an algebraic specification consists of a set of equations that the operations of the software must satisfy. Therefore checking if a program satisfies the specification means checking whether all of the equations are satisfied by the program.

An equation in an algebraic specification consists of two terms as two sides of the equation. A term is constructed from three types of symbols: variables representing arbitrary values of a given data type, constants representing a given data value in a data type, and operators representing data constructors and operations on data types.

Each term has two interpretations in the context of testing. First, a term represents a sequence of calls to the operations that implement the operators specified in the specification. When the variables in the term are replaced with constants, such a sequence of calls to the operations represents a test execution of the program, where the test case consists of the constants substituted for the variables. Second, a term also represents a value, that is, the result of the sequence of operations. Therefore, checking an equation means executing the operation sequences for the two terms on the two sides of the equation and then comparing the results. If the results are the same or equivalent, the program is considered to be correct on this test case, otherwise the implementation has errors. This interpretation allows the use of algebraic specifications as test oracles.

Since variables in a term can be replaced by any value of the data type, there is a great deal of freedom to choose input data for any given sequence of operations. For algebraic specification, values are represented by ground terms, that is, terms without variables. Gaudel [Bouge et al. 1986; Bernot et al. 1991] and her colleagues suggested that the selection of test cases should be based on partitioning the set of ground terms according to their complexity so that the regularity and uniformity hypotheses on the subsets in the partition can be assumed. The complexity of a test case is then the depth of nesting of the operators in the ground term. Therefore, roughly speaking, the selection of test cases should first consider constants specified by the specification, then all the values generated by one application of operations on constants, then values generated by two applications on constants, and so on until the test set covers data of a certain degree of complexity.

The following hypothesis, called the *regularity hypothesis* [Bouge et al. 1986; Bernot et al. 1991], formally states the gap between software correctness and adequate testing by the preceding approach.

Regularity Hypothesis

$$\forall x(complexity(x) \leq K \Rightarrow t(x) = t'(x)) \Rightarrow \forall x(t(x) = t'(x)) \quad (2.1)$$

Informally, the regularity hypothesis assumes that for some complexity degree k, if a program satisfies an equation $t(x) = t'(x)$ on all data of complexity not higher than k, then the program satisfies the equation on all data. This hypothesis captures the intuition of inductive reasoning in software testing. But it cannot be proved formally (at least in its most general form) nor validated empirically. Moreover, there is no way to determine the complexity k such that only the test cases of complexity less than k need to be tested.

2.3 Summary of Structure Coverage Criteria

In summary, when programs are modeled as directed graphs, paths in the flow graph should be exercised by testing. However, only a finite subset of the paths can be checked during testing. The problem is therefore to choose which paths should be exercised.

Control-flow test data adequacy criteria answer this question by specifying restrictions on the complexity of the paths or by specifying restrictions on the redundancy among paths. Data-flow adequacy criteria use data-flow information in the program and select paths that are significant with respect to such information. Data-flow and control-flow adequacy criteria can be extended to dependence coverage criteria, which make more use of semantic information contained in the program under test. However, none of these criteria use information about software requirements or functional specifications.

Specification-based structure coverage criteria specify testing requirements and measure test adequacy according to the extent that the test data cover the required functions specified in formal specifications. These criteria focus on the specification and ignore the program that implements the specification.

As discussed in Section 1, software testing should employ information contained in the program as well as information in the specification. A simple way to combine program-based structural testing with specification-based structure coverage criteria is to measure test adequacy with criteria from both approaches.

3. FAULT-BASED ADEQUACY CRITERIA

Fault-based adequacy criteria measure the quality of a test set according to its effectiveness or ability to detect faults.

3.1 Error Seeding

Error seeding is a technique originally proposed to estimate the number of faults that remain in software. By this method, artificial faults are introduced into the program under test in some suitable random fashion unknown to the tester. It is assumed that these artificial faults are representative of the inherent faults in the program in terms of difficulty of detection. Then, the software is tested and the inherent and artificial faults discovered are counted separately. Let r be the ratio of the number of artificial faults found to the number of total artificial faults. Then the number of inherent faults in the program is statistically predicted with maximum likelihood to be f/r, where f is the number of inherent faults found by testing.

This method can also be used to measure the quality of software testing. The ratio r of the number of artificial faults found to the total number of artificial faults can be considered as a measure of the test adequacy. If only a small proportion of artificial faults are found during testing, the test quality must be poor. In this sense, error seeding can show weakness in the testing.

An advantage of the method is that it is not restricted to measuring test quality for dynamic testing. It is applicable to any testing method that aims at finding errors or faults in the software.

However, the accuracy of the measure is dependent on how faults are introduced. Usually, artificial faults are manually planted, but it has been proved difficult to implement error seeding in practice. It is not easy to introduce artificial faults that are equivalent to inherent faults in difficulty of detection. Generally, artificial errors are much easier to find than inherent errors. In an attempt to overcome this problem, mutation testing introduces faults into a program more systematically.

3.2 Program Mutation Testing

3.2.1 *Principles of Mutation Adequacy Analysis.* *Mutation analysis* is proposed as a procedure for evaluating the degree to which a program is tested, that is, to measure test case adequacy [DeMillo et al. 1978; Hamlet 1977]. Briefly, the method is as follows. We have a program p and a test set t that has been generated in some fashion. The first step in mutation analysis is the construction of a collection of alternative programs that differ from the original program in some fashion. These alternatives are called *mutants* of the original program, a name borrowed from biology. Each mutant is then executed on each member of the test set t, stopping either when an element of t is found on which p and the mutant program produce different responses, or when t is exhausted.

In the former case we say that the mutant has died since it is of no further value, whereas in the latter case we say the mutant lives. These live mutants provide valuable information. A mutant may remain alive for one of the following reasons.

(1) The test data are inadequate.

If a large proportion of mutants live, then it is clear that on the basis of these test data alone we have no more reason to believe that p is correct than to believe that any of the live mutants are correct. In this sense, mutation analysis can clearly reveal a weakness in test data by demonstrating specific programs that are not ruled out by the test data presented. For example, the test data may not exercise the portion of the program that was mutated.

(2) The mutant is equivalent to the original program.

The mutant and the original program always produce the same output, hence no test data can distinguish between the two. Normally, only a small percentage of mutants are equivalent to the original program.

Definition 3.1 (*Mutation Adequacy Score*). The mutation adequacy of a set of test data is measured by an adequacy score computed according to the following equation.

$$\text{Adequacy Score } S = \frac{D}{M - E}$$

where D is the number of dead mutants, M is the total number of mutants, and E is the number of equivalent mutants.

Notice that the general problem of deciding whether a mutant is equivalent to the original program is undecidable.

3.2.2 *Theoretical Foundations of Mutation Adequacy.* Mutation analysis is based on two basic assumptions—the *competent programmer hypothesis* and the *coupling effect hypothesis*. These two assumptions are based on observations made in software development practice, and if valid enable the practical application of mutation testing to real software [DeMillo et al. 1988]. However, they are very strong assumptions whose validity is not self-evident.

A. *Competent programmer assumption.* The competent programmer hypothesis assumes that the program to be tested has been written by competent programmers. That is, "they create programs that are close to being correct" [DeMillo et al. 1988]. A consequence drawn from the assumption by DeMillo et al. [1988] is that "if we are right in our perception of programs as being close to correct, then these errors should be detectable as small deviations from the intended program." In other words, the mutants to be considered in mutation analysis are those within a small deviation from the original program. In practice, such mutants are obtained by systematically and mechanically applying a set of transformations, called mutation operators, to the program under test. These mutation operators would ideally model programming errors made by programmers. In practice, this may be only partly true. We return to mutation operators in Section 3.2.3.

B. *Coupling effect assumption.* The second assumption in mutation analysis is the coupling effect hypothesis, which assumes that simple and complex errors are coupled, and hence test data that cause simple nonequivalent mutants to die will usually also cause complex mutants to die.

Trying to validate the coupling effect assumption, Offutt [1989; 1992] did an empirical study with the Mothra mutation-testing tool. He demonstrated that a test set developed to kill first-order mutants (i.e., mutants) can be very successful at killing second-order mutants (i.e., mutants of mutants). However, the relationship between second-order mutants and complex faults remains unclear.

In the search for the foundation of mutation testing, theories have been developed for fault-based testing in general and mutation testing in particular. Program correctness is usually taken to mean that the program computes the intended function. But in mutation analysis there is another notion of correctness, namely, that a certain class of faults has been ruled out. This is called *local correctness* here to distinguish it from the usual notion of correctness.

According to Budd and Angluin [1982] local correctness of a program p can be defined relative to a *neighborhood* Φ of p, which is a set of programs containing the program p itself. Precisely speaking, Φ is a mapping from a program p to a set of programs $\Phi(p)$. We say that p is *locally correct* with respect to Φ if, for all programs q in $\Phi(p)$, either q is equivalent to p or q fails on at least one test point in the input space. In other words, in the neighborhood of a program only the program and its equivalents are possibly correct, because all others are incorrect. With the competent programmer hypothesis, local correctness implies correctness if the neighborhood is always large enough to cover at least one correct program.

Definition 3.2 (*Neighborhood Adequacy Criterion*). A test set t is Φ-adequate if for all programs $q \neq p$ in Φ, there exists x in t such that $p(x) \neq q(x)$.

Budd and Angluin [1982] studied the computability of the generation and recognition of adequate test sets with respect to the two notions of correctness, but left the neighborhood construction problem open.

The neighborhood construction problem was investigated by Davis and Weyuker [1988; 1983]. They developed a theory of metric space on programs written in a language defined by a set of production rules. A transition sequence from program p to program q was defined to be a sequence of the programs $p_1, p_2, \ldots, p_n$ such that $p_1 = p$ and $p_n = q$, and for each $i = 1, 2, \ldots, n - 1$, either p_{i+1} is obtained from p_i (a forward step) or p_i can be obtained from p_{i+1} (a backward step), using one of the productions of the grammar in addition to a set of short-cut rules to catch the intuition that certain kinds of programs are conceptually close. The length of a transition sequence is defined to be the maximum of the number of forward steps and the number of backward steps in the transition sequence. The distance $\rho(p, q)$ between program p and q is then defined as the smallest length of a transition sequence from p to q. This distance function on the program space can be proved to satisfy the axioms of a metric space, that is, for all programs p, q, r,

Table I. Levels of Analysis in Mutation Testing

Level	Goal	Method
Statement Analysis	Ensure that every branch is taken and every statement is necessary.	Replace statement with CONTINUE; Replace statement with TRAP; Replace logicals and relationals with *true* or *false*; Check labels on arithmetic IF statements for usage; Replace DO statements with FOR statements.
Predicate Analysis	Exercise predicate boundaries	Alter predicate and DO loop limit sub-expressions by small amounts; Insert absolute value operators into predicate sub-expressions; Alter relational operators.
Domain Analysis	Exercise different data domains	Change constants and sub-expressions by small amounts; Insert absolute value operators wherever syntactically correct;
Coincidental Correctness[a] Analysis	Guard against coincidental correctness	Change data references and operators to other syntactically correct alternatives.

[a] An erroneous expression may coincidentally compute correct output on a particular input. For example, assume that a reference to a variable is mistaken by referring to another variable, but in a particular case these two variables have the same value. Then, the expression computes the correct output on the particular input. If such an input is used as a test case, it is unable to detect the fault. Coincidental correctness analysis attempts to analyse whether a test set suffers from such weakness.

(1) $\rho(p, q) \geq 0$;
(2) $\rho(p, q) = 0$ if and only if $p = q$;
(3) $\rho(p, q) = \rho(q, p)$;
(4) $\rho(p, r) \geq \rho(p, q) + \rho(q, r)$.

Then, the neighborhood $\Phi_\delta(p)$ of program p within a distance δ is the set of programs within the distance δ according to ρ; formally, $\Phi_\delta(p) = \{q \mid \rho(p, q) \leq \delta\}$. Davis and Weyuker introduced the notion of critical points for a program with respect to a neighborhood. Intuitively, a critical point for a program is a test case that is the unique input to kill some mutant in the neighborhood set. Formally:

Definition 3.3 (*Critical Points*). An input c is a Φ-critical point for p with respect to Φ if there exists a program $q \in \Phi$ such that for all $x \neq c$, $p(x) = q(x)$ but $p(c) \neq q(c)$.

There is a nice relationship between critical points [Davis and Weyuker 1988] and adequate test sets. First, Φ-critical points must be members of any Φ-adequate test sets. Second, when the distance δ increases, the neighborhood set $\Phi_\delta(p)$ as well as the set of critical points increase in the sense of set inclusion. That is, if c is $\Phi_\delta(p)$-critical, then for all $\varepsilon \geq \delta$, c is also $\Phi_\varepsilon(p)$-critical. Finally, by studying minimally adequate test sets, Davis and Weyuker [1988] obtained a lower bound of the numbers of non-critical points that must be present in an adequate test set.

3.2.3 *Mutation Transformations.* Now let us consider the problem of how to generate mutants from a given program. A practical method is the use of mutation operators. Generally speaking, a mutation operator is a syntactic transformation that produces a mutant when applied to the program under test. It applies to a certain type of syntactic structure in the program and replaces it with another. In current mutation testing tools such as Mothra [King and Offutt 1991], the mutation operators are designed on the basis of many years' study of programmer errors. Different levels of mutation analysis can be done by applying certain types of mutation operators to the corresponding syntactical structures in the program. Table I from Budd [1981], briefly describes the levels of mutation analysis and the corresponding mutation operators. This framework has been used by almost all of the mutation testing tools.

3.2.4 *The Pros and Cons.* Using a mutation-based testing system, a tester supplies the program to be tested and chooses the levels of mutation analysis to be performed. Then the system generates the set of mutants. The tester also supplies a test set. The testing system executes the original program and each mutant on each test case and compares the outputs produced by the original program and its mutants. If the output of a mutant differs from the output of the original program, the mutant is marked dead. Once execution is complete, the tester examines any mutants that are still alive. The tester can then declare a mutant to be equivalent to the original program, or can supply additional test data in an effort to kill the mutant.

Reports on experiments with mutation-based testing have claimed that the method is powerful and has a number of attractive features [DeMillo and Mathur 1990; DeMillo et al. 1988].

(1) Mutation analysis allows a great degree of automation. Mutants are generated by applying mutation operators and are then compiled and executed. The outputs of the mutants and the original programs are compared and then a mutation adequacy score is calculated. All these can be supported by mutation-testing software such as that of Mothra [DeMillo et al. 1988; King and Offutt 1991].
(2) Mutation-based testing systems provide an interactive test environment that allows the tester to locate and remove errors. When a program under test fails on a test case and a mutant does not, the tester should find it easy to locate and remove the error by considering the mutation operator applied and the location where the mutation operator is applied.
(3) Mutation analysis includes many other testing methods as special cases. For example, statement coverage and branch coverage are special cases of statement analysis in mutation testing. This can be achieved by replacing a statement or a branch by the special statement TRAP, which causes the abortion of the execution.

The drawback of mutation testing is the large computation resources (both time and space) required to test large-scale software. It was estimated that the number of mutants for an n-line program is on the order of n^2 [Howden 1982]. Recently, experimental data confirmed this estimate; see Section 5.3 [Offutt et al. 1993]. A major expense in mutation testing is perhaps the substantial human cost of examining large numbers of mutants for possible equivalence, which cannot be determined effectively. An average of 8.8% of equivalent mutants had been observed in experiments [Offutt et al. 1993].

3.3 Variants of Program Mutation Testing

Using the same idea of mutation analysis, Howden [1982] proposed a testing method to improve efficiency. His testing method is called *weak mutation testing* because it is weaker than the original mutation testing. The original method was referred to as strong mutation testing. The fundamental concept of weak mutation testing is to identify classes of program components and errors that can occur in the components. Mutation transformations are then applied to the components to generate mutants of the components. For each component and a mutant of the component, weak mutation testing requires that a test datum be constructed so that it causes the execution of the component and that the original component and the mutant compute different values. The major advantage of weak mutation testing is efficiency. Although there is the same number of mutants in weak mutation testing as in strong mutation testing, it is not necessary to carry out a separate program execution for each mutant. Although a test set that is adequate for weak mutation testing may not be adequate for the strong mutation testing, experiments with weak mutation testing such as Offutt and Lee's [1991] and Marick's [1991] suggest that it can still be an effective testing method.

From a more general view of mutation-testing principles, Woodward and Halewood [1988] proposed *firm mutation testing*. They regarded mutation testing as the process of making small changes in a program and comparing the outcomes of the original and changed versions. They identified a set of parameters of such changes and comparisons. The basic idea of firm mutation testing is to make such parameters explicit so that they can be altered by testers. Strong and weak mutation testing are two extremes of the firm-mutation testing method. The advantage of firm-mutation testing is that it is less expensive than strong-mutation testing but stronger than weak-mutation testing. In addition, firm-mutation testing provides a mechanism for the control of fault introduction and test results comparison. The major disadvantage of firm-mutation testing is that there is no obvious systematic basis on which to select the parameters and the area of program code.

Returning to frameworks for strong mutation testing, *ordered mutation testing* was proposed to improve efficiency without sacrificing effectiveness. The idea is to construct an order $\le$ between mutants such that mutant b is stronger than a (written $a \le b$) if for any test case t, t kills b implies that t necessarily kills a. Therefore, mutant b should be executed on test cases before the execution of a, and a is executed only when the test data failed to kill b. A similar ordering can also be defined on test data. Given a mutant q of program p, q should be executed on test data t before the execution on test data s if t is more likely to kill q than s is. Ordering on mutation operators was proposed to achieve the ordering on mutants in early '80s by Woodward et al. [1980] and Riddell et al. [1982], and reappeared recently in a note by Duncan and Robson [1990]. A mutation operation ϕ is said to be stronger than ϕ' if for all programs p, the application of ϕ to p always gives a mutant that is stronger than the mutants obtained by the application of ϕ' to p at the same location. Taking the relational operator $=$ as an example, mutants can be generated by replacing "$=$" with "$\neq$", "$\le$", "$\ge$", "$<$", and "$>$", respectively. Intuitively, replacing "$=$" with "$\neq$" should be stronger than replacing with "$<$", because if the test data are not good enough to distinguish "$=$" from "$\neq$", it would not be adequate for other relational operators. Based on such arguments, a partial ordering on relational operators was defined. However, Woodward [1991] proved that operator ordering is not the right approach to achieving mutant ordering. The situation turns out to be quite complicated when a mutation operator is applied to a loop body, and a counterexample was given. Experiments are needed to see how effective ordered mutation testing can be and to assess the extent of cost saving.

It was observed that some mutation operators generate a large number of mutants that are bound to be killed if a test set kills other mutants. Therefore, Mathur [1991] suggested applying the mutation testing method without applying the two mutation operators that produce the most mutants. Mathur's method was originally called *constrained mutation testing*. Offutt et al. [1993] extended the method to omit the first N most prevalent mutation operators and called it *N-selective mutation testing*. They did an experiment on selective testing with the Mothra mutation testing environment. They tested 10 small programs using an automatic test-case generator, Godzilla [DeMillo and Offutt 1991; 1993]. Their experimentation showed that with full selective mutation score, an average nonselective mutation score of 99.99, 99.84 and 99.71% was achieved by 2-selective, 4-selective, and 6-selective mutation testing, respectively. Meanwhile, the average savings[5] for these selective mutations were 23.98, 41.36, and 60.56%,

[5] The savings are measured as the reduction of the number of mutants.

respectively [Offutt et al. 1993]. Mathur theorized that selective mutation testing has complexity linear in program size measured as the number of variable references, and yet retains much of the effectiveness of mutation testing. However, experimental data show that the number of mutants is still quadratic although the savings are substantial [Offutt et al. 1993].

3.4 Perturbation Testing

While mutation testing systematically plants faults in programs by applications of syntactic transformations, Zeil's perturbation testing analyses test effectiveness by considering faults in an "error" space. It is concerned with faults in arithmetic expressions within program statements [Zeil 1983]. It was proposed to test *vector-bounded* programs, which have the following properties: (1) vector-bounded programs have a fixed number of variables on a continuous input domain; (2) in the flow-graph model, each node n in the flow graph is associated with a function C_n that transforms the environment v to a new environment v'. Here, environment is a vector $(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m)$ which consists of the current values of the input variables x_i and the values of the intermediate and output variables y_i. The functions associated with nodes are assumed to be in a class **C** which is closed under function composition. (3) each node may have at most two outward arcs. In such cases the node n is also associated with a predicate T_n, which is applied to the new environment obtained by applying C_n and compared with zero to determine the next node for execution. It was assumed that the predicates are simple, not combined with *and*, *or*, or other logical operators. (4) the predicates are assumed to be in a class **T** which is a vector space over the real numbers R of dimension k and is closed under composition with **C**. Examples of vector spaces of functions include the set of linear functions, the set of polynomial functions of degree k, and the set of multinomials of degree k.

There are some major advantages of vector boundedness. First, any finite-dimensioned vector space can be described by a finite set of characteristic vectors such that any member of the vector space is a linear combination of the characteristic vectors. Second, vector spaces are closed under the operations of addition and scalar multiplication. Suppose that some correct function C has been replaced by an erroneous form C'. Then the expression $C - C'$ is the effect of the fault on the transformation function. It is called the *error function* of C' or the *perturbation* to C. Since the vector space contains C and C', the error function must also be in the vector space. This enables us to study faults in a program as functional differences between the correct function and the incorrect function rather than simply as syntactical differences, as in mutation testing. Thus it builds a bridge between fault-based testing and error-based testing, which is discussed in Section 4.

When a particular error function space **E** is identified, a neighborhood of a given function f can be expressed as the set of functions in the form of $f + f_e$, where $f_e \in \mathbf{E}$. Test adequacy can then be defined in terms of a test set's ability to detect error functions in a particular error function space. Assuming that there are no missing path errors in the program, Zeil considered the error functions in the function C and the predicate T associated with a node. The following gives more details about perturbation of these two types of functions.

3.4.1 *Perturbation to the Predicate.* Let n be a node in a flow graph and C_n and T_n be the computation function and predicate function associated with node n. Let A be a path from the begin node to node n and C_A be the function computed through the path. Let T' be the correct predicate that should be used in place of T_n. Suppose that the predicate is tested by an execution through the path A. Recall that a predicate is a real-valued function, and its result is compared with zero to determine the direction of control transfer. If in a test execution the error function $e = T' - T_n$ is evaluated to zero, it does not affect the direction of control transfer. Therefore the error function $e = T' - T_n$ cannot be detected by such an execution if and only if there exists a positive scalar α such that

$$T_n \circ C_A(v_0) = \alpha T' \circ C_A(v_0),$$

for all initial environments v_0 which cause execution through path A, where $\circ$ is functional composition. The subset of $\mathbf{T}$ that consists of the functions satisfying the preceding equation is called the *blindness space* for the path A, denoted by BLIND(A). Zeil identified three types of blindness and provided a constructive method to obtain BLIND(A) for any path A. *Assignment blindness* consists of functions in the following set.

$$null(C_A) = \{e | e \in \mathbf{T} \wedge \forall v.\, e \circ C_A(v) = 0\}.$$

These functions evaluate to zero when the expressions computed in C_A are substituted for the program variables. After the assignment statement "$X := f(v)$", for example, the expression "$X - f(v)$" can be added into the set of undetectable predicates. *Equality blindness* consists of equality restrictions on the path domain. To be an initial environment that causes execution of a path A, it must satisfy some restrictions. A restriction can be an equality, such as $x = 2$. If an input restriction $r(v_0) = 0$ is imposed, then the predicate $r(v) = 0$ is an undetectable error function. The final component of undetectable predicates is the predicate T_n itself. Because $T_n(v)$ compared with zero and $\alpha T_n(v)$ compared with zero are identical for all positive real numbers α, *self-blindness* consists of all predicates of the form $\alpha T_n(v)$. These three types of undetectable predicates can be combined to form more complicated undetectable error functions.

Having given a characterization theorem of the set BLIND(A) and a constructive methods to calculate the set, Zeil defined a test-path selection criterion for predicate perturbations.

Definition 3.4 (*Adequacy of Detecting Predicate Perturbation*). A set P of paths all ending at some predicate T is *perturbation-test-adequate* for predicate T if

$$\bigcap_{p \in P} BLIND(p) = \emptyset.$$

Zeil's criterion was originally stated in the form of a rejection rule: if a program has been reliably tested on a set P of execution paths that all end at some predicate T, then an additional path p also ending at T need not be tested if and only if

$$\bigcap_{x \in P} BLIND(x) \subseteq BLIND(p).$$

Zeil also gave the following theorem about perturbation testing for predicate errors.

THEOREM 3.1 (Minimal Adequate Test Set for Predicate Perturbation Testing). *A minimal set of subpaths adequate for testing a given predicate in a vector bounded program contains at most k subpaths, where k is the dimension of* $\mathbf{T}$.

3.4.2 *Perturbation to Computations.* The perturbation function of the computation associated with node n can also be expressed as $e = C' - C_n$ where C' is the unknown correct computation. However, a fault in computation function may cause two types of errors: domain error or computation error. A computation error can be revealed if there is a path A from the begin node to the node n and a path B from node n to a node that contains an output statement M, such that for some initial environment v_0 that causes the execution of paths A and B

$$M \circ C_B \circ C_n \circ C_A(v_0) \neq M \circ C_B \circ C' \circ C_A(v_0).$$

A sufficient condition of this inequality is that $M \circ C_B \circ e \circ C_A(v_0) \neq 0$. The set of functions that satisfy the equation $M \circ C_B \circ e \circ C_A(v_0) = 0$ for all v_0 that execute A and B is then the *blindness space* of the test execution through the path A followed by B.

A domain error due to an erroneous computation function C_n can be revealed by a path A from the begin node to node n and a path B from node n to a node m with predicate T_m, if for some initial environment v_0 that causes the execution of paths A and B to node m.

$$T_m \circ C_B \circ C_n \circ C_A(v_0) \neq \alpha T_m \circ C_B \circ C' \circ C_A(v_0).$$

In other words, the erroneous computation may cause domain error if the fault affects the evaluation of a predicate. The blindness space of a test path for detecting such perturbations of a computation function can be expressed by the equation: $T_m \circ C_B \circ e \circ C_A(v_0) = 0$, where $e = C_n - C'$.

A solution to the two preceding equations was obtained for *linear-dominated programs*, which are bounded vector programs with the additional restrictions that **C** is the set of linear transformations and **T** is the set of linear functions [Zeil 1983]. The solution essentially consists of two parts: assignment blindness and equality blindness, as in predicate perturbation, and blindness due to computation C_B masking out differences between C_n and C'. An adequacy criterion similar to the predicate perturbation adequacy was also defined.

The principle of perturbation testing can be applied without the assumptions about the computation function space **C** and the predicate function space **T**, but selecting an appropriate error function space **E**. Zeil [1983] gave a characterization of the blindness space for vector error spaces without the condition of linearity. The principle of perturbation testing can also be applied to individual test cases [Zeil 1984]. The notion of error space has been applied to improve domain-testing methods as well [Afifi et al. 1992; Zeil et al. 1992], which are discussed in Section 4.

Although both mutation testing and perturbation testing are aimed at detecting faults in software, there are a number of differences between the two methods. First, mutation testing mainly deals with faults at the syntactical level, whereas perturbation testing focuses on the functional differences. Second, mutation testing systematically generates mutants according to a given list of mutation operators. The perturbation testing perturbs a program by considering perturbation functions drawn from a structured error space, such as a vector space. Third, in mutation testing, a mutant is killed if on a test case it produces an output different from the output of the original program. Hence the fault is detected. However, for perturbation testing, the blindness space is calculated according to the test cases. A fault is undetected if it remains in the blindness space. Finally, although mutation testing is more generally applicable in the sense that there are no particular restrictions on the software under test, perturbation testing guarantees that combinations of the faults can be detected if all simple faults are detected, provided that the error space is a vector space.

3.5 The RELAY Model

As discussed in perturbation testing, it is possible for a fault not to cause an error in output even if the statement containing the fault is executed. With the restriction to linear-domained programs, Zeil provided conditions on which a perturbation (i.e., a fault), can be exposed.

The problem of whether a fault can be detected was addressed by Morrel [1990], where a model of error origination and transfer was proposed. An error is *originated* (called “created” by Morrel) when an incorrect state is introduced at some fault location, and it is

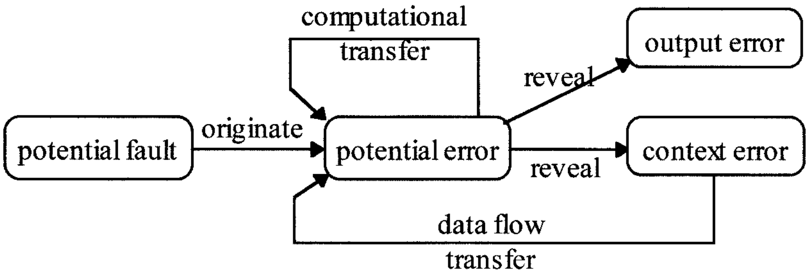

Figure 4. RELAY model of fault detection.

transferred (called "propagated" by Morrel) if it persists to the output.

The RELAY model proposed by Richardson and Thompson [1988; 1993] was built upon Morrel's theory, but with more detailed analysis of how an error is transferred and the conditions of such transfers. The errors considered within the RELAY model are those caused by particular faults in a module. A *potential fault* is a discrepancy between a node n in the module under test and the corresponding node n^* in the hypothetically correct module. This potential fault results in a *potential error* if the expression containing the fault is executed and evaluated to a value different from that of the corresponding hypothetically correct expression. Given a potential fault, a potential error *originates* at the smallest subexpression of the node containing the fault that evaluates incorrectly. The potential error transfers to a *super*expression if the superexpression evaluates incorrectly. Such error transfers are called *computational transfer*. To reveal an output error, execution of a potential fault must cause an error that transfers from node to node until an incorrect output results, where an error in the function computed by a node is called a *context error*. If a potential error is reflected in the value of some variable that is referenced at another node, the error transfer is called a *data-flow transfer*. This process of error transfer is illustrated in Figure 4.

Therefore the conditions under which a fault is detected are:

(1) origination of a potential error in the smallest subexpression containing the fault;
(2) computational transfer of the potential error through each operator in the node, thereby revealing a context error;
(3) data-flow transfer of that context error to another node on the path that references the incorrect context;
(4) cycle through (2) and (3) until a potential error is output.

If there is no input datum for which a potential error originates and all these transfers occur, then the potential fault is not a fault. This view of error detection has an analogy in a relay race, hence the name of the model. Based on this view, the RELAY model develops *revealing conditions* that are necessary and sufficient to guarantee error detection. Test data are then selected to satisfy revealing conditions. When these conditions are instantiated for a particular type of fault, they provide a criterion by which test data can be selected for a program so as to guarantee the detection of an error caused by any fault of that type.

3.6 Specification-Mutation Testing

Specification-fault-based testing attempts to detect faults in the implementation that are derived from misinterpreting the specification or the faults in the specification. Specification-fault-based testing involves planting faults into the specification. The program that implements the original specification is then executed and the results of the execution are checked against the original specification and those with planted faults. The adequacy of the test is determined according to whether all the planted faults are detected by checking.

Gopal and Budd [1983] extended program-mutation testing to specification-mutation testing for specifications written in predicate calculus. They identified a set of mutation operations that are applied to a specification in the form of pre/postconditions to generate mutants of the specification. Then the program under test is executed on test cases to obtain the output of the program. The input/output pair is used to evaluate the mutants of the specification. If a mutant is falsified by the test cases in the evaluation, we say that it is killed by the test cases; otherwise it remains alive.

Gopal and Budd [1983] noticed that some alterations to specifications were not useful as mutation operators. For example, replacing the various clauses in the specification by the truth values "true" or "false" tends to generate mutants that are trivial to kill and appear to be of little practical significance.

In an investigation of mutation operators for algebraic specifications, Woodward [1993] defined his set of mutation operators based on his analysis of errors in the specifications made by students. He considered algebraic specifications as term-rewriting systems. The original specification and the mutants of the specification are compiled into executable codes. When the executions of the original specification and a mutant on a given test case generate two different outputs, the mutant is regarded as dead. Otherwise, it is alive. In this way the test adequacy is measured without executing the program.

3.7 Summary of Fault-Based Adequacy Criteria

Fault-based adequacy criteria focus on the faults that could possibly be contained in the software. The adequacy of a test set is measured according to its ability to detect such faults.

Error seeding is based on the assumption that the artificial faults planted in the program are as difficult to detect as the natural errors. This assumption has proved not true in general.

Mutation analysis systematically and automatically generates a large number of mutants. A mutant represents a possible fault. It can be detected by the test cases if the mutant produces an output different from the output of the original program. The most important issue in mutation-adequacy analysis is the design of mutation operators. The method is based on the competent-programmer and coupling-effect hypotheses. The principle of mutation-adequacy analysis can be extended to specification-based-adequacy analysis. Given a set of test cases and the program under test, mutation adequacy can be calculated automatically except for detection of equivalent mutants.

However, measuring the adequacy of software testing by mutation analysis is expensive. It may require large computation resources to store and execute a large number of mutants. It also requires huge human resources to determine if live mutants are equivalent to the original program. Reduction of the testing expense to a practically acceptable level has been an active research topic. Variants such as weak mutation testing, firm mutation testing, and ordered mutation testing have been proposed. Another approach not addressed in this article is the execution of mutants in parallel computers [Choi et al. 1989; Krauser et al. 1991]. This requires the availability of massive parallel computers and very high portability of the software so that it can be executed on the target machine as well as the testing machine. In summary, although progress has been made to reduce the expense of adequacy measurement by mutation analysis, there remain open problems.

Perturbation testing is concerned with the possible functional differences between the program under test and the hypothetical correct program. The adequacy of a test set is decided by its ability to limit the error space defined in terms of a set of functions.

A test case may not reveal a fault. The RELAY model analyzed the condition under which a test case reveals a fault. Therefore, given a fault, test cases can be selected to satisfy the conditions, and hence guarantee the detection of the fault. This model can also be used as a basis of test adequacy criteria.

4. ERROR-BASED ADEQUACY CRITERIA AND DOMAIN ANALYSIS

Error-based testing methods require test cases to check programs on certain error-prone points [Foster 1980; Myers 1979]. The basic idea of domain analysis and domain testing is to partition the input-output behavior space into subdomains so that the behavior of the software on each subdomain is equivalent, in the sense that if the software behaves correctly for one test case within a subdomain, then we can reasonably assume that the behavior of the software is correct for all data within that subdomain. We may wish, however, to take more than one test case within each subdomain in order to increase our confidence in the conformance of the implementation upon this subdomain. Given a partition into subdomains, the question is how many test cases should be used for each subdomain and where in the subdomain these test cases should be chosen. The answers to these questions are based on assumptions about the whereabouts of errors. However, theoretical studies have proved that testing on certain sets of error-prone points can detect certain sets of faults in the program [Afifi et al. 1992; Clarke et al. 1989; Richardson and Clarke 1985; Zeil 1984; 1983; 1989a; 1989b; 1992].

Before considering these problems, let us first look at how to partition the behavior space.

4.1 Specification-Based Input Space Partitioning

The software input space can be partitioned either according to the program or the specification. When partitioning the input space according to the specification, we consider a subset of data as a subdomain if the specification requires the same function on the data.

For example, consider the following specification of a module DISCOUNT INVOICE, from Hall and Hierons [1991].

Example 4.1 (*Informal Specification of DISCOUNT INVOICE Module*). A company produces two goods, X and Y, with prices £5 for each *X* purchased and £10 for each *Y* purchased. An order consists of a request for a certain number of *X*s and a certain number of *Y*s. The cost of the purchase is the sum of the costs of the individual demands for the two products discounted as explained in the following. If the total is greater than £200, a discount of 5% is given; if the total is greater than £1,000, a discount of 20% is given; also, the company wishes to encourage sales of *X* and give a further discount of 10% if more than thirty *X*s are ordered. Noninteger final costs are rounded down to give an integer value.

When the input x and y to the module DISCOUNT INVOICE has the property that $x \le 30$ and $5^*x + 10^* y \le 200$, the output should be $5^*x + 10^*y$. That is, for all the data in the subset $\{(x, y) \mid x \le 30, 5^*x + 10^*y \le 200\}$, the required computation is the same function $sum = 5^*x + 10^*y$. Therefore the subset should be one subdomain. A careful analysis of the required computation will give the following partition of the input space into six subdomains, as shown in Figure 5.

It seems that there is no general mechanically applicable method to derive partitions from specifications, even if there is a formal functional specification. However, systematic approaches to analyzing formal functional specifications and deriving partitions have been

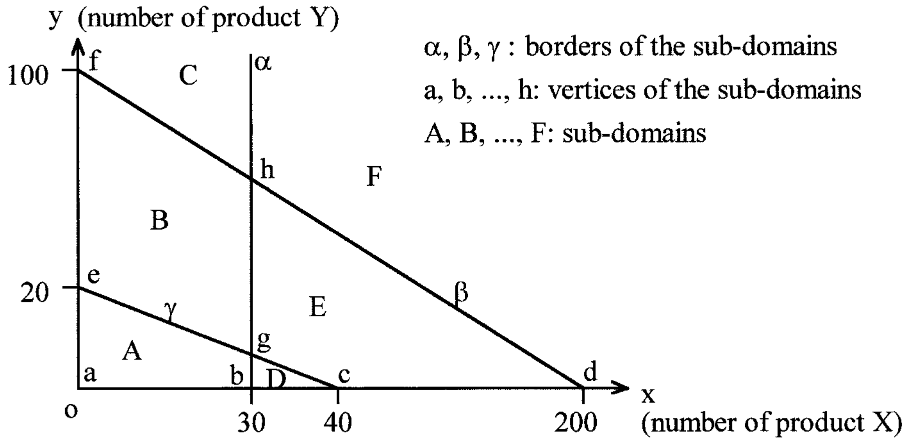
y (number of product Y)
α, β, γ : borders of the sub-domains
a, b, ..., h: vertices of the sub-domains
A, B, ..., F: sub-domains
100
20
o
30
40
200
x
(number of product X)
f
C
α
h
F
B
e
γ
E
β
g
A
a
b
D
c
d

Figure 5. Partition of input space of DISCOUNT INVOICE module.

proposed by a number of researchers, such as Stocks and Carrington [1993]. In particular, for formal specifications in certain normal forms, the derivation of partitions is possible. Hierons [1992] developed a set of transformation rules to transform formal functional specifications written in pre/postconditions into the following normal form.

$$P_1(x_1, x_2, \ldots, x_n)$$
$$\wedge\ Q_1(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m) \vee$$
$$P_2(x_1, x_2, \ldots, x_n)$$
$$\wedge\ Q_2(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m) \vee$$
$$\ldots$$
$$P_K(x_1, x_2, \ldots, x_n)$$
$$\wedge\ Q_K(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m) \vee$$

where $P_i(x_1, x_2, \ldots, x_n)$, $i = 1, 2, \ldots, K$, are preconditions that give the condition on the valid input data and the state before the operation. $Q_i(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m)$, $i = 1, 2, \ldots, K$, are postconditions that specify the relationship between the input data, output data, and the states before and after the operation. The variables x_i are input variables and y_i are output variables.

The input data that satisfy a precondition predicate P_i should constitute a subdomain. The corresponding computation on the subdomain must satisfy the postcondition Q_i. The precondition predicate is called the subdomain's domain condition. When the domain condition can be written in the form of the conjunction of atomic predicates of inequality, such as

$$\text{expl}(x_1, x_2, \ldots, x_n) \le \text{expr}(x_1, x_2, \ldots, x_n) \quad (4.1)$$

$$\text{expl}(x_1, x_2, \ldots, x_n) < \text{expr}(x_1, x_2, \ldots, x_n), \quad (4.2)$$

then the equation

$$\text{expl}(x_1, x_2, \ldots, x_n) = \text{expr}(x_1, x_2, \ldots, x_n) \quad (4.3)$$

defines a border of the subdomain.

Example 4.2 (*Formal specification of the DISCOUNT INVOICE module*). For the DISCOUNT INVOICE module, a formal specification can be written as follows.

$$(x \ge 0 \wedge y \ge 0) \Rightarrow (\text{sum} = 5x + 10y)$$

$$(\text{sum} \le 200) \Rightarrow (\text{discount1} = 100)$$

$$(\text{sum} > 200) \wedge (\text{sum} \leq 1000) \Rightarrow (\text{discount1} = 95)$$

$$(\text{sum} > 1000) \Rightarrow (\text{discount1} = 80))$$

$$(x \leq 30) \Rightarrow (\text{discount2} = 100)$$

$$(x > 30) \Rightarrow (\text{discount2} = 90)$$

$$(\text{total} = \text{round}(\text{sum} \cdot \text{discount1} \cdot \text{discount2}/10000)),$$

where round is the function that rounds down a real number to an integer value; x is the number of product X that a customer ordered; y is the number of product Y that the customer ordered; sum is the cost of the order before discount; discount1, discount2 are the percentages of the first and second type of discount, respectively; and total is the grand total cost of the purchase.

When this specification is transformed into the normal form, we have a clause with the following predicate as precondition.

$$(x \geq 0) \,\&\, (y \geq 0) \,\&\, (\text{sum} \leq 200) \,\&\, \text{not}(\text{sum} > 200 \,\&\, \text{sum} \leq 1000) \,\&\, \text{not}(\text{sum} > 1000) \,\&\, (x \leq 30) \,\&\, \text{not}\ (x > 30)$$

and the following predicate as the corresponding postcondition,

$$(\text{discount1} = 100) \,\&\, (\text{discount2} = 100) \,\&\, (\text{total} = \text{round}(\text{sum}*\text{discount1}*\text{discount2}/10000)).$$

The precondition defines the region A in the partition shown in Figure 5. It is equivalent to

$$(x \geq 0) \,\&\, (y \geq 0) \,\&\, (\text{sum} \leq 200) \,\&\, (x \leq 30). \tag{4.4}$$

Since sum = $5*x + 10*y$, the borders of the region are the lines defined by the following four equations:

$$x = 0, \quad y = 0, \quad x = 30, \quad 5*x + 10*y = 200.$$

They are the x axis, the y axis, the borders α and γ in Figure 5, respectively.

Notice that the DISCOUNT INVOICE module has two input variables. Hence the border equations should be understood as lines in a two-dimensional plane as in Figure 5. Generally speaking, the number of input variables is the dimension of the input space. A border of a subdomain in a K-dimensional space is a surface of $K - 1$ dimension.

4.2 Program-Based Input-Space Partitioning

The software input space can also be partitioned according to the program. In this case, two input data belong to the same subdomain if they cause the same "computation" of the program. Usually, the same execution path of a program is considered as the same computation. Therefore, the subdomains correspond to the paths in the program. When the program contains loops, a particular subset of the paths is selected according to some criterion, such as those discussed in Section 2.1.

Example 4.3 Consider the following program, which implements the DISCOUNT INVOICE.

```
Program DISCOUNT_INVOICE
(x, y: Int)
 Var  discount1,  discount2:
 Int;
 input (x, y);
 if x ≤ 30
 then discount2 := 100
 else discount2 := 90
 endif;
 sum := 5*x + 10*y;
 if sum ≤ 200
 then discount1 := 100
 elseif sum ≤ 1000
```

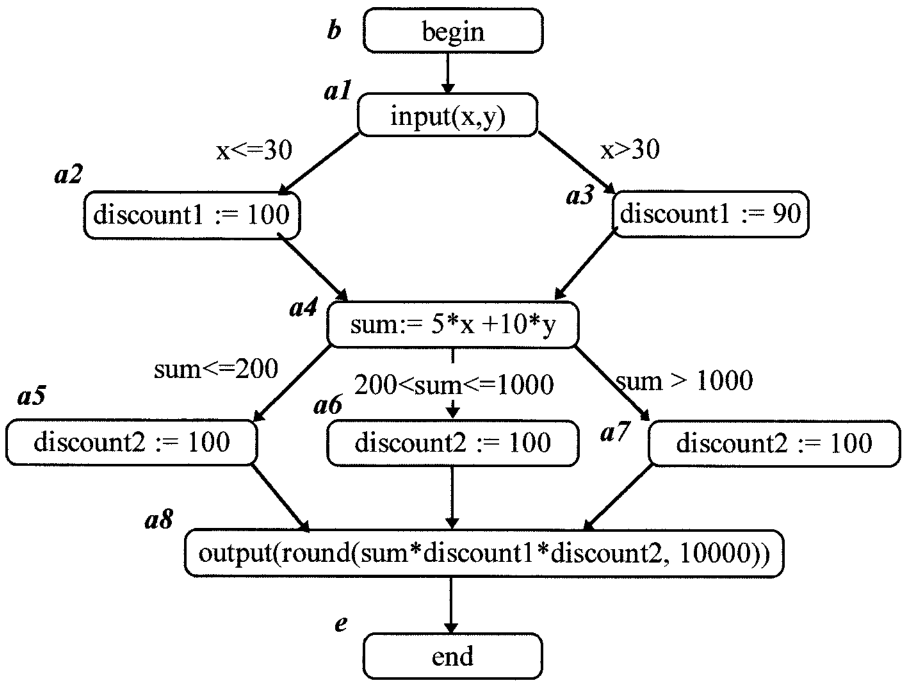

Figure 6. Flow graph of DISCOUNT INVOICE program.

```
     then discount1 := 95
     else discount1 := 80
     endif;
     output (round(sum*discount1
     *discount2/10000))
    end
```

There are six paths in the program; see Figure 6 for its flow graph. For each path there is a condition on the input data such that an input causes the execution of the path if and only if the condition is satisfied. Such a condition is called the *path condition*, and can be derived mechanically by, say, symbolic execution [Girgis 1992; Howden 1977; 1978; Liu et al. 1989; Young and Taylor 1988]. Therefore, path conditions characterize the subdomains (i.e., they are domain conditions). Table II gives the path conditions of the paths.

Notice that, for the DISCOUNT INVOICE example, partitioning according to program paths is different from the partitioning according to specification. The borders of the x axis and the y axis are missing in the partitioning according to the program.

Now let us return to the questions about where and how many test cases should be selected for each subdomain. If only one test case is required and if we do not care about the position of the test case in the subdomain, then the test adequacy is equivalent to path coverage provided that the input space is partitioned according to program paths. But error-based testing requires test cases selected not only within the subdomains, but also on the boundaries, at vertices, and just off the vertices or the boundaries in each of the adjacent subdomains, because these places are traditionally thought to be error-prone. A test case in the subdomain is called an *on* test point; a test case that lies outside the subdomain is called an *off* test point.

This answer stems from the classification of program errors into two main types: domain error and computation

Table II. Paths and Path Conditions

Path	Path Condition
(b, a1, a2, a4, a5, a8, e)	$x \leq 30$, $(5*x + 10*y) \leq 200$
(b, a1, a2, a4, a6, a8, e)	$x \leq 30$, $200 < (5*x + 10*y) \leq 1000$
(b, a1, a2, a4, a7, a8, e)	$x \leq 30$, $1000 < (5*x + 10*y)$
(b, a1, a3, a4, a5, a8, e)	$30 < x$, $(5*x + 10*y) \leq 200$
(b, a1, a3, a4, a6, a8, e)	$30 < x$, $200 < (5*x + 10*y) \leq 1000$
(b, a1, a3, a4, a7, a8, e)	$30 < x$, $1000 < (5*x + 10*y)$

error. Domain errors are those due to the incorrectness in a program's selection of boundaries for a subdomain. Computation errors are those due to the incorrectness of the implementation of the computation on a given subdomain. This classification results in two types of domain testing. The first aims at the correctness of the boundaries for each subdomain. The second is concerned with the computation on each subdomain. The following give two sets of criteria for these two types of testing.

4.3 Boundary Analysis

White and Cohen [1980] proposed a test method called $N \times 1$ domain-testing strategy that requires N test cases to be selected on the borders in an N-dimensional space and one test case just off the border. This can be defined as the following adequacy criterion.

Definition 4.1 (*$N \times 1$ Domain Adequacy*). Let $\{D_1, D_2, \ldots, D_n\}$ be the set of subdomains of software S that has N input variables. A set T of test cases is said to be $N \times 1$ *domain-test adequate* if, for each subdomain D_i, $i = 1, 2, \ldots, n$, and each border B of D_i, there are at least N test cases on the border B and at least one test case which is just off the border B. If the border is in the domain D_i, the test case off the border should be an *off* test point, otherwise, the test case off the border should be an *on* test point.

An adequacy criterion stricter than $N \times 1$ adequacy is the $N \times N$ criterion, which requires N test cases off the border rather than only one test case off the border. Moreover, these N test cases are required to be linearly independent [Clarke et al. 1982].

Definition 4.2 (*$N \times N$ Domain Adequacy*). Let $\{D_1, D_2, \ldots, D_n\}$ be the set of subdomains of software S that has N input variables. A set T of test cases is said to be $N \times N$ *domain-test adequate* if, for each subdomain D_i, $i = 1, 2, \ldots, n$, and each border B of D_i, there are at least N test cases on the border B and at least N linearly independent test cases just off the border B. If the border B is in the domain D_i, the N test cases off the border should be *off* test points, otherwise they should be *on* test points.

The focal point of boundary analysis is to test if the borders of a subdomain are correct. The $N \times 1$ domain-adequacy criterion aims to detect if there is an error of parallel shift of a linear border, whereas the $N \times N$ domain-adequacy is able to detect parallel shift and rotation of linear borders. This is why the criteria select the specific position for the *on* and *off* test cases. Considering that the vertices of a subdomain are the points at the intersection of several borders, Clark et al. [1982] suggested the use of vertices as test cases to improve the efficiency of boundary analysis. The criterion requires that the vertices are chosen as test cases and that for each vertex a point close to the vertex is also chosen as a test case.

Definition 4.3 (*$V \times V$ Domain Adequacy*). Let $\{D_1, D_2, \ldots, D_n\}$ be the set of subdomains of software S. A set T of test cases is said to be *$V \times V$ domain-test adequate* if, for each subdomain D_i, $i = 1, 2, \ldots, n$, T contains the vertices of D_i and for each vertex v of D_i there is a test case just off the vertex v. If a vertex v of D_i is in the subdomain D_i, then the test case just off v should be an *off* test point; otherwise it should be an *on* point.

The preceding criteria are effective for detection of errors in linear domains, where a domain is a linear domain if its borders are linear functions. For nonlinear domains, Afifi et al. [1992] proved that the following criteria were effective for detecting linear errors.

Definition 4.4 (*$N + 2$ Domain Adequacy*). Let $\{D_1, D_2, \ldots, D_n\}$ be the set of subdomains of software S that has N input variables. A set T of test cases is said to be *$N + 2$ domain-test adequate* if, for each subdomain D_i, $i = 1, 2, \ldots, n$, and each border B of D_i, there are at least $N + 2$ test cases $x_1, x_2, \ldots, x_{N+2}$ in T such that

—each set of $(N + 1)$ test cases are in general position, where a set $\{\vec{x}_i\}$ containing $N + 1$ vectors is *in general position* if the N vectors $\vec{x}_i - \vec{x}_1$, $i = 2, 3, \ldots, N + 1$, are linear-independent;

—there is at least one *on* test point and one *off* test point;

—for each pair of test cases, if the two points are of the same type (in the sense that both are *on* test points or both are *off* test points), they should lie on the opposite sides of the hyperplane formed by the other n test points; otherwise they should lie on the same side of the hyperplane formed by the other n test points.

By applying Zeil's [1984; 1983] work on error spaces (see also the discussion of perturbation testing in Section 3.4), Afifi et al. [1992] proved that any test set satisfying the $N + 2$ domain-adequacy criterion can detect all linear errors of domain borders. They also provided a method of test-case selection to satisfy the criterion consisting of the following four steps.

(1) Choose $(N + 1)$ *on* test points in general position; the selection of these points should attempt to spread them as far from one another as possible and put them on or very close to the border.
(2) Determine the open convex region of these points.
(3) If this region contains *off* points, then select one.
(4) If this region has no *off* points, then change each *on* point to be an *off* point by a slight perturbation; now there are $N + 1$ *off* points and there is near certainty of finding an *on* point in the new open convex region.

4.4 Functional Analysis

Although boundary analysis focuses on border location errors, functional analysis emphasizes the correctness of computation on each subdomain. To see how functional analysis works, let us take the DISCOUNT INVOICE module as an example again.

Example 4.4 Consider region A in Figure 5. The function to be calculated on this subdomain specified by the specification is

$$total = 5*x + 10*y. \qquad (4.5)$$

This can be written as $f(x, y) = 5x + 10y$, a linear function of two variables. Mathematically speaking, two points are sufficient to determine a linear function, but one point is not. Therefore at least two test cases must be chosen in the subdomain. If the program also computes a linear function on the subdomain and produces correct outputs on the two test cases, then we can say that the program computes the correct function on the subdomain.

Analyzing the argument given in the preceding example, we can see that functional analysis is based on the following assumptions. First, it assumes that for each subdomain there is a set of functions Φ_f associated with the domain. In the preceding example, Φ_f consists of linear functions of two variables. Second, the specified computation of the software on this domain is considered as a function f^*, which is an element of Φ_f. Third, it is assumed that the function f computed by the software on the subdomain is also an element of Φ_f. Finally, there is a method for selecting a finite number of test cases for f^* such that if f and f^* agree on the test cases, then they are equivalent.

In Section 3.4, we saw the use of an error space in defining the neighborhood set Φ_f and the use of perturbation testing in detecting computation errors [Zeil 1984; 1983].

In Howden's [1978; 1987] algebraic testing, the set of functions associated with the subdomain is taken as the polynomials of degree less than or equal to k, where k is chosen as the degree of the required function, if it is a polynomial. The following mathematical theorem, then, provides a guideline for the selection of test cases in each subdomain [Howden, 1987].

THEOREM 4.1 *Suppose that* Φ *contains all multinomials in n variables* $x_1, x_2, \ldots, x_n$ *of degree less than or equal to k, and* $f, f^* \in \Phi$. *Let* $f(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{(k+1)^n} a_i t_i(x_1, x_2, \ldots, x_n)$, *where* $t_i(x_1, x_2, \ldots, x_n) = x_1^{i_1} x_2^{i_2}, \ldots, x_n^{i_n}$, $0 \le i_1, i_2, \ldots, i_n \le k$. *Then f and* f^* *are identical if they agree on any set of* $m = (k + 1)^n$ *values* $\{\langle c_{i,1}, c_{i,2}, \ldots, c_{i,n}\rangle \mid i = 1, 2, \ldots, m\}$ *such that the matrix* $M = [b_{ij}]$ *is nonsingular, where* $b_{ij} = t_i(c_{j,1}, c_{j,2}, \ldots, c_{j,n})$.

Definition 4.5 (*Functional Adequacy*) [Howden 1978; 1987]. Let $\{D_1, D_2, \ldots, D_n\}$ be the set of subdomains of software S. Suppose that the required function on subdomain D_i is a multinomial in m variables of degree k_i, $i = 1, 2, \ldots, n$. A set T of test cases is said to be functional-test adequate if, for each subdomain D_i, $i = 1, 2, \ldots, n$, T contains at least $(k_i + 1)^m$ test cases $c_j = \langle c_{j,1}, c_{j,2}, \ldots, c_{j,m}\rangle$ in the subdomain D_i such that the matrix $T = [b_{ij}]$ is nonsingular, where $b_{ij} = t_i(c_{j,1}, c_{j,2}, \ldots, c_{j,m})$, and t_i is the same as in the preceding theorem.

4.5 Summary of Domain-Analysis and Error-Based Test Adequacy Criteria

The basic idea behind domain analysis is the classification of program errors into two types: computation errors and domain errors. A computation error is reflected by an incorrect function computed by the program. Such an error may be caused, for example, by the execution of an inappropriate assignment statement that affects the function computed within a path in the program. A domain error may occur, for instance, when a branch predicate is expressed incorrectly or an assignment statement that affects a branch predicate is wrong, thus affecting the conditions under which the path is selected. A boundary-analysis adequacy criterion focuses on the correctness of the boundaries, which are sensitive to domain errors. A functional-analysis criterion focuses on the correctness of the computation, which is sensitive to computation errors. They should be used in a complementary fashion.

It is widely recognized that software testing should take both specification and program into account. A way to combine program-based and specification-based domain-testing techniques is first to partition the input space using the two methods separately and then refine the partition by intersection of the subdomains [Gourlay 1983; Weyuker and Ostrand 1980; Richardson and Clarke 1985]. Finally, for each subdomain in the refined partition, the required function and the computed function are checked to see if they belong to the same set of functions, say, polynomials of degree K, and the test cases are selected according to the set of functions to which they belong.

A limitation of domain-analysis techniques is that they are too complicated to be applicable to software that has a complex input space. For example, process control software may have sequences of interactions between the software and the environment system. It may be difficult to partition the input space into subdomains.

Another shortcoming of boundary-analysis techniques is that they were proposed for numerical input space such that the notions of the closeness of two points in the $V \times V$ adequacy, and "just off a border" in the $N \times N$ and $N \times 1$ adequacy can be formally defined. However, it is not so simple to extend these notions to nonnumerical software such as compilers.

5. COMPARISON OF TEST DATA ADEQUACY CRITERIA

Comparison of testing methods has always been desirable. It is notoriously difficult because testing methods are defined using different models of software and based on different theoretical foundations. The results are often controversial.

In comparing testing adequacy criteria, it must be made very clear in what sense one criterion is better than another. There are three main types of such measures in the software testing literature: fault-detecting ability, software reliability, and test cost. This section reviews the methods and the main results of the comparisons.

5.1 Fault-Detecting Ability

Fault-detecting ability is one of the most direct measures of the effectiveness of test adequacy criteria [Basili and Selby 1987; Duran and Ntafos 1984; Frankl and Weiss 1993; Frankl and Weyuker 1988; Ntafos 1984; Weyuker and Jeng 1991; Woodward et al. 1980]. The methods to compare test adequacy criteria according to this measure can be classified into three types: statistical experiment, simulation, and formal analysis. The following summarizes research in these three approaches.

5.1.1 Statistical Experiments. The basic form of statistical experiments with test adequacy criteria is as follows.

Let $C_1, C_2, \ldots, C_n$ be the test adequacy criteria under comparison. The experiment starts with the selection of a set of sample programs, say, $P_1, P_2, \ldots, P_m$. Each program has a collection of faults that are known due to previous experience with the software, or planted artificially, say, by applying mutation operations. For each program P_i, $i = 1, 2, \ldots, m$, and adequacy criterion C_j, $j = 1, 2, \ldots, n$, k test sets $T^j_{i_1}, T^j_{i_2}, \ldots, T^j_{i_k}$ are generated in some fashion so that $T^j_{i_u}$ is adequate to test program P_i according to the criterion C_j. The proportion of faults $r^j_{i_u}$ detected by the test set $T^j_{i_u}$ over the known faults in the program P_i is calculated for every $i = 1, 2, \ldots, n$, $j = 1, 2, \ldots, m$, and $u = 1, 2, \ldots, k$. Statistical inferences are then made based on the data $r^j_{i_u}$, $i = 1, \ldots, n$, $j = 1, \ldots, m$, $u = 1, 2, \ldots, k$.

For example, Ntafos [1984] compared branch coverage, random testing, and required pair coverage. He used 14 small programs. Test cases for each program were selected from a large set of random test cases and modified as needed to satisfy each of the three testing strategies. The percentages of mutants killed by the test sets were considered the fault-detection abilities.

Hamlet [1989] pointed out two potential invalidating factors in this method. They are:

(1) A particular collection of programs must be used—it may be too small or too peculiar for the results to be trusted.
(2) Particular test data must be created for each method—the data may have

good or bad properties not related to the testing method.

Hamlet [1989] was also critical: "It is not unfair to say that a typical testing experiment uses a small set of toy programs, with uncontrolled human generation of the test data. That is, neither (1) nor (2) is addressed."

In addition to these two problems, there is, in fact, another potential invalidating factor in the method. That is:

(3) A particular collection of known faults in each program must be used—it may not be representative of the faults in software. The faults could be too easy or too difficult to find, or there could be a different distribution of the types of faults. One particular test method or adequacy criterion may have a better ability to detect one type of fault than other types.

To avoid the effects of the potential invalidating factors related to the test data, Basili and Selby [1987] used fractional factorial design methodology of statistical experiments and repeated experimentation. They compared two dynamic testing methods (functional testing and statement coverage) and a static testing method (code review) in three iterative phases involving 74 subjects (i.e., testers) of various backgrounds. However, they only used four sample programs. The faults contained in the sample programs include those made during the actual development of the program as well as artificial faults. Not only the fault-detection ability, but also the fault-detection cost with respect to various classes of faults were compared. Since the test sets for dynamic testing are generated manually and the static testing is performed manually, they also made a careful comparison of human factors in testing. Their main results were:

—*Human factors.* With professional programmers, code reading detected more software faults and yielded a higher fault-detection rate than functional or structural testing did. Although functional testing detected more faults than structural testing did, functional testing and structural testing did not differ in fault-detection rate. In contrast, with advanced students, the three techniques were not different in fault-detection rate.

—*Software type.* The number of faults observed, fault-detection rate, and total effort in software testing all clearly depended on the type of software under test.

—*Fault type.* Different testing techniques have different fault-detection abilities for different types of faults. Code reading detected more interface faults than the other methods did, and functional testing detected more control faults than the other methods did.

Their experiment results indicated the complexity of the task of comparing software testing methods.

Recently, Frankl and Weiss [1993] used another approach to addressing the potential invalidating factor associated with the test data. They compared branch adequacy and all-uses data-flow adequacy criteria using nine small programs of different subjects. Instead of using one adequate test set for each criterion, they generated a large number of adequate test sets for each criterion and calculated the proportion of the test sets that detect errors as an estimate of the probability of detecting errors. Their main results were:

—for five of the nine subjects, the all-uses criterion was more effective than branch coverage at 99% confidence, where the effectiveness of an adequacy criterion is the probability that an adequate test set exposes an error when the test set is selected randomly according to a given distribution on the adequate test sets of the criterion.

—in four of the nine subjects, all-uses adequate test sets were more effective than branch-coverage adequate sets of similar size.

—for the all-uses criterion, the probability of detecting errors clearly depends on the extent of the adequacy only when the adequacy is very close to 100% (i.e., higher than 97%). In four of the subjects, such a dependence existed for the branch-coverage criterion over all the adequacy range.

The advantage of the statistical experiment approach is that it is not limited to comparing adequacy criteria based on a common model. However, given the potential invalidating factors, it is doubtful that it is capable of producing trustworthy results on pairs of adequacy criteria with subtle differences.

5.1.2 *Simulation.* Both simulation and formal analysis are based on a certain simplified model of software testing. The most famous research that uses simulation for comparing software test adequacy criteria are Duran and Ntafos' [1984] comparison of partition testing with random testing and Hamlet and Taylor's [1990] repetition of the comparison.

Duran and Ntafos modeled partition testing methods by a finite number of disjoint subsets of software input spaces. Suppose that the input space D of a given software system is partitioned into k disjoint subsets $D_1, D_2, \ldots, D_k$ and an input chosen at random has probability p_i of lying in D_i. Let θ_i be the failure rate for D_i, then

$$\theta = \sum_{i=1}^{k} p_i \theta_i \tag{5.1}$$

is the probability that the program will fail to execute correctly on an input chosen at random in accordance with the input distribution. Duran and Ntafos investigated the probability that a test set will reveal one or more errors and the expected number of errors that a set of test cases will discover.[6] For random testing, the probability P_r of finding at least one error in n tests is $1 - (1 - \theta)^n$. For a partition testing method in which n_i test cases are chosen at random from subset D_i, the probability P_p of finding at least one error is given by

$$P_p = 1 - \prod_{i=1}^{k} (1 - \theta_i)^{n_i}. \tag{5.2}$$

With respect to the same partition,

$$P_r = 1 - (1 - \theta)^n = 1 - \left(1 - \sum_{i=1}^{k} p_i \theta_i\right)^n, \quad \text{where } n = \sum_{i=1}^{k} n_i. \tag{5.3}$$

The expected number E_p of errors discovered by partition testing is given by

$$E_p(k) = \sum_{i=1}^{k} \theta_i \tag{5.4}$$

if one random test case is chosen from each D_i, where θ_i is the failure rate for D_i.

If in total n random test cases are used in random testing, some set of actual values $\mathbf{n} = \{n_1, n_2, \ldots, n_k\}$ will occur, where n_i is the number of test cases that fall in D_i. If $\mathbf{n}$ were known, then the expected number of errors found would be

$$E(\mathbf{n}, k, n) = \sum_{i=1}^{k} (1 - (1 - \theta_i)^{n_i}). \tag{5.5}$$

[6] Note that an error means that the program's output on a specific input does not meet the specification of the software.

Since $\mathbf{n}$ is not known, the expected number E_r of errors found by n random tests is

$$E_r(k, n) = \sum_{\mathbf{n}} \left(E(\mathbf{n}, k, n) \cdot n! \cdot \left(\prod_{i=1}^{k} p_i^{n_i} \bigg/ \prod_{i=1}^{k} n_i \right) \right) = k - \sum_{i=1}^{k} (1 - p_i \theta_i)^n. \qquad (5.6)$$

They conducted simulations of various program conditions in terms of θ_i distributions and K, and calculated and compared the values of P_p, P_r, E_p, and E_r. Two types of θ_i distributions were investigated. The first type, called hypothetical distributions, considered the situation where a test case chosen from a subset would have a high probability of finding an error affecting that subset. Therefore the θ_is were chosen from a distribution such that 2 percent of the time $\theta_i \geq 0.98$ and 98 percent of the time, $\theta_i < 0.049$. The second type of θ_i distribution was uniform distributions; that is, the θ_is were allowed to vary uniformly from 0 to a value $\theta_{\max}$ that varies from 0.01 to 1.

The main result of their simulation was that when the fault-detecting ability of 100 simulated random test cases was compared to that of 50 simulated partition test cases, random testing was superior. This was considered as evidence to support one of the main conclusions of the paper, that random testing would be more cost-effective than partition testing, because performing 100 random tests was considered less expensive than 50 partition tests.

Considering Duran and Ntafos' result counterintuitive, Hamlet and Taylor [1990] did more extensive simulation and arrived at more precise statements about the relationship between partition probabilities, failure rates, and effectiveness. But their results corroborated Duran and Ntafos' results.

In contrast to statistical experiment, imulation can be performed in an ideal testing scenario to avoid some of the complicated human factors. However, because simulation is based on a certain simplified model of software testing, the realism of the simulation result could be questionable. For example, in Duran and Ntafos' experiment, the choice of the particular hypothetical distribution (i.e., for 2 percent of the time $\theta_i \geq 0.98$ and for 98 percent of the time $\theta_i < 0.049$) seems rather arbitrary.

5.1.3 *Formal Analysis of the Relationships among Adequacy Criteria.* One of the basic approaches to comparing adequacy criteria is to define some relation among adequacy criteria and to prove that the relation holds or does not hold for various pairs of criteria. The majority of such comparisons in the literature use the subsume ordering, which is defined as follows.

Definition 5.1 (*Subsume Relation among Adequacy Criteria*). Let C_1 and C_2 be two software test data adequacy criteria. C_1 is said to subsume C_2 if for all programs p under test, all specifications s and all test sets t, t is adequate according to C_1 for testing p with respect to s implies that t is adequate according to C_2 for testing p with respect to s.

Rapps and Weyuker [1985] studied the subsume relation among their original version of data-flow adequacy criteria, which were not finitely applicable. Frankl and Weyuker [1988] later used this relation to compare their revised feasible version of data-flow criteria. They found that feasibility affects the subsume relation among adequacy criteria. In Figure 7, the subsume relation that holds for the infeasible version of criteria but does not hold for the feasible version of the criteria is denoted by an arrow marked with the symbol "*". Ntafos [1988] also used this relation to compare all the data-flow criteria and several other structural coverage criteria. Among many other works on comparing testing methods by subsume re-

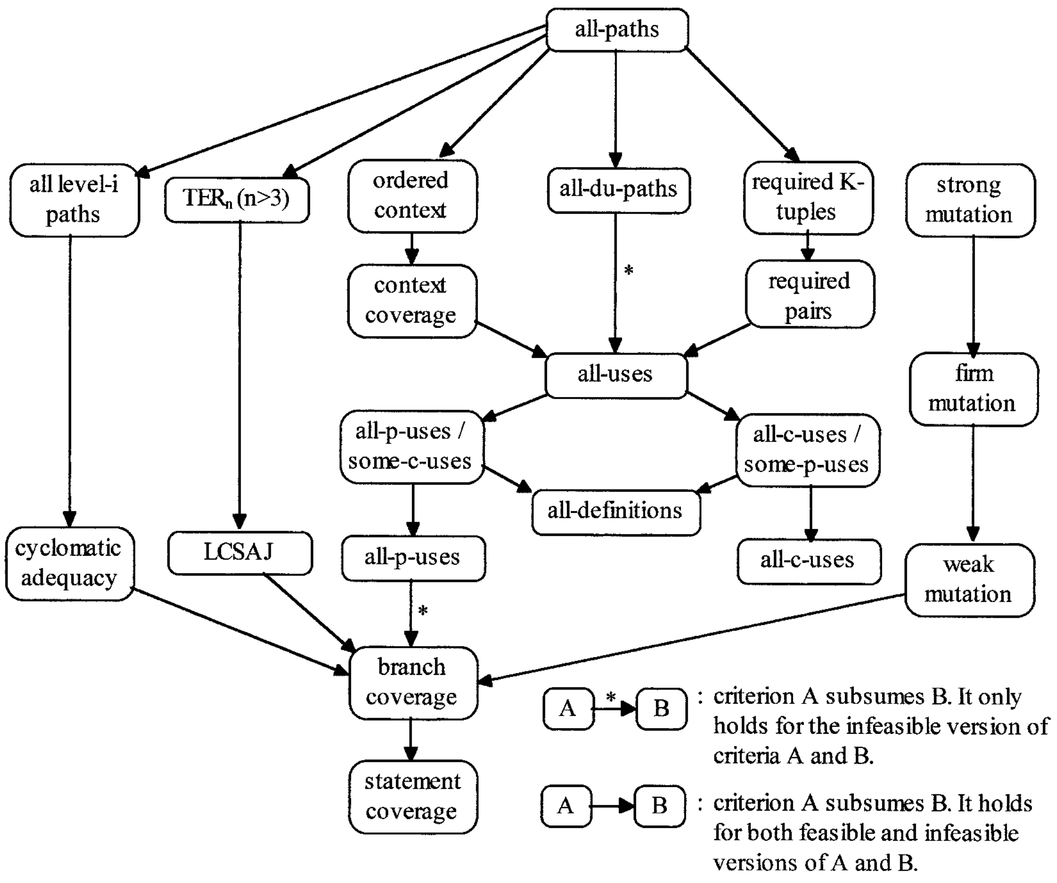

Figure 7. Subsume relation among adequacy criteria.

lation are those by Weiser et al. [1985] and Clark et al. [1989]. Since the adequacy criteria based on flow analysis have a common basis, most of them can be compared; they fall into a fairly simple hierarchy, as shown in Figure 7. A relatively complete picture of the relations among test adequacy criteria can be built. However, many methods are incomparable; that is, one does not subsume the other.

The subsume relation is actually a comparison of adequacy criteria according to the severity of the testing methods. The subsume relation evaluates adequacy criteria in their own terms, without regard for what is really of interest. The relation expresses nothing about the ability to expose faults or to assure software quality. Frankl and Weyuker [1993a] proved that the fact that C_1 subsumes C_2 does not always guarantee that C_1 is better at detecting faults.

Frankl and Weyuker investigated whether a relation on adequacy criteria can guarantee better fault-detecting ability for subdomain testing methods. The testing methods they considered are those by which the input space is divided into a multiset of subspaces so that in each subspace at least one test case is required. Therefore a subdomain test adequacy criterion is equivalent to a function that takes a program p and a specification s and gives a multiset[7] $\{D_1, D_2, \ldots, D_k\}$ of subdomains. They

[7] In a multiset an element may appear more than once.

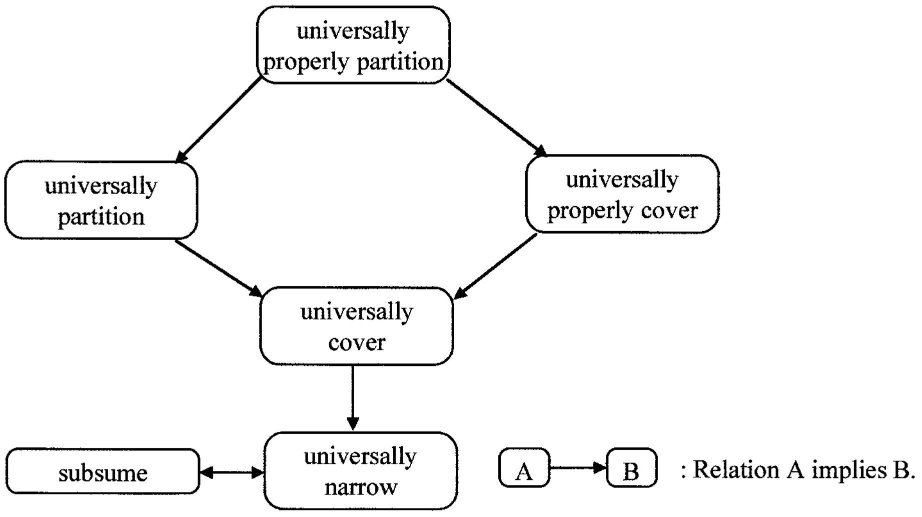

Figure 8. Relationships among partial orderings on adequacy criteria.

defined a number of partial orderings on test adequacy criteria and studied whether these relations are related to the ability of fault detection. The following are the relations they defined, which can be regarded as extensions of the subsume relation.

Definition 5.2 (*Relations among Test Adequacy Criteria*). Let C_1 and C_2 be two subdomain test adequacy criteria and $\pi_{c1}(p, s)$ and $\pi_{c2}(p, s)$ be the multiset of the subsets of the input space for a program p and a specification s according to C_1 and C_2, respectively.

—C_1 *narrows* C_2 for (p, s) if for every $D \in \pi_{C2}(p, s)$, there is a subdomain $D' \in \pi_{C1}(p, s)$ such that $D' \subseteq D$.

—C_1 *covers* C_2 for (p, s) if for every $D \in \pi_{C2}(p, s)$, there is a nonempty collection of subdomains $\{D_1, D_2, \ldots, D_n\}$ belonging to $\pi_{C1}(p, s)$ such that $D_1 \cup D_2 \cup \ldots D_n = D$.

—C_1 *partitions* C_2 for (p, s) if for every $D \in \pi_{C2}(p, s)$, there is a nonempty collection of pairwise disjoint subdomains $\{D_1, D_2, \ldots, D_n\}$ belonging to $\pi_{C1}(p, s)$ such that $D_1 \cup D_2 \cup \ldots \cup D_n = D$.

—Let $\pi_{C1}(p, s) = \{D_1, D_2, \ldots, D_m\}$, $\pi_{C2}(p, s) = \{E_1, E_2, \ldots, E_n\}$. C_1 *properly covers* C_2 for (p, s) if there is a multiset $M = \{D_{1,1}, D_{1,2}, \ldots, D_{1,k1}, \ldots, D_{n,1}, \ldots, D_{n,kn}\} \subseteq \pi_{C1}(p, s)$ such that $E_i = D_{i,1} \cup D_{i,2} \cup \ldots \cup D_{i,ki}$ for all $i = 1, 2, \ldots, n$.

—Let $\pi_{C1}(p, s) = \{D_1, D_2, \ldots, D_m\}$, $\pi_{C2}(p, s) = \{E_1, E_2, \ldots, E_n\}$. C_1 *properly partitions* C_2 for (p, s) if there is a multiset $M = \{D_{1,1}, \ldots, D_{1,k1}, \ldots, D_{n,1}, \ldots, D_{n,kn}\} \subseteq \pi_{C1}(p, s)$ such that $E_i = D_{i,1} \cup D_{i,2} \cup \ldots \cup D_{i,ki}$ for all $i = 1, 2, \ldots, n$, and for each $i = 1, 2, \ldots, n$, the collection $\{D_{i,1}, \ldots, D_{i,ki}\}$ is pairwise disjoint.

For each relation R previously defined, Frankl and Weyuker defined a stronger relation, called universal R, which requires R to hold for all specifications s and all programs p. These relations have the relationships shown in Figure 8.

To study fault-detecting ability, Frankl and Weyuker considered two idealized strategies for selecting the test cases. The first requires the tester to independently select test cases at random from the whole input space according to a uniform distribution until the adequacy criterion is satisfied. The second strategy assumes that the input space has first been divided into subdo-

Table III. Relationships among Fault-Detecting Ability and Orderings on Adequacy Criteria (k_i, i = 1, 2, is the number of subdomains in $\pi_{C_i}(p, s)$)

	$M_1(C_1) \geq M_1(C_2)$?	$M_2(C_1) \geq M_2(C_2)$?	$M_3(C_1, 1) \geq M_3(C_2, k_1/k_2)$?	$E(C_1) \geq E(C_2)$?
C_1 narrows C_2	sometimes	sometimes	sometimes	--
C_1 covers C_2	sometimes	sometimes	sometimes	--
C_1 partitions C_2	always	sometimes	sometimes	--
C_1 properly covers C_2	sometimes	always	sometimes	always
C_1 properly partitions C_2	always	always	sometimes	always

mains and then requires independent random selection of a predetermined number n of test cases from each subdomain. They used three different measures, M_1, M_2, and M_3 in the following definition of fault-detecting ability. In a later paper Frankl and Weyuker [1993b] also studied the expected number of errors detected by a test method. This measure is $E(C, p, s)$ in the following definition. Notice that these measures are the same as those used by Duran and Ntafos [1984] except θ_i is replaced by m_i/d_i; see also Section 5.1.2.

Definition 5.3 (*Measures of Fault-Detection Ability*). Let $\pi_c(p, s) = \{D_1, D_2, \ldots, D_k\}$ be the multiset of the subdomains of the input space of (p, s) according to the criterion C, $d_i = |D_i|$, and m_i be the number of fault-causing inputs in D_i, $i = 1, \ldots, k$. Define:

$$M_1(C, p, s) = \max_{1 \leq i \leq k}\left(\frac{m_i}{d_i}\right) \quad (5.7)$$

$$M_2(C, p, s) = 1 - \prod_{i=1}^{k}\left(1 - \frac{m_i}{d_i}\right) \quad (5.8)$$

$$M_3(C, p, s, n) = 1 - \prod_{i=1}^{k}\left(1 - \frac{m_i}{d_i}\right)^n, \quad (5.9)$$

where $n \geq 1$ is the number of test cases in each subdomain.

$$E(C, p, s) = \sum_{i=1}^{k} \frac{m_i}{d_i}. \quad (5.10)$$

According to Frankl and Weyuker, the first measure $M_1(C, p, s)$ gives a crude lower bound on the probability that an adequate test set selected using either of the two strategies will expose at least one fault. The second measure $M_2(C, p, s)$ gives the exact probability that a test set chosen by using the first sampling strategy will expose at least one fault. $M_3(C, p, s, n)$ is a generalization of M_2 by taking the number n of test cases selected in each subdomain into account. It should be noted that M_2 and M_3 are special cases of Equation (5.2) for P_p in Duran and Ntafos' work that $n_i = 1$ and $n_i = n$, respectively. It is obvious that M_2 and M_3 are accurate only for the first sampling strategy because for the second strategy it is unreasonable to assume that each subdomain has exactly n random test cases.

Frankl and Weyuker [1993a,b] proved that different relations on the adequacy criteria do relate to the fault-detecting ability. Table III summarizes their results.

Given the fact that the universal narrows relation is equivalent to the subsume relation for subdomain testing methods, Frankl and Weyuker concluded that "C_1 subsumes C_2" does not guarantee a better fault-detecting ability for C_1. However, recently Zhu [1996a] proved that in certain testing scenarios the subsumes relation can imply better fault-detecting ability. He identified two software testing scenarios. In the first scenario, a software tester is asked to satisfy a particular adequacy criterion and generates test

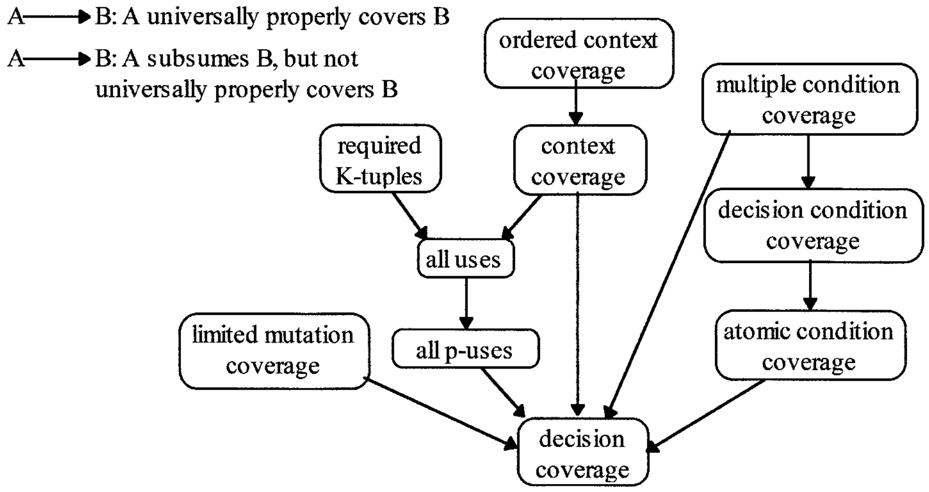
A → B: A universally properly covers B
A → B: A subsumes B, but not universally properly covers B
ordered context coverage
multiple condition coverage
required K-tuples
context coverage
decision condition coverage
all uses
limited mutation coverage
all p-uses
atomic condition coverage
decision coverage

Figure 9. Universally properly cover relation among adequacy criteria.

cases specifically to meet the criterion. This testing scenario is called the prior testing scenario. Frankl and Weyuker's second sampling strategy belongs to this scenario. In the second scenario, the tester generates test cases without any knowledge of test adequacy criterion. The adequacy criterion is only used as a stopping rule so that the tester stops generating test cases only if the criterion is satisfied. This scenario is called posterior testing scenario. Frankl and Weyuker's first sampling strategy belongs to the posterior scenario. It was proved that in the posterior testing scenario, the subsume relation does imply better fault-detecting ability in all the probability measures for error detection and for the expected number of errors. In the posterior testing scenario, the subsume relation also implies more test cases [Zhu 1996a].

Frankl and Weyuker [1993b] also investigated how existing adequacy criteria fit into the partial orderings. They studied the properly cover relation on a subset of data-flow adequacy criteria, a limited mutation adequacy (which is actually branch coverage), and condition coverage. The results are shown in Figure 9.

5.2 Software Reliability

The reliability of a software system that has passed an adequate test is a direct measure of the effectiveness of test adequacy criteria. However, comparing test adequacy criteria with respect to this measure is difficult. There is little work of this type in the literature. A recent breakthrough is the work of Tsoukalas et al. [1993] on estimation of software reliability from random testing and partition testing.

Motivated to explain the observations made in Duran and Ntafos' [1984] experiment on random testing and partition testing, they extended the Thayer-Lipow-Nelson reliability model [Thayer et al. 1978] to take into account the cost of errors, then compared random testing with partition testing by looking at the upper confidence bounds when estimating the cost-weighted failure rate.

Tsoukalas et al. [1993] considered the situation where the input space D is partitioned into k pairwise disjoint subsets so that $D = D_1 \cup D_2 \cup \ldots \cup D_k$. On each subdomain D_i, there is a cost penalty c_i that would be incurred by the program's failure to execute properly on an input from D_i, and a probability p_i

that a randomly selected input will belong to D_i. They defined the cost-weighted failure rate for the program as a whole as

$$C = \sum_{i=1}^{k} c_i p_i \theta_i. \quad (5.11)$$

Usually, the cost-weighted failure rate is unknown in advance, but can be estimated by

$$C \approx \sum_{i=1}^{k} c_i p_i (f_i/n_i), \quad (5.12)$$

where f_i is the number of failures observed over D_i and n_i is the number of random test cases within the subdomain D_i. Given the total number n of test cases, Equation (5.13) gives the maximum likelihood estimate of the cost-weighted failure rate C.

$$C \approx \sum_{i=1}^{k} c_i f_i / n \quad (5.13)$$

But Tsoukalas et al. [1993] pointed out that the real issue is how confident one can be in the estimate. To this end they sought an upper confidence bound on C. Therefore, considering finding the upper bound as a linear programming problem that maximizes $\Sigma_{i=1}^{k} c_i p_i \theta_i$ subject to certain conditions, they obtained expressions for the upper bounds in the two cases of random testing and partition testing and calculated the upper bounds for various special cases.

The conclusion was that confidence is more difficult to achieve for random testing than for partition testing. This agrees with the intuition that it should be easier to have a certain level of confidence for the more systematic of the two testing methods. That is, it is easier to put one's faith in partition testing. But they also confirmed the result of the empirical studies of the effectiveness of random testing by Duran and Ntafos [1984] and Hamlet and Taylor [1990]. They showed that in some cases random testing can perform much better than partition testing, especially when only one test case is selected in each subdomain.

This work has a number of practical implications, especially for safety-critical software where there is a significant penalty for failures in some subdomains. For example, it was proved that when no failures are detected, the optimum way to distribute the test cases over the subdomains in partition testing is to select n_i proportional to $c_i p_i$, $0 \leq i \leq k$, rather than select one test case in each subdomain.

5.3 Test Cost

As testing is an expensive software development activity, the cost of testing to achieve a certain adequacy according to a given criterion is also of importance. Comparison of testing costs involves many factors. One of the simplified measures of test cost is the size of an adequate test set. Weyuker's [1988c, 1993] work on the complexity of data-flow adequacy criteria belongs to this category.

In 1988, Weyuker [1988c] reported an experiment with the complexity of data-flow testing. Weyuker used the suite of programs in *Software Tools in Pascal* by Kernighan and Plauger [1981] to establish empirical complexity estimates for Rapps and Weyuker's family of data-flow adequacy criteria. The suite consists of over 100 subroutines. Test cases were selected by testers who were instructed to select "natural" and "atomic" test cases using the selection strategy of their choice, and were not encouraged to select test cases explicitly to satisfy the selected data-flow criterion. Then seven statistical estimates were computed for each criterion from the data (d_i, t_i) where d_i denoted the number of decision statements in program i and t_i denoted the number of test cases used to satisfy the given criterion for program i. Among the seven measures, the most interesting ones are the least squares line $t = \alpha d + \beta$ (where t is the number of test cases sufficient to satisfy the given criteria for the subject program and d is the number of decision statements in that program), and the weighted average of the ratios of the number of decision statements in a subject program to the number of test cases sufficient to satisfy the selected criterion. Weyuker observed that (1) the coefficients of d in the least-square lines were less than 1 for all the cases and (2) the coefficients were very close to the average number of test cases required for each decision statement. Based on these observations, Weyuker concluded that the experiment results reinforced our intuition that the relationship between the required number of test cases and the program size was linear. A result similar to Weyuker's is also observed for the all *du*-paths criterion by Bieman and Schultz [1992].

The programs in the Kernighan and Plauger suite used in Weyuker's experiment were essentially the same type, well designed and modularized, and hence relatively small in size. Addressing the problem that whether substantially larger, unstructured and modularized programs would require larger amounts of test data relative to their size, Weyuker [1993] conducted another experiment with data-flow test adequacy criteria. She used a subset of ACM algorithms in *Collected Algorithms from ACM* (Vol. 1, ACM Press, New York, 1980) as sample programs that contained known faults and five or more decision statements. In this study, the same information was collected and the same values were calculated. The same conclusion was obtained. However, in this study, Weyuker observed a large proportion of infeasible paths in data-flow testing. The average rates of infeasible paths for the all *du*-path criterion were 49 and 56% for the two groups of programs, respectively. Weyuker pointed out that the large number of infeasible paths implies that assessing the cost of data-flow testing only in terms of the number of test cases needed to satisfy a criterion might yield an optimistic picture of the real effort needed to accomplish the testing.

Offutt et al. [1993] studied the cost of mutation testing with a sample set of 10 programs. They used simple and multiple linear regression models to establish a relationship among the number of generated mutants and the number of lines of code, the number of variables, the number of variable references, and the number of branches. The linear regression models provide a powerful vehicle for finding functional relationships among random variables. The coefficient of determination provides a summary statistic that measures how well the regression equation fits the data, and hence is used to decide whether a relationship between some data exists. Offutt et al. used a statistical package to calculate the coefficient of determination of the following formulas.

$$Y_{\mathrm{mutant}} = \beta_0 + \beta_1 X_{\mathrm{line}} + \beta_2 X_{\mathrm{line}}^2$$

$$Y_{\mathrm{mutant}} = \beta_0 + \beta_1 X_{\mathrm{var}} + \beta_2 X_{\mathrm{var}}^2$$

$$Y_{\mathrm{mutant}} = \beta_0 + \beta_1 X_{\mathrm{var}} + \beta_2 X_{\mathrm{varref}} + \beta_3 X_{\mathrm{var}} X_{\mathrm{varref}},$$

where X_{line} is the number of lines in the program, X_{var} is the number of variables in the program, X_{varref} is the number of variable references in the program, and Y_{mutant} is the number of mutants. They found that for single units, the coefficients of determination of the formulas were 0.96, 0.96, and 0.95, respectively. For programs of multiple units, they established the following formula with a coefficient of determination of 0.91.

$$Y_{\mathrm{mutant}} = \beta_0 + \beta_1 X_{\mathrm{var}} + \beta_2 X_{\mathrm{varref}} + \beta_3 X_{\mathrm{unit}} + \beta_4 X_{\mathrm{var}} X_{\mathrm{varref}},$$

where X_{unit} is the number of units in the program. Therefore their conclusion was that the number of mutants is quadratic. As mentioned in Section 3, Offutt et al. also studied the cost of selective mutation testing and showed that even for 6-selective mutation testing, the number of mutants is still quadratic.

5.4 Summary of the Comparison of Adequacy Criteria

Comparison of test data adequacy criteria can be made with respect to various measures, such as fault detection, confidence in the reliability of the tested software, and the cost of software testing. Various abstract relations among the criteria, such as the subsume relation, have also been proposed and relationships among adequacy criteria have been investigated. The relationship between such relations and fault-detecting ability has also been studied. Recent research results have shown that partition testing more easily achieves high confidence in software reliability than random testing, though random testing may be more cost-effective.

6. AXIOMATIC ASSESSMENT OF ADEQUACY CRITERIA

In Section 5, we have seen two kinds of rationale presented to support the use of one criterion or another, yet there is no clear consensus. The first kind of rationale uses statistical or empirical data to compare testing effectiveness, whereas the second type analytically compares test data adequacy criteria based on certain mathematical models of software testing.

In this section we present a third type of rationale, the axiomatic study of the properties of adequacy criteria. The basic idea is to seek the most fundamental properties of software test adequacy and then check if the properties are satisfied for each particular criterion. Although the idea of using abstract properties as requirements of test adequacy criteria can be found in the literature, such as Baker et al. [1986], Weyuker [1986, 1988a] is perhaps the first who explicitly and clearly employed axiom systems. Work following this direction includes refinement and improvement of the axioms [Parrish and Zweben 1991; 1993; Zhu and Hall 1993; Zhu et al. 1993; Zhu 1995a] as well as analysis and criticism [Hamlet 1989; Zweben and Gourlay 1989].

There are four clearcut roles that axiomatization can play in software testing research, as already seen in physics and mathematics. First, it makes explicit the exact details of a concept or an argument that, before the axiomatization, was either incomplete or unclear. Second, axiomatization helps to isolate, abstract, and study more fully a class of mathematical structures that have recurred in many important contexts, often with quite different surface forms, so that a particular form can draw on results discovered elsewhere. Third, it can provide the scientist with a compact way to conduct a systematic exploration of the implications of the postulates. Finally, the fourth role is the study of what is needed to axiomatize a given empirical phenomenon and what can be axiomatized in a particular form. For instance, a number of computer scientists have questioned whether formal properties can be defined to distinguish one type of adequacy criteria from another when criteria are expressed in a particular form such as predicates [Hamlet 1989; Zweben and Gourlay 1989].

Therefore we believe that axiomatization will improve our understanding of software testing and clarify our notion of test adequacy. For example, as pointed out by Parrish and Zweben [1993], the investigation of the applicability of test data adequacy criteria is an excellent example of the role that axiomatization has played. Applicability was proposed as an axiom by Weyuker in 1986, requiring a criterion to be applicable to any program in the sense of the existence of test sets that satisfy the criterion. In the assessment of test data adequacy criteria against this axiom, the data-flow adequacy criteria originally proposed by Rapps and Weyuker in 1985 were found not applicable. This leads to the redefinition of the criteria and reexamination of the power of the criteria [Frankl and Weyuker 1988]. Weyuker's applicability requirements were defined on the assumption that the program has a finite representable input data space. Therefore any adequate test set is finite. When this assumption is removed and adequacy criteria were considered as measurements, Zhu and Hall [1993] used the term finite applicability to denote the requirement that for any software and any adequacy degree r less than 1 there is a finite test set whose adequacy is greater than r. They also found that finite applicability can be derived from other more fundamental properties of test data adequacy criteria.

6.1 Weyuker's Axiomatization of Program-Based Adequacy Criteria

Weyuker [1986] proposed an informal axiom system of test adequacy criteria. Her original purpose of the system was to present a set of properties of ideal program-based test adequacy criteria and use these properties to assess existing criteria. Regarding test adequacy criteria as stopping rules, Weyuker's axiom system consists of eleven axioms, which were later examined, formalized, and revised by Parrish and Zweben [1991; 1993].

The most fundamental properties of adequacy criteria proposed by Weyuker were those concerning applicability. She distinguished the following three applicability properties of test adequacy criteria.

AXIOM A1 (Applicability). *For every program, there exists an adequate test set.*

Assuming the finiteness of representable points in the input data space, Weyuker rephrased the axiom into the following equivalent form.

AXIOM A1 *For every program, there exists a finite adequate test set.*

Weyuker then analyzed exhaustive testing and pointed out that, although exhaustive testing is adequate, an adequacy criterion should not always ask for exhaustive testing. Hence she defined the following nonexhaustive applicability.

AXIOM A2 (Nonexhaustive Applicability). *There is a program p and a test set t such that p is adequately tested by t and t is not an exhaustive test set.*

Notice that by exhaustive testing Weyuker meant the test set of all representable points of the specification domain. The property of central importance in Weyuker's system is monotonicity.

AXIOM A3 (Monotonicity). *If t is adequate for p and $t \subseteq t'$, then t' is adequate for p.*

AXIOM A4 (Inadequate Empty Set). *The empty set is not adequate for any program.*

Weyuker then studied the relationships among test adequacy and program syntactic structure and semantics. However, her axioms were rather negative. They stated that neither semantic closeness nor syntactic structure closeness are sufficient to ensure that two programs require the same test data. Moreover, an adequately tested program does not imply that the components of the program are adequately tested, nor does adequate testing of components imply adequate testing of the program.

Weyuker assessed some test adequacy criteria against the axioms. The main discovery of the assessment is that the mutation adequacy criterion defined in its original form does not satisfy the monotonicity and applicability axioms, because it requires a correctness condition. The correctness condition requires that the program under test produces correct output on a test set if it is to be considered adequate. For all the adequacy criteria we have discussed, it is possible for a program to fail on a test case after producing correct outputs on an adequate test set. Hence the correctness condition causes a conflict with monotonicity. The correctness condition does not play any fundamental role in mutation adequacy, so Weyuker suggested removal of this condition from the definition.[8]

Aware of the insufficiency of the axiom system, Weyuker [1988a] later proposed three additional axioms. The renaming property requires that a test set be adequate for a program p if and only if it is adequate for a program obtained by systematic renaming variables in p. The complexity property requires that for every natural number n, there is a program p, such that p is adequately tested by a size-n test set but not by any size $(n - 1)$ test set. The statement coverage property requires that all adequate test sets cover all feasible statements of the program under test. These axioms were intended to rule out some "unsuitable notions of adequacy."

Given the fact that Weyuker's axioms are unable to distinguish most adequacy criteria, Hamlet [1989] and Zweben and Gourlay [1989] questioned if the axiomatic approach could provide useful comparison of adequacy criteria. Recently, some of Weyuker's axioms were combined with Baker et al.'s [1986] properties for assessing control-flow adequacy criteria [Zhu 1995a]. Control-flow adequacy criteria of subtle differences were distinguished. The assessment suggested that the cycle combination criterion was the most favorable.

6.2 Parrish and Zweben's Formalization and Analysis of Weyuker's Axioms

Parrish and Zweben [1993; 1991] formalized Weyuker's axioms, but put them in a more general framework that involved specifications as well. They defined the notions of program-independent criteria and specification-independent criteria. A program-independent criterion measures software test adequacy independently of the program, whereas a specification-independent criterion measures software test adequacy independently of the specification. Parrish and Zweben explored the interdependence relations between the notions and the axioms. Their work illustrated the complex levels of analysis that an axiomatic theory of software testing can make possible. Formalization is not merely a mathematical exercise because the basic notions of software testing have to be expressed precisely and clearly so that they can be critically analyzed. For example, informally, a program-limited criterion only requires test data selected in the domain of the program, and a specification-limited criterion requires test data only selected in the domain of the specification. Parrish and Zweben formally defined the notion to be that if a test case falls outside the domain, the test set is considered inadequate. This definition is counterintuitive and conflicts with the monotonicity axiom. A more logical definition is that the test cases outside the domain are not taken into account in determining test adequacy [Zhu and Hall 1993]. Parrish and Zweben [1993] also formalized the notion of the correctness condition such that a test set is inadequate if it contains a test case on which the program is incorrect. They tried to put it into the axiom system by modifying the monotonicity axiom to be applied only to test sets on which the software is correct. However, Weyuker [1986] argued that the correctness condition plays no role in determining test adequacy. It should not be an axiom.

6.3 Zhu and Hall's Measurement Theory

Based on the mathematical theory of measurement, Zhu and Hall [1993; Zhu et al. 1993] proposed a set of axioms for the measurement of software test adequacy. The mathematical theory of measurement is the study of the logical and philosophical concepts underlying measurement as it applies in all sciences. It studies the conceptual structures of

[8] The definition of mutation adequacy used in this article does not use the correctness condition.

measurement systems and their properties related to the validation of the use of measurements [Roberts 1979; Krantz et al. 1971; Suppes et al. 1989; Luce et al. 1990]. Therefore the measurement of software test adequacy should conform to the theory. In fact, recent years have seen rapid growth of rigorous applications of the measurement theory to software metrics [Fenton 1991].

According to the theory, measurement is the assignment of numbers to properties of objects or events in the real world by means of an objective empirical operation. The modern form of measurement theory is representational; that is, numbers assigned to objects/events must represent the perceived relation between the properties of those objects/events. Readers are referred to Roberts' [1979] book for various applications of the theory and Krantz et al.'s trilogy [1971; Suppes et al. 1989; Luce et al. 1990] for a systematic treatment of the subject. Usually, such a theory comprises three main sections: the description of an empirical relational system, a representation theorem, and a uniqueness condition.

An empirical relational system $\mathbf{Q} = (Q, R)$ consists of a set Q of manifestations of the property or attribute and a family of relations $R = \{R_1, R_2, \ldots, R_n\}$ on Q. The family R of relations provides the basic properties of the manifestations with respect to the property to be measured. They are usually expressed as axioms derived from the empirical knowledge of the real world. For software testing, the manifestations are the software tests $P \times S \times T$, which consist of a set of programs P, a set of specifications S, and a set of test sets T. Zhu et al. [1994] defined a relation $\leq$ on the space of software testing such that $t_1 \leq t_2$ means that test t_2 is at least as adequate as t_1. It was argued that the relation has the properties of reflexivity, transitivity, comparability, boundedness, equal inadequacy for empty tests, and equal adequacy for exhaustive tests. Therefore this relation is a total ordering with maximal and minimal elements.

To assign numbers to the objects in the empirical system, a numerical system $\mathbf{N} = (N, G)$ must be defined so that the measurement mapping is a homomorphism from the empirical system to the numerical system. It has been proved that for any numerical system $(N, \leq_N)$ and measurement γ of test adequacy on N, there is a measurement μ on the unit interval of real numbers $([0,1], \leq)$ such that $\gamma = \tau \circ \mu$, where τ is an isomorphism between $(N, \leq_N)$ and $([0,1], \leq)$ [Zhu and Hall 1993; Zhu et al. 1994; Zhu 1995b]. Therefore properties of test adequacy measurements can be obtained by studying adequacy measurements on the real unit interval.

Zhu and Hall's axiom system for adequacy measurements on the real unit interval consists of the following axioms.

AXIOM B1 (Inadequacy of Empty Test Set). *For all programs p and specifications s, the adequacy of the empty test set is 0.*

AXIOM B2 (Adequacy of Exhaustive Testing). *For all programs p and specifications s, the adequacy of the exhaustive test set D is 1.*

AXIOM B3 (Monotonicity). *For all programs p and specifications s, if test set t_1 is a subset of test set t_2, then the adequacy of t_1 is less than or equal to the adequacy of t_2.*

AXIOM B4 (Convergence). *Let $t_1, t_2, \ldots, t_n, \ldots \in T$ be test sets such that $t_1 \subseteq t_2 \subseteq \ldots \subseteq t_n \subseteq \ldots$. Then, for all programs p and specifications s,*

$$\lim_{n\to\infty} C_p^s(t_n) = C_p^s\left(\bigcup_{n=1}^{\infty} t_n\right),$$

where $C_p^s(t)$ is the adequacy of test set t for testing program p with respect to specification s.

AXIOM B5 (Law of Diminishing Returns). *The more a program has been tested, the less a given test set can fur-*

ther contribute to the test adequacy. Formally, for all programs p, specifications s, and test sets t,

$$\forall c, d \in T.(c \subseteq d \Rightarrow C_p^s(t|c) \geq C_p^s(t|d)),$$
$$\text{where } C_p^s(t|c) = C_p^s(t \cup c) - C_p^s(c).$$

This axiom system has been proved to be consistent [Zhu and Hall 1993; Zhu et al. 1994]. From these axioms, a number of properties of test data adequacy criteria can be proved. For example, if a test data adequacy criterion satisfies Axiom B2 and Axiom B4, then it is finitely applicable, which means that for any real number r less than 1, there is a finite test set whose adequacy is greater than or equal to r. Therefore an adequacy criterion can fail to be finitely applicable because of two reasons: not satisfying the adequacy of exhaustive test, such as requiring coverage of infeasible elements, or not satisfying the convergence property such as the path-coverage criterion. Another example of derivable property is subadditivity, which can be derived from the law of diminishing returns. Here an adequacy measurement is subadditive if the adequacy of the union of two test sets t_1 and t_2 is less than or equal to the sum of the adequacy of t_1 and the adequacy of t_2.

The measurement theory of software test adequacy was concerned not only with the properties derivable from the axioms, but also with measurement-theoretical properties of the axiom systems. Zhu et al. [1994] proved a uniqueness theorem that characterizes the admissible transformations between any two adequacy measurements that satisfy the axioms. However, the irregularity of adequacy measurements was also formally proved. That is, not all adequacy measurements can be transformed from one to another. In fact, few existing adequacy criteria are convertible from one to another. According to measurement theory, these two theorems lay the foundation for the investigation of the meaningfulness of statements and statistical operations that involve test adequacy measurements [Krantz et al. 1971; Suppes et al. 1989; Luce et al. 1990].

The irregularity theorem confirms our intuition that the criteria are actually different approximations to the ideal measure of test adequacy. Therefore it is necessary to find the "errors" for each criterion. In an attempt to do so, Zhu et al. [1994] also investigated the properties that distinguish different classes of test adequacy criteria.

6.4 Semantics of Software Test Adequacy

The axioms of software test adequacy criteria characterize the notion of software test adequacy by a set of properties for adequacy. These axioms do not directly answer what test adequacy means. The model theory [Chang and Keisler 1973] of mathematical logic provides a way of assigning meanings to formal axioms. Recently, the model theory was applied to assigning meanings to the axioms of test adequacy criteria [Zhu 1996b, 1995b]. Software testing was interpreted as inductive inference such that a finite number of observations made on the behavior of the software under test are used to inductively derive general properties of the software. In particular, Gold's [1967] inductive inference model of identification in the limit was used to interpret Weyuker's axioms and a relationship between adequate testing and software correctness was obtained in the model [Zhu 1996b]. Valiant's [1984] PAC inductive inference model was used to interpret Zhu and Hall's axioms of adequacy measurements. Relationships between software reliability and test adequacy were obtained [Zhu 1995b].

7. CONCLUSION

Since Goodenough and Gerhart [1975, 1977] pointed out that test criteria are a central problem of software testing, test criteria have been a major focus of the research on software testing. A large number of test data adequacy criteria have been proposed and various rationales have been presented for the support of one or another criterion. In this article, various types of software test adequacy criteria proposed in the literature are reviewed. The research on comparison and evaluation of criteria is also surveyed. The notion of test adequacy is examined and its roles in software testing practice and theoretical research are discussed. Although whether an adequacy criterion captures the true notion of test adequacy is still a matter of controversy, it appears that software test adequacy criteria as a whole are clearly linked to the fault-detecting ability of software testing as well as to the dependability of the software that has passed an adequate test. It was predicted that with the advent of a quality assurance framework based upon ISO 9001, which calls for specific mechanisms for defect removal, the acceptance of more formal measurement of the testing process can now be anticipated [Wichmann 1993]. There is a tendency towards systematic approaches to software testing through using test adequacy criteria.

ACKNOWLEDGMENT

The authors would like to thank the anonymous referees for their valuable comments.

APPENDIX

Glossary of Graph-Theory-Related Terminology

Complete computation path: A complete computation path is a path that starts with the begin node and ends at the end node of the flow graph. It is also called a computation path or execution path.

Concatenation of paths: If $p = (n_1, n_2, \ldots, n_s)$, $q = (n_{s+1}, \ldots, n_t)$ are two paths, and $r = (n_1, n_2, \ldots, n_s, n_{s+1}, \ldots, n_t)$ is also a path, we say that r is the concatenation of p to q and write $r = p \wedge q$.

Cycle: A path $p = (n_1, n_2, \ldots, n_t, n_1)$ is called a cycle.

Cycle-free path: A path is said to be cycle-free if it does not contain cycles as subpaths.

Directed graph, node, and edge: A directed graph consists of a set N of nodes and a set $E \subseteq N \times N$ of directed edges between nodes.

Elementary cycle and simple cycle: If the nodes $n_1, n_2, \ldots, n_t$ in the cycle $(n_1, n_2, \ldots, n_t, n_1)$ are all different, then the cycle is called an elementary cycle. If the edges in the cycle are all different, then it is called a simple cycle.

Elementary path and simple path: A path is called an elementary path if the nodes in the path are all different. It is called a simple path if the edges in the path are all different.

Empty path: A path is called an empty path if its length is 1. In this case, the path contains no edge but only a node, and hence is written as (n_1).

Feasible and infeasible paths: Not all complete computation paths necessarily correspond to an execution of the program. A feasible path is a complete computation path such that there exist input data that can cause the execution of the path. Otherwise, it is an infeasible path.

Flow graph: A flow graph is a directed graph that satisfies the following conditions: (1) it has a unique begin node which has no inward edge; (2) it has a unique end node which has no outward edge; and (3) every node in a flow graph must be on a path from the begin node to the end node.

Flow graph model of program structure: The nodes in a flow graph represent linear sequences of computations. The edges in the flow graph represent control transfers. Each edge is associated with a predicate that represents the condition of control transfer from the computation on the start node of the edge to that on the end node of the edge. The begin node and the end node in the flow graph are where the computation starts and finishes.

Inward edges and outward edges of a node: The edge $\langle n_1, n_2 \rangle$ is an inward edge of the node n_2. It is an outward edge of the node n_1.

Length of a path: The length of a path $(n_1, n_2, \ldots, n_t)$ is t.

Path: A path p in a graph is a sequence $(n_1, n_2, \ldots, n_t)$ of nodes such that $\langle n_i, n_{i+1} \rangle$ is an edge in the graph, for all $i = 1, 2, \ldots, t - 1, t > 0$.

Start node and end node of an edge: The node n_1 is the start node of the edge $\langle n_1, n_2 \rangle$ and n_2 is the end node of the edge.

Start node and end node of a path: The start node of a path $(n_1, n_2, \ldots, n_t)$ is n_1, and n_t is the end node of the path.

Strongly connected graph: A graph is strongly connected if for any two nodes a and b, there exists a path from a to b and a path from b to a.

Subpath: A subsequence $n_u, n_{u+1}, \ldots, n_v$ of the path $p = (n_1, n_2, \ldots, n_t)$ is called a subpath of p, where $1 \leq u \leq v \leq t$.

REFERENCES

ADRION, W. R., BRANSTAD, M. A., AND CHERNIAVSKY, J. C. 1982. Validation, verification, and testing of computer software. *Comput. Surv. 14,* 2 (June), 159–192.

AFIFI, F. H., WHITE, L. J., AND ZEIL, S. J. 1992. Testing for linear errors in nonlinear computer programs. In *Proceedings of the 14th IEEE International Conference on Software Engineering* (May), 81–91.

AMLA, N. AND AMMANN, P. 1992. Using Z specifications in category partition testing. In *Proceedings of the Seventh Annual Conference on Computer Assurance* (June), IEEE, 3–10.

AMMANN, P. AND OFFUTT, J. 1994. Using formal methods to derive test frames in category-partition testing. In *Proceedings of the Ninth Annual Conference on Computer Assurance* (Gaithersburg, MD, June), IEEE, 69–79.

BACHE, R. AND MULLERBURG, M. 1990. Measures of testability as a basis for quality assurance. *Softw. Eng. J.* (March), 86–92.

BAKER, A. L., HOWATT, J. W., AND BIEMAN, J. M. 1986. Criteria for finite sets of paths that characterize control flow. In *Proceedings of the 19th Annual Hawaii International Conference on System Sciences,* 158–163.

BASILI, V. R. AND RAMSEY, J. 1984. Structural coverage of functional testing. Tech. Rep. TR-1442, Department of Computer Science, University of Maryland at College Park, Sept.

BASILI, V. R. AND SELBY, R. W. 1987. Comparing the effectiveness of software testing. *IEEE Trans. Softw. Eng. SE-13,* 12 (Dec.), 1278–1296.

BAZZICHI, F. AND SPADAFORA, I. 1982. An automatic generator for compiler testing. *IEEE Trans. Softw. Eng. SE-8,* 4 (July), 343–353.

BEIZER, B. 1983. *Software Testing Techniques.* Van Nostrand Reinhold, New York.

BEIZER, B. 1984. *Software System Testing and Quality Assurance.* Van Nostrand Reinhold, New York.

BENGTSON, N. M. 1987. Measuring errors in operational analysis assumptions, *IEEE Trans. Softw. Eng. SE-13,* 7 (July), 767–776.

BENTLY, W. G. AND MILLER, E. F. 1993. CT coverage—initial results. *Softw. Quality J. 2,* 1, 29–47.

BERNOT, G., GAUDEL, M. C., AND MARRE, B. 1991. Software testing based on formal specifications: A theory and a tool. *Softw. Eng. J.* (Nov.), 387–405.

BIEMAN, J. M. AND SCHULTZ, J. L. 1992. An empirical evaluation (and specification) of the all *du*-paths testing criterion. *Softw. Eng. J.* (Jan.), 43–51.

BIRD, D. L. AND MUNOZ, C. U. 1983. Automatic generation of random self-checking test cases. *IBM Syst. J. 22,* 3.

BOUGE, L., CHOQUET, N., FRIBOURG, L., AND GAUDEL, M.-C. 1986. Test set generation from algebraic specifications using logic programming. *J. Syst. Softw. 6,* 343–360.

BUDD, T. A. 1981. Mutation analysis: Ideas, examples, problems and prospects. In *Computer Program Testing,* Chandrasekaran and Radicchi, Eds., North Holland, 129–148.

BUDD, T. A. AND ANGLUIN, D. 1982. Two notions of correctness and their relation to testing. *Acta Inf.* 18, 31–45.

BUDD, T. A., LIPTON, R. J., SAYWARD, F. G., AND DEMILLO, R. A. 1978. The design of a prototype mutation system for program testing. In *Proceedings of National Computer Conference,* 623–627.

CARVER, R. AND KUO-CHUNG, T. 1991. Replay and testing for concurrent programs. *IEEE Softw.* (March), 66–74.

CHAAR, J. K., HALLIDAY, M. J., BHANDARI, I. S., AND CHILLAREGE, R. 1993. In-process evaluation for software inspection and test. *IEEE Trans. Softw. Eng. 19,* 11, 1055–1070.

CHANDRASEKARAN, B. AND RADICCHI, S. (EDS.) 1981. *Computer Program Testing,* North-Holland.

CHANG, C. C. AND KEISLER, H. J. 1973. *Model Theory.* North-Holland, Amsterdam.

CHANG, Y.-F. AND AOYAMA, M. 1991. Testing the limits of test technology. *IEEE Softw.* (March), 9–11.

CHERNIAVSKY, J. C. AND SMITH, C. H. 1987. A recursion theoretic approach to program testing. *IEEE Trans. Softw. Eng. SE-13,* 7 (July), 777–784.

CHERNIAVSKY, J. C. AND SMITH, C. H. 1991. On Weyuker's axioms for software complexity measures. *IEEE Trans. Softw. Eng. SE-17,* 6 (June), 636–638.

CHOI, B., MATHUR, A., AND PATTISON, B. 1989. PMothra: Scheduling mutants for execution on a hypercube. In *Proceedings of SIGSOFT Symposium on Software Testing, Analysis and Verification 3* (Dec.) 58–65.

CHUSHO, T. 1987. Test data selection and quality estimation based on the concept of essential branches for path testing. *IEEE Trans. Softw. Eng. SE-13,* 5 (May), 509–517.

CLARKE, L. A., HASSELL, J., AND RICHARDSON, D. J. 1982. A close look at domain testing. *IEEE Trans. Softw. Eng. SE-8,* 4 (July), 380–390.

CLARKE, L. A., PODGURSKI, A., RICHARDSON, D. J., AND ZEIL, S. J. 1989. A formal evaluation of data flow path selection criteria. *IEEE Trans. Softw. Eng. 15,* 11 (Nov.), 1318–1332.

CURRIT, P. A., DYER, M., AND MILLS, H. D. 1986. Certifying the reliability of software. *IEEE Trans. Softw. Eng. SE-6,* 1 (Jan.) 2–13.

DAVIS, M. AND WEYUKER, E. 1988. Metric space-based test-data adequacy criteria. *Comput. J. 13,* 1 (Feb.), 17–24.

DEMILLO, R. A. AND MATHUR, A. P. 1990. On the use of software artifacts to evaluate the effectiveness of mutation analysis for detecting errors in production software. In *Proceedings of 13th Minnowbrook Workshop on Software Engineering* (July 24–27, Blue Mountain Lake, NY), 75–77.

DEMILLO, R. A. AND OFFUTT, A. J. 1991. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng. 17,* 9 (Sept.), 900–910.

DEMILLO, R. A. AND OFFUTT, A. J. 1993. Experi mental results from an automatic test case generator. *ACM Trans. Softw. Eng. Methodol. 2,* 2 (April), 109–127.

DEMILLO, R. A., GUINDI, D. S., MCCRACKEN, W. M., OFFUTT, A. J., AND KING, K. N. 1988. An extended overview of the Mothra software testing environment. In *Proceedings of SIGSOFT Symposium on Software Testing, Analysis and Verification 2,* (July), 142–151.

DEMILLO, R. A., LIPTON, R. J., AND SAYWARD, F. G. 1978. Hints on test data selection: Help for the practising programmer. *Computer 11,* (April), 34–41.

DEMILLO, R. A., MCCRACKEN, W. M., MATIN, R. J., AND PASSUFIUME, J. F. 1987. *Software Testing and Evaluation,* Benjamin-Cummings, Redwood City, CA.

DENNEY, R. 1991. Test-case generation from Prolog-based specifications. *IEEE Softw.* (March), 49–57.

DIJKSTRA, E. W. 1972. Notes on structured programming. In *Structured Programming,* by O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Academic Press.

DOWNS, T. 1985. An approach to the modelling of software testing with some applications. *IEEE Trans. Softw. Eng. SE-11,* 4 (April), 375–386.

DOWNS, T. 1986. Extensions to an approach to the modelling of software testing with some performance comparisons. *IEEE Trans. Softw. Eng. SE-12,* 9 (Sept.), 979–987.

DOWNS, T. AND GARRONE, P. 1991. Some new models of software testing with performance comparisons. *IEEE Trans. Rel. 40,* 3 (Aug.), 322–328.

DUNCAN, I. M. M. AND ROBSON, D. J. 1990. Ordered mutation testing. *ACM SIGSOFT Softw. Eng. Notes 15,* 2 (April), 29–30.

DURAN, J. W. AND NTAFOS, S. 1984. An evaluation of random testing. *IEEE Trans. Softw. Eng. SE-10,* 4 (July), 438–444.

FENTON, N. 1992. When a software measure is not a measure. *Softw. Eng. J.* (Sept.), 357–362.

FENTON, N. E. 1991. Software metrics—a rigorous approach. Chapman & Hall, London.

FENTON, N. E., WHITTY, R. W., AND KAPOSI, A. A. 1985. A generalised mathematical theory of structured programming. *Theor. Comput. Sci. 36,* 145–171.

FORMAN, I. R. 1984. An algebra for data flow anomaly detection. In *Proceedings of the Seventh International Conference on Software Engineering* (Orlando, FL), 250–256.

FOSTER, K. A. 1980. Error sensitive test case analysis (ESTCA). *IEEE Trans. Softw. Eng. SE-6,* 3 (May), 258–264.

FRANKL, P. G. AND WEISS, S. N. 1993. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Trans. Softw. Eng. 19,* 8 (Aug.), 774–787.

FRANKL, P. G. AND WEYUKER, J. E. 1988. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng. SE-14,* 10 (Oct.), 1483–1498.

FRANKL, P. G. AND WEYUKER, J. E. 1993a. A formal analysis of the fault-detecting ability of testing methods. *IEEE Trans. Softw. Eng. 19,* 3 (March), 202–213.

FRANKL, P. G. AND WEYUKER, E. J. 1993b. Provable improvements on branch testing. *IEEE Trans. Softw. Eng. 19,* 10, 962–975.

FREEDMAN, R. S. 1991. Testability of software components. *IEEE Trans. Softw. Eng. SE-17,* 6 (June), 553–564.

FRITZSON, P., GYIMOTHY, T., KAMKAR, M., AND SHAHMEHRI, N. 1991. Generalized algorithmic debugging and testing. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, June 26–28).

FUJIWARA, S., V. BOCHMANN, G., KHENDEK, F., AMALOU, M., AND GHEDAMSI, A. 1991. Test selection based on finite state models. *IEEE Trans. Softw. Eng. SE-17,* 6 (June), 591–603.

GAUDEL, M.-C. AND MARRE, B. 1988. Algebraic specifications and software testing: Theory and application. In Rapport LRI 407.

GELPERIN, D. AND HETZEL, B. 1988. The growth of software testing. *Commun. ACM 31,* 6 (June), 687–695.

GIRGIS, M. R. 1992. An experimental evaluation of a symbolic execution system. *Softw. Eng. J.* (July), 285–290.

GOLD, E. M. 1967. Language identification in the limit. *Inf. Cont. 10,* 447–474.

GOODENOUGH, J. B. AND GERHART, S. L. 1975. Toward a theory of test data selection. *IEEE Trans. Softw. Eng. SE-3* (June).

GOODENOUGH, J. B. AND GERHART, S. L. 1977. Toward a theory of testing: Data selection criteria. In *Current Trends in Programming Methodology,* Vol. 2, R. T. Yeh, Ed., Prentice-Hall, Englewood Cliffs, NJ, 44–79.

GOPAL, A. AND BUDD, T. 1983. Program testing by specification mutation. Tech. Rep. TR 83-17, University of Arizona, Nov.

GOURLAY, J. 1983. A mathematical framework for the investigation of testing. *IEEE Trans. Softw. Eng. SE-9,* 6 (Nov.), 686–709.

HALL, P. A. V. 1991. Relationship between specifications and testing. *Inf. Softw. Technol. 33,* 1 (Jan./Feb.), 47–52.

HALL, P. A. V. AND HIERONS, R. 1991. Formal methods and testing. Tech. Rep. 91/16, Dept. of Computing, The Open University.

HAMLET, D. AND TAYLOR, R. 1990. Partition testing does not inspire confidence. *IEEE Trans. Softw. Eng. 16* (Dec.), 206–215.

HAMLET, D., GIFFORD, B., AND NIKOLIK, B. 1993. Exploring dataflow testing of arrays. In *Proceedings of 15th ICSE* (May), 118–129.

HAMLET, R. 1989. Theoretical comparison of testing methods. In *Proceedings of SIGSOFT Symposium on Software Testing, Analysis, and Verification 3* (Dec.), 28–37.

HAMLET, R. G. 1977. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng. 3,* 4 (July), 279–290.

HARROLD, M. J., MCGREGOR, J. D., AND FITZPATRICK, K. J. 1992. Incremental testing of object-oriented class structures. In *Proceedings of 14th ICSE* (May) 68–80.

HARROLD, M. J. AND SOFFA, M. L. 1990. Interprocedural data flow testing. In *Proceedings of SIGSOFT Symposium on Software Testing, Analysis, and Verification 3* (Dec.), 158–167.

HARROLD, M. J. AND SOFFA, M. L. 1991. Selecting and using data for integration testing. *IEEE Softw.* (March), 58–65.

HARTWICK, D. 1977. Test planning. In *Proceedings of National Computer Conference,* 285–294.

HAYES, I. J. 1986. Specification directed module testing. *IEEE Trans. Softw. Eng. SE-12,* 1 (Jan.), 124–133.

HENNELL, M. A., HEDLEY, D., AND RIDDELL, I. J. 1984. Assessing a class of software tools. In *Proceedings of the Seventh ICSE,* 266–277.

HERMAN, P. 1976. A data flow analysis approach to program testing. *Aust. Comput. J. 8,* 3 (Nov.), 92–96.

HETZEL, W. 1984. *The Complete Guide to Software Testing,* Collins.

HIERONS, R. 1992. Software testing from formal specification. Ph.D. Thesis, Brunel University, UK.

HOFFMAN, D. M. AND STROOPER, P. 1991. Automated module testing in Prolog. *IEEE Trans. Softw. Eng. 17,* 9 (Sept.), 934–943.

HORGAN, J. R. AND LONDON, S. 1991. Data flow coverage and the C language. In *Proceedings of TAV4* (Oct.), 87–97.

HORGAN, J. R. AND MATHUR, A. P. 1992. Assessing testing tools in research and education. *IEEE Softw.* (May), 61–69.

HOWDEN, W. E. 1975. Methodology for the generation of program test data. *IEEE Trans. Comput. 24,* 5 (May), 554–560.

HOWDEN, W. E. 1976. Reliability of the path analysis testing strategy. *IEEE Trans. Softw. Eng. SE-2,* (Sept.), 208–215.

HOWDEN, W. E. 1977. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Trans. Softw. Eng. SE-3* (July), 266–278.

HOWDEN, W. E. 1978a. Algebraic program testing. *ACTA Inf. 10,* 53–66.

HOWDEN, W. E. 1978b. Theoretical and empirical studies of program testing. *IEEE Trans. Softw. Eng. SE-4,* 4 (July), 293–298.

HOWDEN, W. E. 1978c. An evaluation of the effectiveness of symbolic testing. *Softw. Pract. Exper. 8,* 381–397.

HOWDEN, W. E. 1980a. Functional program testing. *IEEE Trans. Softw. Eng. SE-6,* 2 (March), 162–169.

HOWDEN, W. E. 1980b. Functional testing and design abstractions. *J. Syst. Softw. 1,* 307–313.

HOWDEN, W. E. 1981. Completeness criteria for testing elementary program functions. In *Proceedings of Fifth International Conference on Software Engineering* (March), 235–243.

HOWDEN, W. E. 1982a. Validation of scientific programs. *Comput. Surv. 14,* 2 (June), 193–227.

HOWDEN, W. E. 1982b. Weak mutation testing and completeness of test sets. *IEEE Trans. Softw. Eng. SE-8,* 4 (July), 371–379.

HOWDEN, W. E. 1985. The theory and practice of functional testing. *IEEE Softw.* (Sept.), 6–17.

HOWDEN, W. E. 1986. A functional approach to program testing and analysis. *IEEE Trans. Softw. Eng. SE-12,* 10 (Oct.), 997–1005.

HOWDEN, W. E. 1987. Functional program testing and analysis. McGraw-Hill, New York.

HUTCHINS, M., FOSTER, H., GORADIA, T., AND OSTRAND, T. 1994. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of 16th IEEE International Conference on Software Engineering* (May).

INCE, D. C. 1987. The automatic generation of test data. *Comput. J. 30,* 1, 63–69.

INCE, D. C. 1991. Software testing. In *Software Engineer's Reference Book,* J. A. McDermid, Ed., Butterworth-Heinemann (Chapter 19).

KARASIK, M. S. 1985. Environmental testing techniques for software certification. *IEEE Trans. Softw. Eng. SE-11,* 9 (Sept.), 934–938.

KEMMERER, R. A. 1985. Testing formal specifications to detect design errors. *IEEE Trans. Softw. Eng. SE-11,* 1 (Jan.), 32–43.

KERNIGHAN, B. W. AND PLAUGER, P. J. 1981. *Software Tools in Pascal,* Addison-Wesley, Reading, MA.

KING, K. N. AND OFFUTT, A. J. 1991. A FORTRAN language system for mutation-based software testing. *Softw. Pract. Exper. 21,* 7 (July), 685–718.

KOREL, B., WEDDE, H., AND FERGUSON, R. 1992. Dynamic method of test data generation for distributed software. *Inf. Softw. Tech. 34,* 8 (Aug.), 523–532.

KOSARAJU, S. 1974. Analysis of structured programs. *J. Comput. Syst. Sci. 9,* 232–255.

KRANTZ, D. H., LUCE, R. D., SUPPES, P., AND TVERSKY, A. 1971. *Foundations of Measurement, Vol. 1: Additive and Polynomial Representations.* Academic Press, New York.

KRAUSER, E. W., MATHUR, A. P., AND REGO, V. J. 1991. High performance software testing on SIMD machines. *IEEE Trans. Softw. Eng. SE-17,* 5 (May), 403–423.

LASKI, J. 1989. Testing in the program development cycle. *Softw. Eng. J.* (March), 95–106.

LASKI, J. AND KOREL, B. 1983. A data flow oriented program testing strategy. *IEEE Trans. Softw. Eng. SE-9,* (May), 33–43.

LASKI, J., SZERMER, W., AND LUCZYCKI, P. 1993. Dynamic mutation testing in integrated regression analysis. In *Proceedings of 15th International Conference on Software Engineering* (May), 108–117.

LAUTERBACH, L. AND RANDALL, W. 1989. Experimental evaluation of six test techniques. In *Proceedings of COMPASS 89* (Washington, DC, June), 36–41.

LEVENDEL, Y. 1991. Improving quality with a manufacturing process. *IEEE Softw.* (March), 13–25.

LINDQUIST, T. E. AND JENKINS, J. R. 1987. Test case generation with IOGEN. In *Proceedings of the 20th Annual Hawaii International Conference on System Sciences,* 478–487.

LITTLEWOOD, B. AND STRIGINI, L. 1993. Validation of ultra-high dependability for software-based systems. *C ACM 36,* 11 (Nov.), 69–80.

LIU, L.-L. AND ROBSON, D. J. 1989. Symbolic evaluation in software testing, the final report. Computer Science Tech. Rep. 10/89, School of Engineering and Applied Science, University of Durham, June.

LUCE, R. D., KRANTZ, D. H., SUPPES, P., AND TVERSKY, A. 1990. *Foundations of Measurement, Vol. 3: Representation, Axiomatization, and Invariance.* Academic Press, San Diego.

MALAIYA, Y. K., VONMAYRHAUSER, A., AND SRIMANI, P. K. 1993. An examination of fault exposure ratio. *IEEE Trans. Softw. Eng. 19,* 11, 1087–1094.

MARICK, B. 1991. The weak mutation hypothesis. In *Proceedings of SIGSOFT Symposium on Software Testing, Analysis, and Verification 4* (Oct.), 190–199.

MATHUR, A. P. 1991. Performance, effectiveness, and reliability issues in software testing. In *Proceedings of the 15th Annual International Computer Software and Applications Conference* (Tokyo, Sept.), 604–605.

MARSHALL, A. C. 1991. A Conceptual model of software testing. *J. Softw. Test. Ver. Rel. 1,* 3 (Dec.), 5–16.

MCCABE, T. J. 1976. A complexity measure. *IEEE Trans. Softw. Eng. SE-2,* 4, 308–320.

MCCABE, T. J. (ED.) 1983. *Structured Testing.* IEEE Computer Society Press, Los Alamitos, CA.

MCCABE, T. J. AND SCHULMEYER, G. G. 1985. System testing aided by structured analysis: A practical experience. *IEEE Trans. Softw. Eng. SE-11,* 9 (Sept.), 917–921.

MCMULLIN, P. R. AND GANNON, J. D. 1983. Combining testing with specifications: A case study. *IEEE Trans. Softw. Eng. SE-9,* 3 (May), 328–334.

MEEK, B. AND SIU, K. K. 1988. The effectiveness of error seeding. Alvey Project SE/064: Quality evaluation of programming language processors, Report No. 2, Computing Centre, King's College London, Oct.

MILLER, E. AND HOWDEN, W. E. 1981. *Tutorial: Software Testing and Validation Techniques,* (2nd ed.). IEEE Computer Society Press, Los Alamitos, CA.

MILLER, K. W., MORELL, L. J., NOONAN, R. E., PARK, S. K., NICOL, D. M., MURRILL, B. W., AND VOAS, J. M. 1992. Estimating the probability of failure when testing reveals no failures. *IEEE Trans. Softw. Eng. 18,* 1 (Jan.), 33–43.

MORELL, L. J. 1990. A theory of fault-based testing. *IEEE Trans. Softw. Eng. 16,* 8 (Aug.), 844–857.

MYERS, G. J. 1977. An extension to the cyclomatic measure of program complexity. *SIGPLAN No. 12,* 10, 61–64.

MYERS, G. J. 1979. *The Art of Software Testing.* John Wiley and Sons, New York.

MYERS, J. P., JR. 1992. The complexity of software testing. *Softw. Eng. J.* (Jan.), 13–24.

NTAFOS, S. C. 1984. An evaluation of required element testing strategies. In *Proceedings of the Seventh International Conference on Software Engineering,* 250–256.

NTAFOS, S. C. 1984. On required element testing. *IEEE Trans. Softw. Eng. SE-10,* 6 (Nov.), 795–803.

NTAFOS, S. C. 1988. A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng. SE-14* (June), 868–874.

OFFUTT, A. J. 1989. The coupling effect: Fact or fiction. In *Proceedings of SIGSOFT Symposium on Software Testing, Analysis, and Verification 3* (Dec. 13–15), 131–140.

OFFUTT, A. J. 1992. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol. 1,* 1 (Jan.), 5–20.

OFFUTT, A. J. AND LEE, S. D. 1991. How strong is weak mutation? In *Proceedings of SIGSOFT Symposium on Software Testing, Analysis, and Verification 4* (Oct.), 200–213.

OFFUTT, A. J., ROTHERMEL, G., AND ZAPF, C. 1993. An experimental evaluation of selective mutation. In *Proceedings of 15th ICSE* (May), 100–107.

OSTERWEIL, L. AND CLARKE, L. A. 1992. A proposed testing and analysis research initiative. *IEEE Softw.* (Sept.), 89–96.

OSTRAND, T. J. AND BALCER, M. J. 1988. The category-partition method for specifying and generating functional tests. *Commun. ACM 31,* 6 (June), 676–686.

OSTRAND, T. J. AND WEYUKER, E. J. 1991. Dataflow-based test adequacy analysis for languages with pointers. In *Proceedings of SIGSOFT Symposium on Software Testing, Analysis, and Verification 4,* (Oct.), 74–86.

OULD, M. A. AND UNWIN, C., EDS. 1986. *Testing in Software Development.* Cambridge University Press, New York.

PAIGE, M. R. 1975. Program graphs, an algebra, and their implication for programming. *IEEE Trans. Softw. Eng. SE-1,* 3, (Sept.), 286–291.

PAIGE, M. R. 1978. An analytical approach to software testing. In *Proceedings COMPSAC'78,* 527–532.

PANDI, H. D., RYDER, B. G., AND LANDI, W. 1991. Interprocedural Def-Use associations in C programs. In *Proceedings of SIGSOFT Symposium on Software Testing, Analysis, and Verification 4,* (Oct.), 139–153.

PARRISH, A. AND ZWEBEN, S. H. 1991. Analysis and refinement of software test data adequacy properties. *IEEE Trans. Softw. Eng. SE-17,* 6 (June), 565–581.

PARRISH, A. S. AND ZWEBEN, S. H. 1993. Clarifying some fundamental concepts in software testing. *IEEE Trans. Softw. Eng. 19,* 7 (July), 742–746.

PETSCHENIK, N. H. 1985. Practical priorities in system testing. *IEEE Softw.* (Sept.), 18–23.

PIWOWARSKI, P., OHBA, M., AND CARUSO, J. 1993. Coverage measurement experience during function testing. In *Proceedings of the 15th ICSE* (May), 287–301.

PODGURSKI, A. AND CLARKE, L. 1989. The implications of program dependences for software testing, debugging and maintenance. In *Proceedings of SIGSOFT Symposium on Software Testing, Analysis, and Verification 3,* (Dec.), 168–178.

PODGURSKI, A. AND CLARKE, L. A. 1990. A formal model of program dependences and its implications for software testing, debugging and maintenance. *IEEE Trans. Softw. Eng. 16,* 9 (Sept.), 965–979.

PRATHER, R. E. AND MYERS, J. P. 1987. The path prefix software testing strategy. *IEEE Trans. Softw. Eng. SE-13,* 7 (July).

PROGRAM ANALYSIS LTD., UK. 1992. Testbed technical description. May.

RAPPS, S. AND WEYUKER, E. J. 1985. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng. SE-11,* 4 (April), 367–375.

RICHARDSON, D. J. AND CLARKE, L. A. 1985. Partition analysis: A method combining testing and verification. *IEEE Trans. Softw. Eng. SE-11,* 12 (Dec.), 1477–1490.

RICHARDSON, D. J., AHA, S. L., AND O'MALLEY, T. O. 1992. Specification-based test oracles for reactive systems. In *Proceedings of 14th International Conference on Software Engineering* (May), 105–118.

RICHARDSON, D. J. AND THOMPSON, M. C. 1988. The RELAY model of error detection and its application. In *Proceedings of SIGSOFT Symposium on Software Testing, Analysis, and Verification 2* (July).

RICHARDSON, D. J. AND THOMPSON, M. C. 1993. An analysis of test data selection criteria using the relay model of fault detection. *IEEE Trans. Softw. Eng. 19,* 6, 533–553.

RIDDELL, I. J., HENNELL, M. A., WOODWARD, M. R., AND HEDLEY, D. 1982. Practical aspects of program mutation. Tech. Rep., Dept. of Computational Science, University of Liverpool, UK.

ROBERTS, F. S. 1979. Measurement Theory, Encyclopedia of Mathematics and Its Applications, Vol. 7. Addison-Wesley, Reading, MA.

ROE, R. P. AND ROWLAND, J. H. 1987. Some theory concerning certification of mathematical subroutines by black box testing. *IEEE Trans. Softw. Eng. SE-13,* 6 (June), 677–682.

ROUSSOPOULOS, N. AND YEH, R. T. 1985. SEES: A software testing environment support system. *IEEE Trans. Softw. Eng. SE-11,* 4, (April), 355–366.

RUDNER, B. 1977. Seeding/tagging estimation of software errors: Models and estimates. Rome Air Development Centre, Rome, NY, RADC-TR-77-15, also AD-A036 655.

SARIKAYA, B., BOCHMANN, G. V., AND CERNY, E. 1987. A test design methodology for protocol testing. *IEEE Trans. Softw. Eng. SE-13,* 5 (May), 518–531.

SHERER, S. A. 1991. A cost-effective approach to testing. *IEEE Softw.* (March), 34–40.

SOFTWARE RESEARCH. 1992. *Software TestWorks—Software Testers Workbench System.* Software Research, Inc.

SOLHEIM, J. A. AND ROWLAND, J. H. 1993. An empirical-study of testing and integration strategies using artificial software systems. *IEEE Trans. Softw. Eng. 19,* 10, 941–949.

STOCKS, P. A. AND CARRINGTON, D. A. 1993. Test templates: A specification-based testing framework. In *Proceedings of 15th International Conference on Software Engineering* (May), 405–414.

SU, J. AND RITTER, P. R. 1991. Experience in testing the Motif interface. *IEEE Softw.* (March), 26–33.

SUPPES, P., KRANTZ, D. H., LUCE, R. D., AND TVERSKY, A. 1989. *Foundations of Measurement, Vol. 2: Geometrical, Threshold, and Probabilistic Representations.* Academic Press, San Diego.

TAI, K.-C. 1993. Predicate-based test generation for computer programs. In *Proceedings of 15th International Conference on Software Engineering* (May), 267–276.

TAKAHASHI, M. AND KAMAYACHI, Y. 1985. An empirical study of a model for program error prediction. IEEE, 330–336.

THAYER, R., LIPOW, M., AND NELSON, E. 1978. *Software Reliability.* North-Holland.

TSAI, W. T., VOLOVIK, D., AND KEEFE, T. F. 1990. Automated test case generation for programs specified by relational algebra queries. *IEEE Trans. Softw. Eng. 16,* 3 (March), 316–324.

TSOUKALAS, M. Z., DURAN, J. W., AND NTAFOS, S. C. 1993. On some reliability estimation problems in random and partition testing. *IEEE Trans. Softw. Eng. 19,* 7 (July), 687–697.

URAL, H. AND YANG, B. 1988. A structural test selection criterion. *Inf. Process. Lett. 28,* 3 (July), 157–163.

URAL, H. AND YANG, B. 1993. Modeling software for accurate data flow representation. In *Proceedings of 15th International Conference on Software Engineering* (May), 277–286.

VALIANT, L. C. 1984. A theory of the learnable. *Commun. ACM 27,* 11, 1134–1142.

VOAS, J., MORRELL, L., AND MILLER, K. 1991. Predicting where faults can hide from testing. *IEEE Softw.* (March), 41–48.

WEISER, M. D., GANNON, J. D., AND MCMULLIN, P. R. 1985. Comparison of structural test coverage metrics. *IEEE Softw.* (March), 80–85.

WEISS, S. N. AND WEYUKER, E. J. 1988. An extended domain-based model of software reliability. *IEEE Trans. Softw. Eng. SE-14,* 10 (Oct.), 1512–1524.

WEYUKER, E. J. 1979a. The applicability of program schema results to programs. *Int. J. Comput. Inf. Sci. 8,* 5, 387–403.

WEYUKER, E. J. 1979b. Translatability and decidability questions for restricted classes of program schema. *SIAM J. Comput. 8,* 5, 587–598.

WEYUKER, E. J. 1982. On testing non-testable programs. *Comput. J. 25,* 4, 465–470.

WEYUKER, E. J. 1983. Assessing test data adequacy through program inference. *ACM Trans. Program. Lang. Syst. 5,* 4, (Oct.), 641–655.

WEYUKER, E. J. 1986. Axiomatizing software test data adequacy. *IEEE Trans. Softw. Eng. SE-12,* 12, (Dec.), 1128–1138.

WEYUKER, E. J. 1988a. The evaluation of program-based software test data adequacy criteria. *Commun. ACM 31,* 6, (June), 668–675.

WEYUKER, E. J. 1988b. Evaluating software complexity measures. *IEEE Trans. Softw. Eng. SE-14,* 9, (Sept.), 1357–1365.

WEYUKER, E. J. 1988c. An empirical study of the complexity of data flow testing. In *Proceedings of SIGSOFT Symposium on Software Testing, Analysis, and Verification 2* (July), 188–195.

WEYUKER, E. J. 1993. More experience with data-flow testing. *IEEE Trans. Softw. Eng. 19,* 9, 912–919.

WEYUKER, E. J. AND DAVIS, M. 1983. A formal notion of program-based test data adequacy. *Inf. Cont. 56,* 52–71.

WEYUKER, E. J. AND JENG, B. 1991. Analyzing partition testing strategies. *IEEE Trans. Softw. Eng. 17,* 7 (July), 703–711.

WEYUKER, E. J. AND OSTRAND, T. J. 1980. Theories of program testing and the application of revealing sub-domains. *IEEE Trans. Softw. Eng. SE-6,* 3 (May), 236–246.

WHITE, L. J. 1981. Basic mathematical definitions and results in testing. In *Computer Program Testing,* B. Chandrasekaran and S. Radicchi, Eds., North-Holland, 13–24.

WHITE, L. J. AND COHEN, E. I. 1980. A domain strategy for computer program testing. *IEEE Trans. Softw. Eng. SE-6,* 3 (May), 247–257.

WHITE, L. J. AND WISZNIEWSKI, B. 1991. Path testing of computer programs with loops using a tool for simple loop patterns. *Softw. Pract. Exper. 21,* 10 (Oct.).

WICHMANN, B. A. 1993. Why are there no measurement standards for software testing? *Comput. Stand. Interfaces 15,* 4, 361–364.

WICHMANN, B. A. AND COX, M. G. 1992. Problems and strategies for software component testing standards. *J. Softw. Test. Ver. Rel. 2,* 167–185.

WILD, C., ZEIL, S., CHEN, J., AND FENG, G. 1992. Employing accumulated knowledge to refine test cases. *J. Softw. Test. Ver. Rel. 2,* 2 (July), 53–68.

WISZNIEWSKI, B. W. 1985. Can domain testing overcome loop analysis? IEEE, 304–309.

WOODWARD, M. R. 1991. Concerning ordered mutation testing of relational operators. *J. Softw. Test. Ver. Rel. 1,* 3 (Dec.), 35–40.

WOODWARD, M. R. 1993. Errors in algebraic specifications and an experimental mutation testing tool. *Softw. Eng. J.* (July), 211–224.

WOODWARD, M. R. AND HALEWOOD, K. 1988. From weak to strong—dead or alive? An analysis of some mutation testing issues. In *Proceedings of Second Workshop on Software Testing, Verification and Analysis* (July) 152–158.

WOODWARD, M. R., HEDLEY, D., AND HENNEL, M. A. 1980. Experience with path analysis and testing of programs. *IEEE Trans. Softw. Eng. SE-6,* 5 (May), 278–286.

WOODWARD, M. R., HENNEL, M. A., AND HEDLEY, D. 1980. A limited mutation approach to program testing. Tech. Rep. Dept. of Computational Science, University of Liverpool.

YOUNG, M. AND TAYLOR, R. N. 1988. Combining static concurrency analysis with symbolic execution. *IEEE Trans. Softw. Eng. SE-14,* 10 (Oct.), 1499–1511.

ZEIL, S. J. 1983. Testing for perturbations of program statements. *IEEE Trans. Softw. Eng. SE-9,* 3, (May), 335–346.

ZEIL, S. J. 1984. Perturbation testing for computation errors. In *Proceedings of Seventh International Conference on Software Engineering* (Orlando, FL), 257–265.

ZEIL, S. J. 1989. Perturbation techniques for detecting domain errors. *IEEE Trans. Softw. Eng. 15,* 6 (June), 737–746.

ZEIL, S. J., AFIFI, F. H., AND WHITE, L. J. 1992. Detection of linear errors via domain testing. *ACM Trans. Softw. Eng. Methodol. 1,* 4, (Oct.), 422–451.

ZHU, H. 1995a. Axiomatic assessment of control flow based software test adequacy criteria. *Softw. Eng. J.* (Sept.), 194–204.

ZHU, H. 1995b. An induction theory of software testing. *Sci. China 38* (Supp.) (Sept.), 58–72.

ZHU, H. 1996a. A formal analysis of the subsume relation between software test adequacy criteria. *IEEE Trans. Softw. Eng. 22,* 4 (April), 248–255.

ZHU, H. 1996b. A formal interpretation of software testing as inductive inference. *J. Softw. Test. Ver. Rel. 6* (July), 3–31.

ZHU, H. AND HALL, P. A. V. 1992a. Test data adequacy with respect to specifications and related properties. Tech. Rep. 92/06, Department of Computing, The Open University, UK, Jan.

ZHU, H. AND HALL, P. A. V. 1992b. Testability of programs: Properties of programs related to test data adequacy criteria. Tech. Rep. 92/05, Department of Computing, The Open University, UK, Jan.

ZHU, H. AND HALL, P. A. V. 1993. Test data adequacy measurements. *Softw. Eng. J. 8,* 1 (Jan.), 21–30.

ZHU, H., HALL, P. A. V., AND MAY, J. 1992. Inductive inference and software testing. *J. Softw. Test. Ver. Rel. 2,* 2 (July), 69–82.

ZHU, H., HALL, P. A. V., AND MAY, J. 1994. Understanding software test adequacy—an axiomatic and measurement approach. In *Mathematics of Dependable Systems, Proceedings of IMA First Conference* (Sept., London), Oxford University Press, Oxford.

ZWEBEN, S. H. AND GOURLAY, J. S. 1989. On the adequacy of Weyuker's test data adequacy axioms. *IEEE Trans. Softw. Eng. SE-15,* 4, (April), 496–501.

Received November 1994; revised March 1996; accepted October 1996